

# **Library Compiler™ User Guide: Modeling Timing, Signal Integrity, and Power in Technology Libraries**

---

Version Z-2007.03, March 2007

## Comments?

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

**SYNOPSYS®**

# Copyright Notice and Proprietary Information

Copyright © 2007 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

## Right to Copy Documentation

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

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

## Destination Control Statement

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

## Disclaimer

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

## Registered Trademarks (®)

Synopsys, AMPS, Cadabra, CATS, CRITIC, CSim, Design Compiler, DesignPower, DesignWare, EPIC, Formality, HSIM, HSPICE, iN-Phase, in-Sync, Leda, MAST, ModelTools, NanoSim, OpenVera, PathMill, Photolynx, Physical Compiler, PrimeTime, SiVL, SNUG, SolvNet, System Compiler, TetraMAX, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

## Trademarks (™)

AFGen, Apollo, Astro, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, Columbia, Columbia-CE, Cosmos, CosmosEnterprise, CosmosLE, CosmosScope, CosmosSE, DC Expert, DC Professional, DC Ultra, Design Analyzer, Design Vision, DesignerHDL, Direct Silicon Access, Discovery, Encore, Galaxy, HANEX, HDL Compiler, Hercules, plus Hierarchical Optimization Technology, HSIM<sup>plus</sup>, HSPICE-Link, i-Virtual Stepper, iN-Tandem, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Magellan, Mars, Mars-Xtalk, Milkyway, ModelSource, Module Compiler, Planet, Planet-PL, Polaris, Power Compiler, Raphael, Raphael-NES, Saturn, Scirocco, Scirocco-i, Star-RCXT, Star-SimXT, Taurus, TSUPREM-4, VCS Express, VCSi, VHDL Compiler, VirSim, and VMC are trademarks of Synopsys, Inc.

## Service Marks (SM)

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

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

ARM and AMBA are registered trademarks of ARM Limited.

Saber is a registered trademark of SabreMark Limited Partnership and is used under license.

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

# Contents

---

What's New in This Release .....	xxvi
About This Guide.....	xxvi
Customer Support.....	xxx
1. Modeling Power and Electromigration	
Modeling Power Terminology .....	1-3
Static Power .....	1-3
Dynamic Power .....	1-3
Internal Power .....	1-4
Switching Power .....	1-5
Switching Activity .....	1-6
Modeling Cells for Power .....	1-7
Modeling for Leakage Power .....	1-9
Representing Leakage Power Information .....	1-10
cell_leakage_power Simple Attribute.....	1-10
Using the leakage_power Group for a Single Value .....	1-10
when Simple Attribute .....	1-11

value Simple Attribute .....	1-12
Using the leakage_power Group for a Polynomial.....	1-13
Specifying leakage power in a power Group .....	1-14
leakage_power_unit Simple Attribute .....	1-16
default_cell_leakage_power Simple Attribute .....	1-17
default_leakage_power_density Simple Attribute .....	1-18
Environmental Derating Factors Attributes .....	1-22
 Modeling for Internal and Switching Power .....	1-23
Modeling Internal Power Lookup Tables .....	1-26
Clock Pin Power .....	1-28
Output Pin Power .....	1-28
Calculating Switching Power .....	1-31
 Creating a Scalable Polynomial Power Model.....	1-32
Polynomial Representation .....	1-33
Model Description .....	1-34
 Representing Internal Power Information .....	1-36
Specifying the Power Model.....	1-36
Using Lookup Table Templates .....	1-37
power_lut_template Group .....	1-37
Scalar power_lut_template Group .....	1-40
Using Scalable Polynomial Power Modeling .....	1-41
power_poly_template Group .....	1-41
 Defining Internal Power Groups .....	1-45
Naming Power Relationships, Using the internal_power Group.....	1-45

Power Relationship Between a Single Pin and a Single Related Pin . . . . .	1-46
Power Relationships Between a Single Pin and Multiple Related Pins . . . . .	1-47
Power Relationships Between a Bundle and a Single Related Pin . . . . .	1-48
Power Relationships Between a Bundle and Multiple Related Pins . . . . .	1-49
Power Relationships Between a Bus and a Single Related Pin . . . . .	1-51
Power Relationships Between a Bus and Multiple Related Pins . . . . .	1-53
Power Relationships Between a Bus and Related Bus Pins . . . . .	1-55
internal_power Group . . . . .	1-56
equal_or_opposite_output Simple Attribute . . . . .	1-57
falling_together_group Simple Attribute . . . . .	1-57
power_level Simple Attribute . . . . .	1-58
related_pin Simple Attribute . . . . .	1-61
rising_together_group Simple Attribute . . . . .	1-62
switching_interval Simple Attribute . . . . .	1-63
switching_together_group Simple Attribute . . . . .	1-64
when Simple Attribute . . . . .	1-65
fall_power Group . . . . .	1-66
power Group . . . . .	1-69
rise_power Group . . . . .	1-71
Power-Scaling Factors . . . . .	1-75
Internal Power Examples . . . . .	1-76
One-Dimensional Power Lookup Table . . . . .	1-76
Two-Dimensional Power Lookup Table . . . . .	1-78

Three-Dimensional Power Lookup Table .....	1-80
Modeling Libraries With Multiple Power Supplies .....	1-82
Power Supply Information .....	1-82
Defining Power Supply Groups .....	1-84
power_rail Complex Attribute .....	1-84
default_power_rail Simple Attribute .....	1-85
Defining Cells With Multiple Power Supplies .....	1-85
rail_connection Complex Attribute .....	1-85
input_signal_level and output_signal_level Simple Attributes .....	1-86
Multiple-Power-Supply Library Requirements .....	1-88
Example Libraries With Multiple Power Supplies .....	1-89
Input and Output Voltage Information .....	1-93
input_voltage Group .....	1-94
output_voltage Group .....	1-95
Modeling Libraries With Integrated Clock-Gating Cells .....	1-96
What Clock Gating Does .....	1-96
Looking at a Gated Clock .....	1-98
Using an Integrated Clock-Gating Cell .....	1-100
Setting Pin Attributes for an Integrated Cell .....	1-101
Modeling Electromigration .....	1-111
Overview .....	1-111
Controlling Electromigration .....	1-113
em_lut_template Group .....	1-115
electromigration Group .....	1-118
Scalable Polynomial Electromigration Model .....	1-123

<b>2. Advanced Low Power Modeling</b>	
Power and Ground (PG) Pins . . . . .	2-3
Syntax . . . . .	2-3
Library-Level Attributes . . . . .	2-5
Cell-Level Attributes . . . . .	2-5
Pin-Level Attributes . . . . .	2-7
Library Checking Rules . . . . .	2-10
Level Shifter and Isolation Cells in a Multivoltage Design . . . . .	2-11
Operating Voltages . . . . .	2-12
Functionality . . . . .	2-12
Syntax . . . . .	2-13
Cell-Level Attributes . . . . .	2-13
Pin-Level Attributes . . . . .	2-16
Level-Shifter Modeling Examples . . . . .	2-17
Simple Buffer-Type Level-Shifter Example . . . . .	2-17
Overdrive Level-Shifter Example . . . . .	2-19
Enable Level-Shifter Example . . . . .	2-22
Enable Isolation Cell Example . . . . .	2-25
Library Checking Rules . . . . .	2-28
Switch Cell Modeling . . . . .	2-31
Coarse-Grain Switch Cells . . . . .	2-32
Syntax . . . . .	2-33
Cell-Level Attribute . . . . .	2-35
Pin-Level Attributes . . . . .	2-35
pg_pin Group . . . . .	2-36
lu_table_template Group . . . . .	2-36

dc_current Group . . . . .	2-37
Switch-Cell Modeling Examples . . . . .	2-38
Simple Coarse-Grain Switch Cell Example . . . . .	2-38
Complex Coarse-Grain Switch Cell Example . . . . .	2-40
Complex Coarse-Grain Switch Cell With an Internal Switch Pin Example . . . . .	2-42
Library Checking Rules . . . . .	2-45
<b>3. Delay Models</b>	
CMOS Generic Delay Model . . . . .	3-3
Total Delay Equation . . . . .	3-3
Total Delay Scaling . . . . .	3-5
Slope Delay . . . . .	3-7
Intrinsic Delay . . . . .	3-8
Transition Delay . . . . .	3-9
Connect Delay . . . . .	3-10
Interconnect Delay . . . . .	3-15
Delay Calculation Example . . . . .	3-17
CMOS Nonlinear Delay Model . . . . .	3-20
Total Delay Equation . . . . .	3-20
Cell Delay . . . . .	3-23
Propagation Delay . . . . .	3-24
Transition Delay . . . . .	3-24
Connect Delay . . . . .	3-25
CMOS Nonlinear Delay Model Calculation . . . . .	3-25
Process, Voltage, and Temperature Scaling . . . . .	3-29

Scalable Polynomial Delay Model .....	3-31
Polynomial Representation .....	3-32
Model Description .....	3-33
Scalable Polynomial Calculation .....	3-35
Using the Conversion Feature .....	3-36
CMOS Piecewise Linear Delay Model. ....	3-37
Total Delay Equation .....	3-38
Total Delay Scaling .....	3-40
Piecewise Linear Model.....	3-41
Intrinsic Delay .....	3-43
Slope Delay .....	3-44
Transition Delay .....	3-45
Connect Delay.....	3-47
Interconnect Delay.....	3-52
Delay Calculation Example .....	3-54
Delay Calculation Module (DCM) Delay Model .....	3-56
 4. Timing Arcs	
Understanding Timing Arcs.....	4-3
Combinational Timing Arcs .....	4-4
Sequential Timing Arcs .....	4-4
Modeling Method Alternatives.....	4-6
Defining timing Groups .....	4-7
Naming Timing Arcs, Using the timing Group .....	4-8

Timing Arc Between a Single Pin and a Single Related Pin . . . . .	4-9
Timing Arcs Between a Pin and Multiple Related Pins . . . . .	4-10
Timing Arcs Between a Bundle and a Single Related Pin . . . . .	4-11
Timing Arcs Between a Bundle and Multiple Related Pins . . . . .	4-12
Timing Arcs Between a Bus and a Single Related Pin . . . . .	4-14
Timing Arcs Between a Bus and Multiple Related Pins . . . . .	4-16
Timing Arcs Between a Bus and Related Bus Pins . . . . .	4-17
Timing Arcs Between a Bus and a Related Bus Equivalent . . . . .	4-19
Delay Models . . . . .	4-21
delay_model Attribute. . . . .	4-23
Defining the CMOS Nonlinear Delay Model Template . . . . .	4-23
Creating Lookup Tables . . . . .	4-28
Defining the Scalable Polynomial Delay Model Template. . . . .	4-30
timing Group Attributes . . . . .	4-32
related_pin Simple Attribute . . . . .	4-34
related_bus_pins Simple Attribute . . . . .	4-35
timing_sense Simple Attribute . . . . .	4-36
timing_type Simple Attribute. . . . .	4-37
mode Complex Attribute . . . . .	4-46
Describing Three-State Timing Arcs . . . . .	4-51
Describing Three-State-Disable Timing Arcs. . . . .	4-51
Describing Three-State-Enable Timing Arcs . . . . .	4-52

Describing Edge-Sensitive Timing Arcs .....	4-54
Describing Preset and Clear Timing Arcs .....	4-55
Describing Preset Arcs .....	4-56
Describing Clear Arcs .....	4-58
Describing Clock Insertion Delay .....	4-60
Describing Intrinsic Delay .....	4-61
In the CMOS Generic Delay Model .....	4-62
intrinsic_rise Simple Attribute .....	4-62
intrinsic_fall Simple Attribute .....	4-62
In the CMOS Piecewise Linear Delay Model .....	4-63
In the CMOS Nonlinear Delay Model.....	4-63
In the Scalable Polynomial Delay Model .....	4-64
Describing Transition Delay .....	4-64
In the CMOS Generic Delay Model .....	4-64
rise_resistance Simple Attribute.....	4-64
fall_resistance Simple Attribute .....	4-65
In the CMOS Piecewise Linear Delay Model .....	4-66
rise_delay_intercept Complex Attribute .....	4-66
fall_delay_intercept Complex Attribute .....	4-66
rise_pin_resistance Complex Attribute .....	4-66
fall_pin_resistance Complex Attribute .....	4-67
In the CMOS Nonlinear Delay Model.....	4-68
Defining Delay Arcs With Lookup Tables .....	4-68
Modeling Transition Time Degradation .....	4-79
Modeling Load Dependency .....	4-83
In the CMOS Nonlinear Delay Model.....	4-83

In the CMOS Scalable Polynomial Delay Model .....	4-85
Describing Slope Sensitivity .....	4-87
In the CMOS Generic Delay Model and Piecewise Linear Delay Model .....	4-88
slope_rise Simple Attribute .....	4-88
slope_fall Simple Attribute .....	4-88
Describing State-Dependent Delays .....	4-88
when Simple Attribute .....	4-89
sdf_cond Simple Attribute .....	4-92
Setting Setup and Hold Constraints .....	4-94
In the CMOS Generic Delay Model and Piecewise Linear Delay Model .....	4-96
Setup Constraints .....	4-96
Hold Constraints .....	4-98
In the CMOS Nonlinear Delay Model .....	4-100
rise_constraint and fall_constraint Groups .....	4-100
In the Scalable Polynomial Delay Model .....	4-103
Identifying Interdependent Setup and Hold Constraints .....	4-104
Setting Nonsequential Timing Constraints .....	4-104
Setting Recovery and Removal Timing Constraints .....	4-107
Recovery Constraints .....	4-107
Removal Constraint .....	4-109
Setting No-Change Timing Constraints .....	4-111
In the CMOS Generic Delay Model .....	4-114
In the CMOS Nonlinear Delay Model .....	4-115
In the CMOS Scalable Polynomial Delay Model .....	4-116

Setting Skew Constraints .....	4-118
Setting Conditional Timing Constraints .....	4-120
when and sdf_cond Simple Attributes .....	4-121
when_start Simple Attribute .....	4-121
sdf_cond_start Simple Attribute .....	4-122
when_end Simple Attribute .....	4-122
sdf_cond_end Simple Attribute .....	4-123
sdf_edges Simple Attribute .....	4-123
min_pulse_width Group .....	4-124
Example .....	4-124
constraint_high and constraint_low Simple Attributes .....	4-124
when and sdf_cond Simple Attributes .....	4-125
minimum_period Group .....	4-126
Example .....	4-126
constraint Simple Attribute .....	4-126
when and sdf_cond Simple Attributes .....	4-127
Using Conditional Attributes With No-Change Constraints .....	4-127
Generating an SDF File .....	4-128
Timing Arc Restrictions .....	4-129
Impossible Transitions .....	4-129
Examples of Libraries Using Delay Models .....	4-131
CMOS Generic Delay Model .....	4-131
CMOS Piecewise Linear Delay Model .....	4-133
CMOS Nonlinear Delay Model .....	4-137
CMOS Scalable Polynomial Delay Model .....	4-140
Clock Insertion Delay Example .....	4-145

Describing a Transparent Latch Clock Model . . . . .	4-146
Checking Timing Library Data . . . . .	4-149
Driver Waveform Support . . . . .	4-151
Syntax . . . . .	4-152
Library-Level Tables, Attributes, and Variables . . . . .	4-153
normalized_voltage Variable . . . . .	4-153
normalized_driver_waveform Group . . . . .	4-154
driver_waveform_name Attribute . . . . .	4-154
Cell-level Attributes . . . . .	4-155
driver_waveform Attribute . . . . .	4-155
driver_waveform_rise and driver_waveform_fall Attributes .	4-156
Pin-Level Attributes . . . . .	4-157
driver_waveform Attribute . . . . .	4-157
driver_waveform_rise and driver_waveform_fall Attributes .	4-157
Screening Rules . . . . .	4-157
Example . . . . .	4-159
Sensitization Support . . . . .	4-160
sensitization Group . . . . .	4-161
pin_names Attribute . . . . .	4-162
vector Attribute . . . . .	4-162
Cell-Level Attributes . . . . .	4-163
sensitization_master Attribute . . . . .	4-163
pin_name_map Attribute . . . . .	4-163
Timing Group Attributes . . . . .	4-164
sensitization_master Attribute . . . . .	4-164
pin_name_map Attribute . . . . .	4-164
wave_rise and wave_fall Attributes . . . . .	4-165

wave_rise_sampling_index and wave_fall_sampling_index Attributes . . . . .	4-166
wave_rise_time_interval and wave_fall_time_interval Attributes . . . . .	4-167
Syntax . . . . .	4-168
Library Screener . . . . .	4-169
Using the LibChecker Utility . . . . .	4-174
 5. Composite Current Source Modeling	
Modeling Cells With Composite Current Source Information . . . . .	5-2
Representing Composite Current Source Driver Information . . . . .	5-2
Composite Current Source Lookup Table Model . . . . .	5-3
Defining the output_current_template Group. . . . .	5-3
Defining the Lookup Table Output Current Groups . . . . .	5-4
vector Group. . . . .	5-5
Representing Composite Current Source Receiver Information . . . . .	5-8
Composite Current Source Lookup Table Model . . . . .	5-8
Defining the Receiver Capacitance Group at the Pin Level . . . . .	5-9
Defining the lu_table_template Group . . . . .	5-9
Defining the Lookup Table receiver_capacitance Group . . . . .	5-10
Defining the Receiver Capacitance Groups at the Timing Level . . . . .	5-13
Defining the lu_table_template Group . . . . .	5-13
Defining the Lookup Table receiver_capacitance Groups . . . . .	5-14
Screening Checks . . . . .	5-17
Example of Composite Current Source Driver and Receiver Model . . . . .	5-20

## 6. Advanced Composite Current Source Modeling

Modeling Cells With Advanced Composite Current Source Information .....	6-2
Compact CCS Timing Model Support.....	6-2
Using the ccs_cet Utility.....	6-3
Modeling With CCS Timing Base Curves .....	6-4
Compact CCS Timing Model Syntax.....	6-7
base_curves Group .....	6-8
compact_lut_template Group .....	6-9
compact_ccs_{risefall} Group .....	6-10
Library Compiler Screener.....	6-11
CCS Timing Library Example .....	6-13
Variation-Aware Timing Modeling Support .....	6-15
Variation-Aware Compact CCS Timing Driver Model .....	6-16
The timing_based_variation Group .....	6-17
The va_compact_ccs_rise and va_compact_ccs_fall Groups .....	6-19
timing_based_variation and pin_based_variation Groups ..	6-21
Variation-Aware CCS Timing Receiver Model .....	6-24
Variation-Aware Timing Constraint Model .....	6-28
Library Compiler Screener for Timing Models .....	6-31
Variation-Aware Syntax Examples.....	6-32

## 7. Composite Current Source Signal Integrity Modeling

Composite Current Source Signal Integrity Noise Model .....	7-2
Syntax .....	7-3
Screening Rules.....	7-4

Sample CCS Noise Library .....	7-9
<b>8. Composite Current Source Power Modeling</b>	
Composite Current Source Power Modeling .....	8-2
Cell Leakage Current.....	8-3
Leakage Current Syntax Checking.....	8-4
Gate Leakage Modeling in Leakage Current.....	8-6
Intrinsic Parasitic .....	8-10
Intrinsic Parasitic Model Syntax Checking .....	8-10
Parasitics Modeling in Macro Cells .....	8-12
Dynamic Power .....	8-14
Dynamic Power Modeling in Macro Cells.....	8-15
Power and Ground Current Syntax Checking .....	8-20
Dynamic Current Syntax .....	8-21
Dynamic Current Syntax Checking .....	8-22
Composite Current Source Dynamic Power Examples .....	8-28
<b>9. Interface Timing of Complex Sequential Blocks</b>	
Interface Timing Specification .....	9-3
Sequential Cell Timing Without Interface Timing Specification .....	9-4
Supported Features of Interface Timing .....	9-5
Describing Blocks With Interface Timing .....	9-7
Using Library Compiler Syntax .....	9-7
Differences From Regular Cell Specification .....	9-8
Advanced Modeling Technology.....	9-8
model Group.....	9-9

model and short Examples . . . . .	9-10
generated_clock Group . . . . .	9-11
Generated Clock Example . . . . .	9-15
Similarities to Regular Cell Specification . . . . .	9-16
Examples of Interface Timing Relationships . . . . .	9-17
Input Setup and Hold . . . . .	9-17
Output Propagation . . . . .	9-18
Through Combinational Paths . . . . .	9-20
Interface Timing of a Complex Sequential Block . . . . .	9-21
Using Blocks With Interface Timing in Synthesis . . . . .	9-26
Interpreting Timing Relationships . . . . .	9-27
Examples Using Interface Timing Specification . . . . .	9-29
 10. Modeling Noise	
Modeling Noise Terminology . . . . .	10-2
Noise Calculation . . . . .	10-3
Noise Immunity . . . . .	10-3
Noise Propagation . . . . .	10-3
Modeling Cells for Noise . . . . .	10-4
I-V Characteristics and Drive Resistance . . . . .	10-4
Noise Immunity . . . . .	10-6
Using the Hyperbolic Model . . . . .	10-10
Noise Propagation . . . . .	10-10
Representing Noise Calculation Information . . . . .	10-12
I-V Characteristics Lookup Table Model . . . . .	10-13
iv_lut_template Group . . . . .	10-13

Defining the Lookup Table Steady-State Current Groups . . . . .	10-15
I-V Characteristics Curve Polynomial Model . . . . .	10-16
poly_template Group . . . . .	10-17
Defining Polynomial Steady-State Current Groups . . . . .	10-18
Using Steady-State Resistance Simple Attributes . . . . .	10-20
Using I-V Curves and Steady-State Resistance for tied_off Cells . . . . .	10-21
Defining tied_off Attribute Usage . . . . .	10-21
Representing Noise Immunity Information . . . . .	10-23
Noise Immunity Lookup Table Model . . . . .	10-23
noise_lut_template Group . . . . .	10-23
Defining the Noise Immunity Table Groups . . . . .	10-25
Noise Immunity Polynomial Model . . . . .	10-27
poly_template Group . . . . .	10-27
Defining the Noise Immunity Polynomial Groups . . . . .	10-29
Input Noise Width Ranges at the Pin Level . . . . .	10-30
Defining the input_noise_width Range Limits . . . . .	10-30
Defining the Hyperbolic Noise Groups . . . . .	10-32
Representing Propagated Noise Information . . . . .	10-34
Propagated Noise Lookup Table Model . . . . .	10-35
propagation_lut_template Group . . . . .	10-35
Defining the Propagated Noise Table Groups . . . . .	10-36
Propagated Noise Polynomial Model . . . . .	10-39
poly_template Group . . . . .	10-40
Defining Propagated Noise Groups for Polynomial Representation . . . . .	10-41
Examples of Modeling Noise . . . . .	10-44

Scalable Polynomial Model Noise Sample .....	10-44
Nonlinear Delay Model Library With Noise Information .....	10-52
Checking Noise Library Data.....	10-57
<b>11. Generating Library Reports</b>	
Generating Default Reports.....	11-3
Generating Specific Reports .....	11-5
Timing Reports .....	11-6
Library Defaults .....	11-6
Library-Level Templates .....	11-7
Cell-Level Data.....	11-10
Pin-Level Information .....	11-11
Delay Reports.....	11-17
Timing Arcs Report .....	11-22
Timing Label Reports.....	11-23
Timing Report Statistics .....	11-24
Noise Reports .....	11-24
Power Reports .....	11-28
Library Defaults .....	11-28
Library-Level Templates .....	11-29
Library-Level Multiple Power Supply Information .....	11-30
Cell-Level Information.....	11-32
Pin-Level Information .....	11-33
Electromigration Reports .....	11-35
Library Defaults .....	11-35
Library-Level Templates .....	11-36
Pin-Level Information .....	11-36

Functionality Reports . . . . .	11-37
Statetable Report . . . . .	11-37
FPGA Reports . . . . .	11-40
Library Defaults . . . . .	11-40
Parts Information . . . . .	11-41
Cell-Level Information. . . . .	11-41
User-Defined Data Reports . . . . .	11-42
Comprehensive Reports . . . . .	11-45

## 12. Building Environments

Library-Level Default Attributes . . . . .	12-3
Setting Default Cell Attributes . . . . .	12-3
default_cell_leakage_power Simple Attribute. . . . .	12-3
default_leakage_power_density Simple Attribute. . . . .	12-4
Setting Default Pin Attributes . . . . .	12-5
All Delay Models. . . . .	12-5
CMOS Generic Delay Model . . . . .	12-7
Piecewise Linear Delay Model . . . . .	12-8
Setting Wire Load Defaults . . . . .	12-10
default_wire_load Attribute. . . . .	12-10
Synchronizing Design and Model Modes. . . . .	12-10
default_wire_load_capacitance Attribute . . . . .	12-10
default_wire_load_resistance Attribute . . . . .	12-11
default_wire_load_area Attribute . . . . .	12-11
Setting Other Environment Defaults. . . . .	12-11
default_max_utilization Attribute . . . . .	12-11
default_min_porosity Attribute . . . . .	12-12
default_operating_conditions Attribute . . . . .	12-13

default_connection_class Attribute . . . . .	12-13
Examples of Library-Level Default Attributes . . . . .	12-13
em_temp_degradation_factor Attribute . . . . .	12-14
Defining Operating Conditions . . . . .	12-15
operating_conditions Group . . . . .	12-15
Determining Operating Conditions . . . . .	12-17
Defining Operating Conditions at Runtime . . . . .	12-18
timing_range Group . . . . .	12-18
Defining Timing Ranges at Runtime . . . . .	12-20
Defining Power Supply Cells . . . . .	12-21
power_supply group . . . . .	12-21
Defining Wire Load Groups . . . . .	12-22
wire_load Group . . . . .	12-22
Calculating Wire Area . . . . .	12-27
wire_load_table Group . . . . .	12-30
Selecting Wire Load Groups Automatically . . . . .	12-31
wire_load_from_area Attribute . . . . .	12-32
Specifying Default Wire Load Settings . . . . .	12-33
Specifying Delay Scaling Attributes . . . . .	12-35
Calculating Delay Factors . . . . .	12-36
Calculating Voltage Delay Factors . . . . .	12-36
Calculating Temperature Delay Factors . . . . .	12-37
Assigning Process Delay Factors . . . . .	12-39
Setting Combined Scaling Factor . . . . .	12-39
Intrinsic Delay Factors . . . . .	12-40

Slope Sensitivity Factors .....	12-41
Drive Capability Factors.....	12-42
Pin and Wire Capacitance Factors.....	12-43
CMOS Wire Resistance Factors .....	12-45
Pin Resistance Factors .....	12-45
Intercept Delay Factors .....	12-46
Power Scaling Factors .....	12-48
Timing Constraint Factors .....	12-49
Delay Scaling Factors Example .....	12-58
Scaling Factors for Individual Cells .....	12-59
Scaling Factors Associated With the Nonlinear Delay Model ..	12-60
Specifying Nonlinear Delay Tables.....	12-62
 13. Verifying CMOS Libraries	
Library Verification.....	13-2
Checking Library Consistency.....	13-3
Preparing a Test Circuit.....	13-4
Verifying Functionality .....	13-7
Preparing for Function Verification.....	13-7
Creating the Test-Mapping Library .....	13-8
Mapping the Test Circuit Netlist .....	13-10
Creating the Simulation Test Vectors.....	13-11
Simulating the Mapped and Original Test Netlists .....	13-12
Identifying Function Description Errors .....	13-12
Verifying Technology and VHDL Libraries Together.....	13-13
Creating the Test-Mapping Library .....	13-13

Mapping the Test Circuit Netlist . . . . .	13-14
Completing the Verification. . . . .	13-14
Verifying Timing Parameters . . . . .	13-15
Testing Parameters . . . . .	13-15
General Testing Methodology . . . . .	13-16
Debugging With the report_delay_calculation Command . . . . .	13-19
Testing Basic Cell-Timing Parameters . . . . .	13-20
Intrinsic Delays . . . . .	13-21
Resistance (Generic Delay Model). . . . .	13-21
Resistance (Piecewise Linear Delay Model) . . . . .	13-22
Capacitance . . . . .	13-23
Input Slope . . . . .	13-24
Setup . . . . .	13-25
Hold . . . . .	13-27
Rising and Falling Edges . . . . .	13-29
Area . . . . .	13-29
Ranges for Piecewise Linear Model. . . . .	13-30
Testing the Interconnect Models . . . . .	13-31
Testing Environmental Scaling. . . . .	13-33

## Index

# Preface

---

This preface includes the following sections:

- What's New in This Release
- About This Guide
- Customer Support

---

## What's New in This Release

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

To see the *Library Compiler Release Notes*,

1. Go to <http://solvnet.synopsys.com/ReleaseNotes>. (If prompted, enter your user name and password. If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.)
2. Click Library Compiler, then click the release you want in the list that appears at the bottom.

---

## About This Guide

The Library Compiler tool from Synopsys captures ASIC libraries and translates them into Synopsys internal database format for physical synthesis or into VHDL format for simulation. The Library Compiler documentation includes a three volume reference manual and a three volume user guide.

The reference manual presents the syntax of the group statements that identify the characteristics of a CMOS technology library, a symbol library, a physical library, and a VHDL library; the `lc_shell` command syntax; and the delay analysis equations for CMOS libraries.

Following is a description of each volume's contents:

- *Library Compiler Reference Manual: Technology and Symbol Libraries* provides information to be used with synthesis, test, and power tools.
- *Library Compiler Reference Manual: VHDL Libraries* provides information to be used with simulation tools.
- *Library Compiler Reference Manual: Physical Libraries* provides information required for floorplanning, RC estimation and extraction, placement, and routing.
- *Library Compiler User Guide: Methodology and Modeling Functionality in Technology Libraries* describes the Library Compiler software and explains how to build libraries and define cells.
- *Library Compiler User Guide: Modeling Timing, Signal Integrity, and Power in Technology Libraries* describes how to model power, timing, optimization, and a physical library for Library Compiler.
- *Library Compiler User Guide: Physical Libraries* describes how to develop physical 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 the Design Compiler tool from Synopsys, as well as experience in reading manufacturers' specification sheets for ASIC components.

Using Library Compiler to generate VHDL simulation libraries requires knowledge of the VHDL simulation language.

---

## **Related Publications**

For additional information about Library Compiler, see

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

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

- Design Compiler
- Formality
- FPGA Compiler II
- Module Compiler
- Physical Compiler
- Power Management
- PrimeTime and PrimeTime *SI*
- Test Automation

---

## Conventions

The following conventions are used in Synopsys documentation.

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

---

## **Customer Support**

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

---

### **Accessing SolvNet**

SolvNet includes an electronic knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. SolvNet also gives you access to a wide range of Synopsys online services including software downloads, documentation on the Web, and “Enter a Call to the Support Center.”

To access SolvNet,

1. Go to the SolvNet Web page at <http://solvnet.synopsys.com>.
2. If prompted, enter your user name and password. (If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.)

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

---

## Contacting the Synopsys Technical Support Center

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

- Open a call to your local support center from the Web by going to <http://solvnet.synopsys.com> (Synopsys user name and password required), then clicking “Enter a Call to the Support Center.”
- Send an e-mail message to your local support center.
  - E-mail [support\\_center@synopsys.com](mailto:support_center@synopsys.com) from within North America.
  - Find other local support center e-mail addresses at [http://www.synopsys.com/support/support\\_ctr](http://www.synopsys.com/support/support_ctr).
- Telephone your local support center.
  - Call (800) 245-8005 from within the continental United States.
  - Call (650) 584-4200 from Canada.
  - Find other local support center telephone numbers at [http://www.synopsys.com/support/support\\_ctr](http://www.synopsys.com/support/support_ctr).

Preface

xxxii

# 1

## 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
- Modeling for Internal and Switching Power
- Creating a Scalable Polynomial Power Model

- Representing Internal Power Information
- Defining Internal Power Groups
- Modeling Libraries With Multiple Power Supplies
- 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

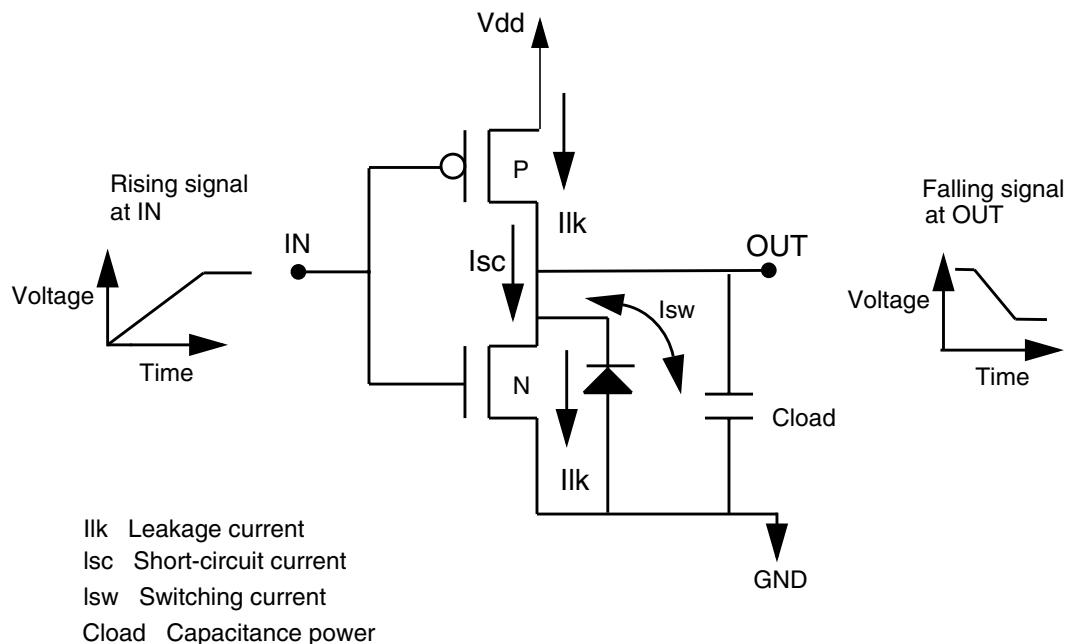
- Internal power
- Switching power

## 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 1-1** 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 1-1 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 1-1](#) 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 Reference Manual*.

---

## 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 ]

$P_{\text{Dynamic}}$

Total dynamic power dissipation of the design.

$P_{\text{Cellinternal}}$

Internal power of each cell.

$V_{dd}$

Supply voltage.

$C_{\text{Load}}$

Capacitive load of each net.

The unit for switching power and internal power is a derived unit. It is derived from the following function:

$$(\text{capacitive\_load\_unit} \cdot \text{voltage\_unit}^2) / \text{time\_unit}$$

For more information about dynamic power unit derivation, see the *Power Compiler Reference Manual*.

---

## 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 ( $I_{\text{supply}}$ ) 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 characterization effort at the expense of accuracy.

---

## Representing Leakage Power Information

In the syntax of the Library Compiler tool, you can represent leakage power information with

- The `cell_leakage_power` attribute
  - The `leakage_power` group with a single value
  - The `leakage_power` group with a polynomial
  - The associated library-level attributes that specify scaling factors, units, and a default value 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, Library Compiler assigns the value of the `default_cell_leakage_power` attribute.

#### **Syntax**

```
cell_leakage_power : valuefloat ;
```

---

### **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 simple attributes:

- `when`
- `value`

## Syntax

```
leakage_power ( ) {  
    when : "Boolean expression";  
    value: float;  
}
```

## 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 1-1](#) lists valid operators.

*Table 1-1 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. 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

This 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;  
}
```

---

## Using the leakage\_power Group for a Polynomial

You can represent leakage power information in a library by specifying a scalable polynomial power template group (`power_poly_template` group) or a scalable polynomial power group (`power` group) within a `leakage_power` group.

The Library Compiler syntax lets you use either polynomial equations or single float values to model leakage power. Polynomial-based leakage power modeling includes variables for supply voltage, temperature, and state dependency.

The Library Compiler syntax for modeling power is shown below.

### Syntax

```
library (poly_sample) {
    delay_model : polynomial ; /* polynomial template */
    power_supply () {
        ...
    } /* end power_supply */
    power_poly_template(poly_template_name_id) {
        variables (variable_1, variable_2, ..., variable_n) ;
        variable_1_range (minfloat, maxfloat) ;
        variable_2_range (minfloat, maxfloat) ;
        ...
        variable_n_range (minfloat, maxfloat) ;
        mapping (voltage1, power_rail_name_id) ;
        mapping (voltage2, power_rail_name_id) ;
        domain (domain_name_id) {
            variables (variable_1, variable_2, ..., variable_n) ;
            variable_1_range (minfloat, maxfloat) ;
            variable_2_range (minfloat, maxfloat) ;
            ...
            variable_n_range (minfloat, maxfloat) ;
            mapping (voltage1, power_rail_name_id) ;
            mapping (voltage2, power_rail_name_id) ;
        } /* end power_poly_template */
        ...
    }
}
```

```

}
cell(name) {
    leakage_power() {
        when : "Boolean expression" ;
        power(poly_template_nameid) {
            /* variable ranges are optional for overwriting */
            variable_1_range (minfloat, maxfloat) ;
            variable_2_range (minfloat, maxfloat) ;
            .....
            variable_n_range (minfloat, maxfloat) ;
            orders ("int , ..., int") ;
            coefs("float, ..., float") ;
            ...
        }
    }
}
cell(name) {
    /* piecewise polynomial */
    leakage_power() {
        when : "Boolean expression" ;
        power(poly_template_nameid) {
            domain (domain_nameid) {
                /* variable ranges are optional for overwriting */
                variable_1_range (minfloat, maxfloat) ;
                variable_2_range (minfloat, maxfloat) ;
                ...
                variable_n_range (minfloat, maxfloat) ;
                orders ("int , ..., int") ;
                coefs ("float, ..., float") ;
            } /* end power */
            ...
        } /* end leakage_power */
    } /* end cell */
} /* end library */

```

## Specifying leakage power in a power Group

The power group is defined within a leakage\_power group in a cell group at the library level, as shown here:

```

library (name) {
    cell (name) {

```

```

leakage_power () {
    power (template name) {
        ... power template description ...
    }
}
}

```

## Complex Attributes

`orders ("integer, ..., integer")`  
`coefs ("float, ..., float")`

### Group

`domain`

#### orders Attribute

This attribute specifies the orders of the variables for the polynomial.

#### coefs Attribute

This attribute specifies the coefficients used in the polynomial used to characterize power information.

### domain Group

```

leakage_power () {
    power (template name) {
        ... power template description ...
        domain() {
            ...
        }
    }
}

```

---

## **leakage\_power\_unit Simple Attribute**

This attribute indicates the units of the leakage-power values in the library. [Table 1-2](#) shows the possible unit values you can enter and their corresponding mathematical symbol equivalent.

*Table 1-2 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
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. If you do not define the `default_cell_leakage_power` attribute, it defaults to 0.0.

### **Example**

```
default_cell_leakage_power : 0.5;
```

Library Compiler issues a warning if the library has `cell_leakage_power` information but does not have the `default_cell_leakage_power` attribute defined.

Library Compiler looks for a definition for both the `default_cell_leakage_power` and `default_leakage_power_density` attributes. For details about the `default_leakage_power_density` attribute, see the next section.

In checking for the `default_cell_leakage_power` and `default_leakage_power_density` attributes, Library Compiler proceeds in the following manner:

- If you have not defined both attributes, Library Compiler issues a warning and sets both attributes to 0.0.
- If you define both attributes, Library Compiler takes no action.
- If you define the `default_cell_leakage_power` attribute but not the `default_leakage_power_density` attribute, Library Compiler issues a warning and creates a `default_leakage_power_density` attribute set to its default, 0.0.

- If you define the `default_leakage_power_density` attribute but not the `default_cell_leakage_power` attribute, Library Compiler issues a warning and creates a `default_cell_leakage_power` attribute set to its default, 0.0.

---

## **default\_leakage\_power\_density Simple Attribute**

The `default_leakage_power_density` attribute provides the mean static leakage power per area unit in the given technology. This attribute must be a nonnegative floating-point number. If you do not define this attribute, it defaults to 0.

### **Example**

```
default_leakage_power_density : 0.5;
```

Library Compiler looks for a definition for both the `default_leakage_power_density` and `default_cell_leakage_power` attributes. For details about the `default_cell_leakage_power` attribute, see “[default\\_cell\\_leakage\\_power Simple Attribute](#)” on page 1-17.

In checking for the `default_leakage_power_density` and `default_cell_leakage_power` attributes, Library Compiler proceeds in the following manner:

- If you have not defined both attributes, Library Compiler issues a warning and sets both attributes to 0.0.
- If you define both attributes, Library Compiler takes no action.

- If you define the `default_leakage_power_density` attribute but not the `default_cell_leakage_power` attribute, Library Compiler issues a warning and creates a `default_cell_leakage_power` attribute set to its default, 0.0.
- If you define the `default_cell_leakage_power` attribute but not the `default_leakage_power_density` attribute, Library Compiler issues a warning and creates a `default_leakage_power_density` attribute set to its default, 0.0.

## 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);
    /* power supply definatation */
    power_supply() {
        default_power_rail : VDD;
        power_rail (VDD1, 1.0);
        power_rail (VDD2, 2.0);
    }
    cell (NAND2) {
        rail_connection (P1,VDD1);
        rail_connection (P2,VDD2);
        cell_leakage_power : 1.0 ;
        leakage_power() {
            power_level : "VDD1";
            when : "!A1 !A2" ;
            value : 1.5 ;
        }
        leakage_power() {
            power_level : "VDD1";
            when : "!A1 A2" ;
            value : 2.0 ;
        }
    }
}
```

```

    }
leakage_power() {
    power_level : "VDD1";
    when : "A1 !A2" ;
    value : 3.0 ;
}
leakage_power() {
    power_level : "VDD1";
    when : "A1 A2" ;
    value : 4.0 ;
}
leakage_power() {
    power_level : "VDD2";
    when : "!A1 !A2" ;
    value : 3.5 ;
}
leakage_power() {
    power_level : "VDD2";
    when : "!A1 A2" ;
    value : 3.0 ;
}
leakage_power() {
    power_level : "VDD2";
    when : "A1 !A2" ;
    value : 4.0 ;
}
leakage_power() {
    power_level : "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 "
    power_level : "VDD1";
    rise_power( scalar ) {
        values("0.0");
    }
    fall_power( scalar ) {
        values("0.0");
    }
} /* end of internal power */
internal_power() {
    related_pin : " A1 "
    power_level : "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 "
            power_level : "VDD1";
            rise_power( scalar ) {
                values("0.0");
            }
            fall_power( scalar ) {
                values("0.0");
            }
        } /* end of internal power */
        internal_power() {
            related_pin : " A2 "
            power_level : "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 */

```

## Environmental Derating Factors Attributes

Use the following simple attributes to specify the environmental derating factors (voltage, temperature, and process) for the `cell_leakage_power` attribute.

- `k_volt_cell_leakage_power`
- `k_temp_cell_leakage_power`
- `K_process_cell_leakage_power`

The range for these scaling factors is –100.0 to 100.0. The default value is 0.

### Example

```
library(power_sample) {
    leakage_power_unit : 1nW;
    default_cell_leakage_power : 0.1;
    default_leakage_power_density : 0.5;
    k_volt_cell_leakage_power : 0.000000;
    k_temp_cell_leakage_power : 0.000000;
    k_process_cell_leakage_power : 0.000000;
    ...
    cell(AN2) {
        ...
        cell_leakage_power : 0.2;
        ...
        leakage_power () {
            when : "A" ;
            value : 2.0 ;
        }
    }
}
```

See the *Library Compiler Reference Manual: Technology and Symbol Libraries* for the `cell_leakage_power` attribute syntax. See [Chapter 12, “Building Environments,”](#) for information about scaling factors, operating conditions, and default values.

---

## 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

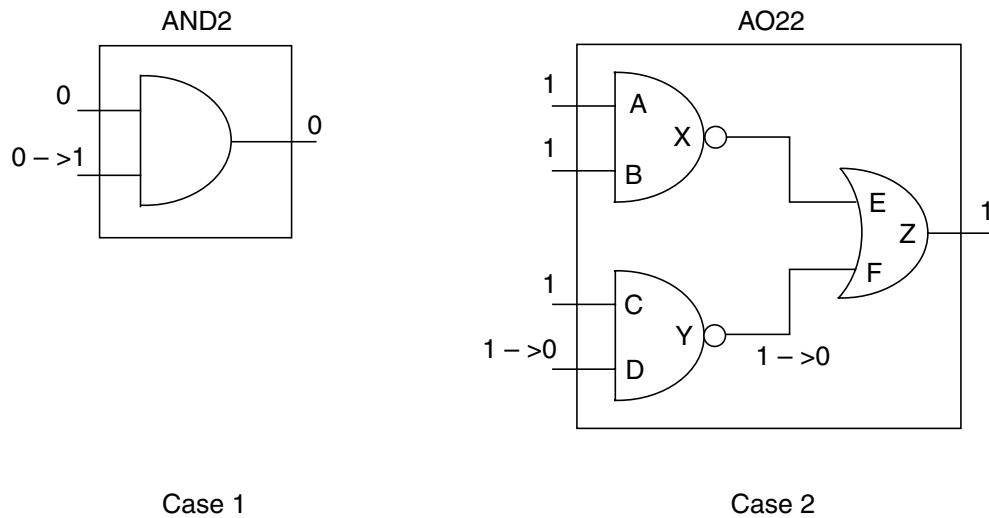
- To include the effect of the output capacitance in the `internal_power` attribute, which gives the output pins zero capacitance
- To give the output pins a real capacitance, which causes them to be included in the switching power, and model only short-circuit power as the cell's internal power

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.

This description is not entirely accurate, because some power is 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 1-2](#) shows two examples of an input transition that does not cause a corresponding output transition.

*Figure 1-2 Complex Gate Example*



Case 1

Case 2

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 will be 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  $T \times C$  times, once for each input transition time and output load capacitance combination. For example, if internal power will be modeled by use of a 3x3 table at the output of the cell, the design will have 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.

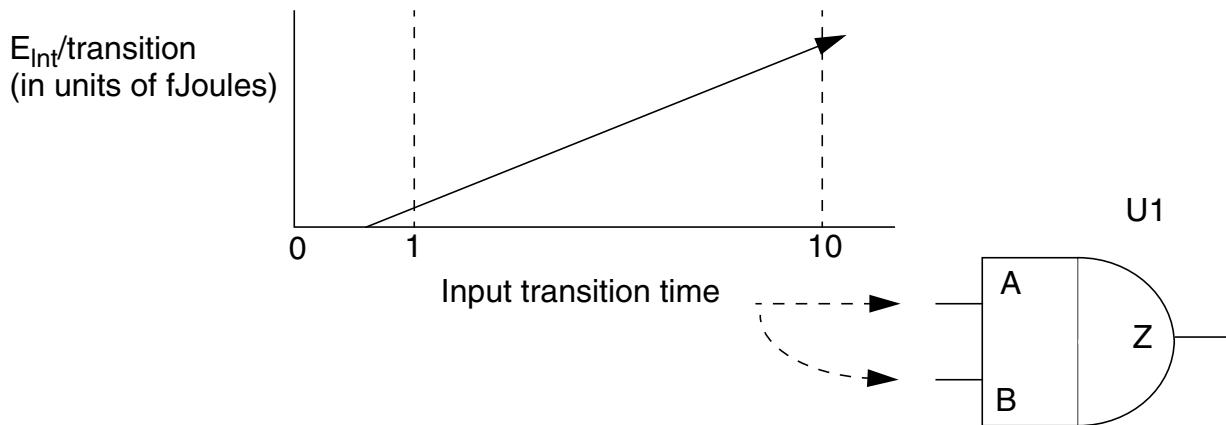
You can set `internal_power` attributes for input pins, which are indexed by input transition time. In this way, you can model gate AO22 in [Figure 1-2](#) or, more important, the power consumed by the flip-flop clock or reset pins.

Note:

The input pin power is added to the output pin power. When you model the library, avoid double counting.

[Figure 1-3](#) shows how to calculate the input pin power information for the one-dimensional lookup table describing internal cell U1.

Figure 1-3 Internal Power for Cell U1



To calculate the internal power for cell U1, use the following equation:

$$P_{\text{Int}} = (E_Z \times AF_Z) + (E_A \times AF_A) + (E_B \times AF_B)$$

$P_{\text{Int}}$

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
```

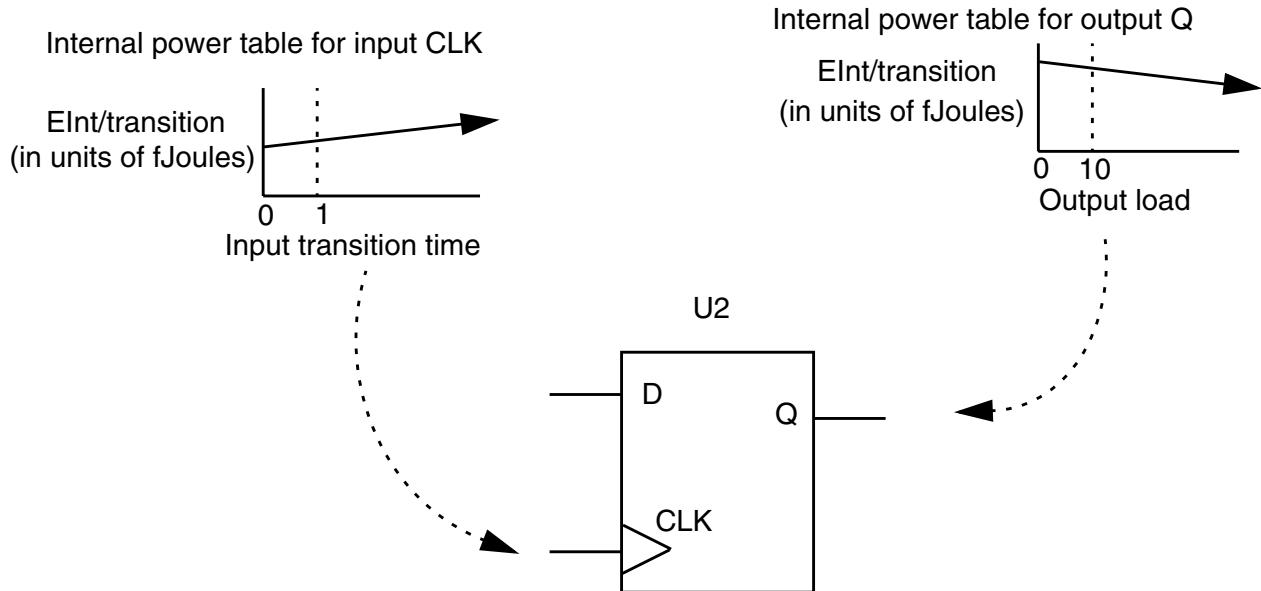
$$\text{Output_Pin_Energy} = (\text{Out_total})/2 - \text{Clk_Total}$$

Note:

Note that the output pin energy is adjusted by subtraction of the input pin power. This prevents double counting during simulation.

[Figure 1-4](#) shows how to calculate the input pin power information for internal cell U2.

*Figure 1-4 Internal Power for Cell U2*



To calculate the internal power for cell U2, use the following equation:

$$P_{Int} = (E_Q \times AF_Q) + (E_{CLK} \times AF_{CLK})$$

$P_{Int}$

Total internal power for the cell.

E

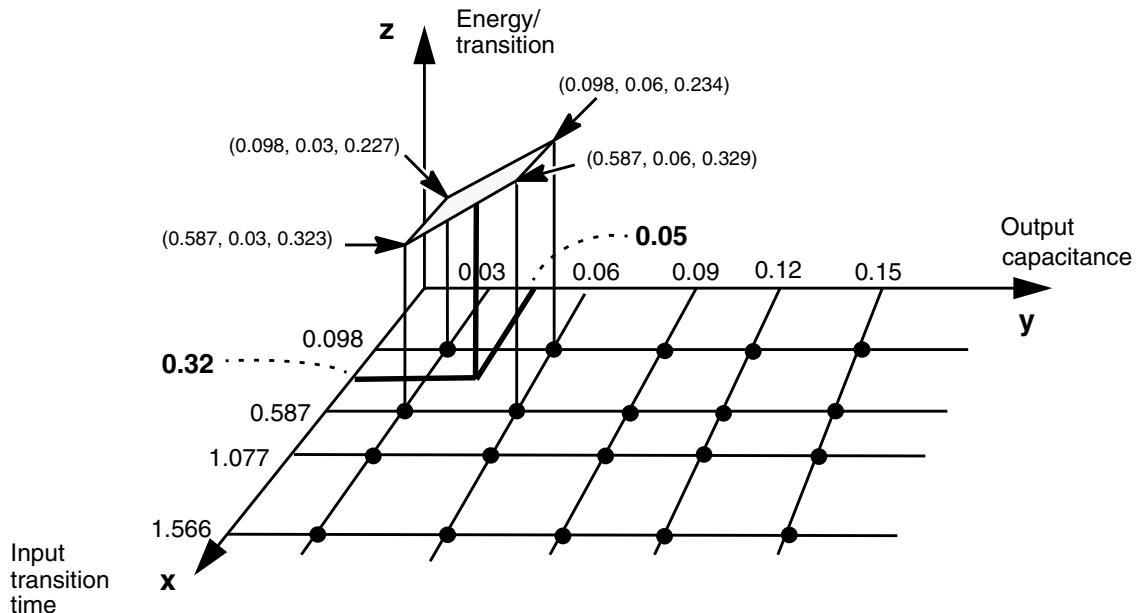
Internal energy for the pin.

AF

Activity factor.

**Figure 1-5** is an example of the two-dimensional lookup table for modeling output pin power in a cell.

*Figure 1-5 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:

$$\begin{aligned}
 \text{Net\_Power} &= \frac{V_{dd}^2}{2} \sum_{\forall \text{nets}(i)} (C_{\text{Load}_i} \times \text{TR}_i) \\
 &= \frac{5^2}{2} \times 100 \times \frac{2}{100} \times (0.1 \mu\text{W}) \\
 &= 25 \times (0.1 \mu\text{W}) \\
 &= 2.5 \mu\text{W}
 \end{aligned}$$

$\text{TR}$

Toggle rate (number of toggles per unit of time).

$C_{\text{Load}}$

Capacitive load of each net.

## Creating a Scalable Polynomial Power Model

Scalable polynomials provide a smaller and faster alternative to nonlinear lookup tables.

To create a scalable polynomial power model, you should be familiar with the following concepts and tasks:

- Polynomial representation
- Model description
- Scalable polynomial power calculation

The term *scalable* indicates that the form and the order of the polynomials are determined by, or scaled to, the given data. Given a predefined accuracy, most power data can be modeled with polynomials.

The advantages of using a predefined set of polynomials, instead of an arbitrary equation that attempts to fit all cases, are efficiency and controllability. Also, by allowing for the inclusion of temperature and voltage (besides slope and load) as additional dimensions for timing arcs, you can develop a single library instead of one library for each operating condition.

Note:

The scalable polynomial model does not alter the process of library construction. You still need to run a circuit-level simulator such as SPICE to collect timing data, and you need to use curve-fitting methods to convert the characterization data into computationally efficient polynomial equations with sufficient user-defined accuracy.

---

## Polynomial Representation

Polynomials play a key role in numerical computations. The fundamental theory is Taylor expansions, where an analytical function can be expressed as a finite series of polynomials.

The complete decomposed polynomial form represents the scalable polynomial power model syntax. The following example shows two variable functions, but it is easy to extend to the case of more variables. A two-variable polynomial function  $D(x,y)$  can be written as  $D(x,y)=P_m(x)Q_n(y)$ , where  $x$  and  $y$  are variables and  $P_m$  and  $Q_n$  are

the  $m$ th- and  $n$ th-order polynomials, respectively. There are  $(m+1)(n+1)$  coefficients for any given  $m$  or  $n$ , as the following equation illustrates.

$$\begin{aligned} P_1 Q_2 &= (a_0 + a_1 x_1) (b_0 + b_1 x_2 + b_2 x_2^2) = \\ A_{00} + A_{10}x_1 + A_{01}x_2 + A_{11}x_1x_2 + A_{02}x_2^2 + A_{12}x_1x_2^2 \end{aligned}$$

---

## Model Description

The scalable polynomial power model syntax allows you to specify up to variable  $n$  polynomials. The dimensions are temperature, voltage, input\_net\_transition, output\_net\_transition, total\_output\_net\_capacitance, equal\_or\_opposite\_output\_net\_transition, and voltage $i$ .

Most cells reference a single voltage rail, specified by voltage. In the case of level shifter cells, both voltage and voltage1 are used for voltage rails. The model considers analytical parameters (physical parameters) that affect the power calculation results to be variables.

For a large data set with abrupt changes, a single polynomial equation might not fit the entire operating region of interest. In this case, use the piecewise (adaptive domain) syntax to specify the breakpoints over the characterization data domains.

As with lookup tables, you describe a template specifying the polynomial form before specifying the values (in this case, polynomial orders and coefficients). Orders of coefficients can be included in the cell power, propagation delay, and transition delay.

You specify polynomials in the expanded decomposed form. Assume  $m$ ,  $n$ ,  $p$ ,  $q$ ,  $r$ ,  $s$ , and  $u$  as the orders of each of seven dimensions ( $x_1$ — $x_7$ ). The number of coefficients is

$$(m+1)(n+1)(p+1)(q+1)(r+1)(s+1)(u+1)$$

The coefficients are expressed as

$$\sum_{f=0}^{\infty} \sum_{e=0}^{\infty} \sum_{d=0}^{\infty} \sum_{c=0}^{\infty} \sum_{b=0}^{\infty} \sum_{a=0}^{\infty} A_{abcdef} x_1^a x_2^b x_3^c x_4^d x_5^e x_6^f$$

## Examples

**Example 1-1** shows the sequence of the coefficients in the case of a three-dimensional polynomial ( m=3, n=1, p=2 ).

### *Example 1-1 Sequence of Coefficients in the Case of a Three-Dimensional Polynomial*

$$A_{000} + A_{100}x + A_{200}x^2 + A_{300}x^3 + A_{010}y + A_{110}xy + A_{210}x^2y + \\ A_{310}x^3y + A_{001}z + A_{101}xz + A_{201}x^2z + A_{301}x^3z + A_{011}yz + \\ A_{111}xyz + A_{211}x^2yz + A_{311}x^3yz + A_{002}z^2 + A_{102}xz^2 + \\ A_{202}x^2z^2 + A_{302}x^3z^2 + A_{012}yz^2 + A_{112}xyz^2 + A_{212}x^2yz^2 + A_{312}x^3yz^2$$

**Example 1-2** shows how the sequence of the coefficients is represented in the power group:

### *Example 1-2 Sequence of Coefficients in the power Group*

"A<sub>000</sub>, A<sub>100</sub>, A<sub>200</sub>, A<sub>300</sub>, A<sub>010</sub>, A<sub>110</sub>, A<sub>210</sub>, A<sub>310</sub>, A<sub>001</sub>z, A<sub>101</sub>, A<sub>201</sub>, A<sub>301</sub>, A<sub>011</sub>, A<sub>111</sub>, A<sub>211</sub>, A<sub>311</sub>, A<sub>002</sub>, A<sub>102</sub>, A<sub>202</sub>, A<sub>302</sub>, A<sub>012</sub>, A<sub>112</sub>, A<sub>212</sub>, A<sub>312</sub>"

The scalable polynomial power model significantly improves memory usage, because

- Relatively few polynomial coefficients are stored, compared to table values

- Only one library is required, instead of one library per operating condition

The following features are also supported in the scalable polynomial power model:

- If it is not possible to curve-fit a complex table with sufficient accuracy and at a reasonable computation cost, you can still retain the lookup table in the library.
- You can specify piecewise polynomials and their range by using the piecewise polynomial template. To ensure continuity and completeness, specify piecewise polynomials to the entire specified variable domain.
- To ensure that the polynomial is used within the intended range, specify its range by using the `variable_n_range` attribute. This helps avoid explosions of the calculated slew or load due to the higher-order terms in the polynomial equation. The scalable polynomial power model linearly extrapolates outside the range of the equation, using the closest boundary values.

---

## Representing Internal Power Information

You can describe power dissipation in your libraries by using lookup tables or scalable polynomials.

---

### Specifying the Power Model

Use the library level `power_model` attribute to specify the power model for your library. The two valid values are `table_lookup` and `polynomial`. If you do not specify a power model, Library Compiler 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 1-45)
- The associated library-level attributes that specify scaling factors and a default value

### **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 1-45.

## 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 increasing order.

The number of floating-point numbers in the indexes determines the size of each dimension, as [Example 1-3](#) illustrates.

For each `power_lut_template` group, you must define at least one `variable_1` and one `index_1`.

[Example 1-3](#) shows four `power_lut_template` groups that have one-, two-, or three-dimensional templates.

### *Example 1-3 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.

Library Compiler has a predefined template named scalar; its value size is 1. You can refer to it by specifying 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.

---

## Using Scalable Polynomial Power Modeling

Instead of using lookup tables, you can represent power information directly in the library by specifying a scalable polynomial power model template and coefficients. Use the following groups and attributes to define your scalable polynomial delay model:

- The library-level `power_poly_template` group
- The `internal_power` group (see “[Defining Internal Power Groups](#)” on page 1-45)
- The associated library-level attributes that specify scaling factors and a default value

### **power\_poly\_template Group**

When you specify your delay model as polynomial, you can use the `power_poly_template` group to represent power information directly in the library, instead of using power lookup tables.

### **Syntax**

```
library (library_name_id) {  
    ...  
    power_poly_template(power_poly_template_name_id) {  
        variables(variable_i_enum, ..., variable_n_enum)  
        variable_i_range(min_value_float, max_value_float)  
        ...  
        variable_n_range(min_value_float, max_value_float)  
        mapping(value_enum, power_rail_name_id)  
        domain(domain_name_id) {  
            calc_mode : name_id ;  
            variables((variable_i_enum, ..., variable_n_enum)  
            variable_1_range(min_value_float, max_value_float)  
            ...  
            variable_n_range(min_value_float, max_value_float)  
            mapping(value_enum, power_rail_name_id)
```

```
        }
    }
}
```

## **power\_poly\_template Variables**

The `power_poly_template` group for specifying power accepts the following variables (`variable_i`, ..., `variable_n`), which you specify in the `variables` complex attribute. The variables you specify in the `power_poly_template` group represent the variables used in the equation to characterize cells in the library for internal power. The variables are

`temperature`

The temperature.

`voltage`

The primary power supply voltage.

`voltagei`

Used, in addition to `voltage`, when a cell requires many supply voltages, as in the case of level shifter cells.

`input_net_transition`

The input transition time of the pin specified in the `related_pin` attribute in the `internal_power` group.

`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.

*parameter*/

Used to reference user-defined process variables.

## Mapping Power Relationships

You use the `mapping` attribute in the `power_poly_template` group to specify the relationships between voltage variables in the polynomial and their corresponding power rails.

Depending on the case, specifying the `mapping` attribute can be optional or required, as shown in the following examples.

### Case 1

The `mapping` attribute is not required, because there is no voltage declaration.

```
variables(temperature, input_net_transition) ;
```

### Case 2

The `mapping` attribute is optional. When the attribute is omitted, Library Compiler assumes the value defined in the `voltage` attribute within the `operating_conditions` group.

```
variables(temperature, ..., voltage) ;
```

### Case 3

The `mapping` attribute, although optional, is declared to specify a value other than the default.

```
variables(temperature, ..., voltage) ;
...
mapping(voltage, VDD1) ;
```

## Case 4

The **mapping** attribute is required in order to specify a value defined in a **power\_rail** group for **voltage1**. Because there is no mapping statement for the **voltage** variable, Library Compiler assumes the value defined in the **voltage** attribute in the **operating\_conditions** group.

```
variables(temperature, ..., voltage, voltage1) ;  
...  
mapping(voltage1, VDD2) ;
```

## Case 5

The **mapping** attribute is required in order to specify a value defined in a **power\_rail** group for **voltage1**.

```
variables(temperature, ..., voltage1) ;  
...  
mapping(voltage1, VDD3) ;
```

## Case 6

The **mapping** attribute is required in order to specify a value defined in a **power\_rail** groups for **voltage1** and optionally to specify a value other than the default for **voltage**.

```
variables(temperature, ..., voltage, voltage1) ;  
...  
mapping(voltage, VDD4) ;  
mapping(voltage1, VDD5) ;
```

---

## 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 1-5 on page 1-74](#)).

---

## 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.

Library Compiler 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 we assume that 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's 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.

Library Compiler 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
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 we assume that 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

From pin	To pin	Label
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` group attribute.

Library Compiler 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 we assume that 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.

Library Compiler 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

From pin	To pin	Label
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 we assume that 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. You identify the resulting multiple power relationships by entering a name list, using the `internal_power` group attribute.

Library Compiler assumes that 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 in the name list 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 and groups:

Simple attributes:

- `equal_or_opposite_output`
- `falling_together_group`
- `power_level`
- `related_pin`
- `rising_together_group`
- `switching_interval`
- `switching_together_group`
- `when`

Groups:

- `fall_power`

- power
- rise\_power

## **equal\_or\_opposite\_output Simple Attribute**

This attribute designates an optional output pin or pins whose capacitance Library Compiler uses 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` variable 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 1-63 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 (namestring) {
    pin (namestring) {
        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.

## **power\_level Simple Attribute**

This attribute is used for multiple power supply modeling. In the `internal_power` group at the pin level, you can specify the power level used to characterize the tables.

For more information, see “[Modeling Libraries With Multiple Power Supplies](#)” on page 1-82 or “`internal_power Group`” in *Library Compiler Reference Manual: Technology and Symbol Libraries*.

## Syntax

```
power_level : "name" ;
```

*name*

Name of the power rail defined in the power\_supply group.

## Example

```
power_level : "VDD1" ;
```

For an output pin within a multiple power supply cell, regardless of the output\_signal\_level value, you must define internal\_power tables for all the rail\_connection of the cell.

## Example

```
cell ( cell7 ) {  
    area : 6.000 ;  
    rail_connection(PV1, VDD1);  
    rail_connection(PV2, VDD2);  
    pin ( Z ) {  
        direction : output ;  
        capacitance : 0.000 ;  
        function : "(A & B)";  
        output_signal_level : VDD1;  
        internal_power() {  
            related_pin : "A";  
            power_level : VDD1; /* must be there */  
            power (power_load) {  
                values (" 0.111111, 0.111111, 0.111111, 0.111111,\ 0.111111");  
            }  
        }  
        internal_power() {  
            related_pin : "A";  
            power_level : VDD2; /* must be there */  
            power (power_load) {  
                values (" 0.111111, 0.111111, 0.111111, 0.111111,\ 0.111111");  
            }  
        }  
    }  
    ...  
}
```

For an input pin within a multiple power supply cell, you must define an `internal_power` table to match the `input_signal_level` value. The rest of the rail connections are optional.

## Example

```
cell ( cell1 ) {
    rail_connection(PV1, VDD1);
    rail_connection(PV2, VDD2);
    rail_connection(PV3, VDD3);
    pin ( CP ) {
        direction : input ;
        input_signal_level : VDD1;

        internal_power() {
            power_level : VDD1 /* this table is required */
            power (power_ramp) {
                values ("1.934150, 2.148130, 2.449420, 3.345050,\ 4.305690");
            }
        }

        internal_power() {
            power_level : VDD2 ; /* this table is optional */
            power (power_ramp) {
                values ("1.934150, 2.148130, 2.449420, 3.345050,\ 4.305690");
            }
        }
        /* no table for VDD3 */
    }
}
```

If you want to use the same Boolean expression for multiple `when` statements in an `internal_power` group, you must specify a different power rail for each `internal_power` group, as shown in this example.

## Example

```
library (example) {
    ...
    power_supply() {
        default_power_rail : vdd;
        power_rail(VDD1, 1.95);
        power_rail(VDD2, 1.85);
        power_rail(VDD3, 1.75);
    } ;
```

```

...
cell ( cell1 ) {
    rail_connection(PV1, VDD1);
    rail_connection(PV2, VDD2);
    rail_connection(PV3, VDD3);
    pin(A) {
        ...
    }
    pin(B) {
        ...
    }
    pin ( CP ) {
        direction : input ;
        input_signal_level : VDD1;
        ...
        internal_power() {
            power_level : VDD1 ;
            when : "A !B";
            power (power_ramp) {
                values ("1.934150, 2.148130, 2.449420, 3.345050, 4.305690");
            }
        }
        internal_power() {
            power_level : VDD2 ;
            when : "A !B";
            power (power_ramp) {
                values ("1.934150, 2.148130, 2.449420, 3.345050, 4.305690");
            }
        }
    /* no table for VDD3 */
}
}

```

## **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. Library Compiler 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 “[switching\\_interval Simple Attribute](#),” which follows, 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 1-57; “[rising\\_together\\_group Simple Attribute](#)” on page 1-62; and “[switching\\_together\\_group Simple Attribute](#),” which follows.

## 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 1-63 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 1-1 on page 1-11](#) lists the Boolean operators valid in a `when` statement.

### **Example**

```
when : "A B" ;
```

Library Compiler checks that the `when` attribute and `related_pin` attribute do not contain the same pin, as they do in the following example:

**Example 1-4 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, Library Compiler generates an error message similar to the following:

```
Error: Line 183, The 'A' when condition includes the 'A'
related pin. (LIBG-215)
```

## **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.

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 */
orders ("integer, ..., integer")/* polynomial */
coefs ("float, ..., float")/* polynomial */
```

## Group

```
domain /* polynomial */
```

### **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");
```

### **orders Attribute**

This attribute specifies the orders of the variables for the polynomial.

### **coefs Attribute**

This attribute specifies the coefficients in the polynomial used to characterize power information.

## **domain Group**

```
domain (name_string) {
    pin (name_string) {
        internal_power () {
            fall_power (template name) {
                ... fall power description ...
            }
        }
    }
}
```

```

        domain() {
            ...
        }
    }
}
}

```

## Complex Attributes

variable\_n\_range  
orders  
coefs

You must ensure that the internal power units exactly match the derived units of switching power ( $CV^2$ ). This is necessary because Design Compiler adds the `internal_power` value and switching power to get the dynamic power consumption without any consideration of units.

Design Compiler 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 */
orders ("integer, ..., integer")/* polynomial */
coefs ("float, ..., float")/* polynomial */
```

## Group

```
domain /* polynomial */
```

### **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.

### **orders Attribute**

This attribute specifies the orders of the variables for the polynomial.

### **coefs Attribute**

This attribute specifies the coefficients in the polynomial used to characterize power information.

### **domain Group**

```
cell (name_string) {
    pin (name_string) {
        internal_power () {
            power (template name) {
                ... power template description ...
                domain() {
                    ...
                }
            }
        }
    }
}
```

### **Complex Attributes**

```
variable_n_range
orders
coefs
```

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.

## **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 */
orders ("integer, ..., integer")/* polynomial */
coefs ("float, ..., float")/* polynomial */
```

## **Group**

```
domain /* polynomial */
```

### **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.

## **orders Attribute**

This attribute specifies the orders of the variables for the polynomial.

## **coefs Attribute**

This attribute specifies the coefficients in the polynomial used to characterize power information.

## **domain Group**

```
domain (name_string) {  
    pin (name_string) {  
        internal_power () {  
            rise_power (template name) {  
                ... rise power description ...  
                domain() {  
                    ...  
                }  
            }  
        }  
    }  
}
```

## **Complex Attributes**

```
variable_n_range  
orders  
coefs
```

### **Note:**

Design Compiler 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 1-67.

## **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 1-5](#) shows cells that contain internal power information in the `pin` group.

### *Example 1-5 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 ;
        ...
    }
}
}

```

## Power-Scaling Factors

You use the following attributes to specify the environmental derating factors (voltage, temperature, and process) for the `internal_power` group. The range for these scaling factors is -100.0 to 100.0. The default value is 0.0.

### Syntax

```

k_volt_internal_power : float ;
k_temp_internal_power : float ;
k_process_internal_power : float ;

```

### Example

```

library(power_sample) {
    ...

```

```

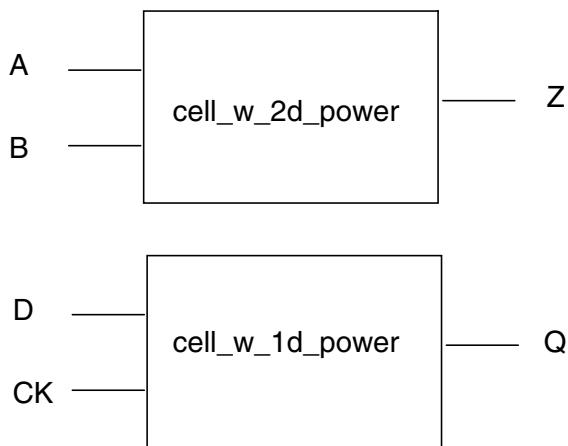
k_volt_internal_power : 0.000000;
k_temp_internal_power : 0.000000;
k_process_internal_power : 0.000000;
...
}

```

## Internal Power Examples

The examples in this section show how to describe the internal power of the 2-input sequential gate in [Figure 1-6](#), using one-, two-, and three-dimensional lookup tables.

*Figure 1-6 Library Cells With Internal Power Information*



## One-Dimensional Power Lookup Table

[Example 1-6](#) is the library description of the cell shown in [Figure 1-6](#), a cell with a one-dimensional internal power table defined in the pin groups.

*Example 1-6 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 1-6](#), 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:

$$\begin{array}{lcl} E_{CP} & = & 2.6 \\ E_Q & = & 5.0 \end{array}$$

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 Reference Manual*.

## Two-Dimensional Power Lookup Table

[Example 1-7](#) is the library description of the cell in [Figure 1-6](#), a cell with a two-dimensional internal power table defined in the `pin` groups.

### Example 1-7 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,\n
                        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  $E_Z$  and  $P_{int}$  are

$$\begin{aligned}E_Z &= 2.0 \times 0.0739 + 2.0263 \\&= 2.1741 \\&= 2.2\end{aligned}$$

$$\begin{aligned}P_{int} &= E_Z \times AF_Z \\&= 2.2 \times 1 \times 10^{-E2} \\&= 0.022\end{aligned}$$

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 1-8](#) is the library description for the cell in [Figure 1-6](#), a cell with a three-dimensional internal power table.

### *Example 1-8 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.

---

## Modeling Libraries With Multiple Power Supplies

This section describes the multiple supply voltage syntax that enables the PrimeTime *SI* tool to analyze designs with different power supply voltages for different cells and to perform a voltage-level mismatch check.

This syntax supports

- Multiple power supplies
  - Multiple input/output voltages
- 

## Power Supply Information

PrimeTime *SI* accurately determines changes in delay and slew resulting from differences in supply voltage, taking advantage of cell delay models specified in the technology library with NLDM (Nonlinear Delay Model) or SPDM (Scalable Polynomial Delay Model). PrimeTime *SI* also adjusts propagated transition times between drivers and loads that use different supply voltages.

In most technologies, factors such as operating temperature, supply voltage, and the manufacturing process can strongly affect circuit performance. However, the power supply for many new processes

has switched from 5 volts to 3.3 volts to lower the power consumption. Therefore, to support more than one power supply (for example, 3.3 volts and 5 volts), you need to make several changes to your technology library.

Library developers annotate the library cells with either the default single power supply with the existing syntax or with two or more power supplies, using the `power_supply` group.

The `power_supply` group, defined at the library level, specifies the various nominal voltage supplies needed in this library.

In addition to the operating temperature, the default voltage supply, and the manufacturing process, the `operating_conditions` group, defined at the library level, specifies voltage values in nominal operating conditions for all power rails defined in the `power_supply` group.

You can model cells with multiple power supplies in the technology library by defining the `rail_connection` attribute in the `cell` group and the `input_signal_level` and `output_signal_level` attributes in the `pin` group.

You can specify which voltage is used to characterize an `internal_power` group of a pin, by defining a `power_level` attribute in the `internal_power` group.

To report multiple power supply information for the technology library, use the `report_lib -power` command. For more information about using the `report_lib -power` command, see Chapter 6, “Advanced Composite Current Source Modeling.”

For more examples, see “[Example Libraries With Multiple Power Supplies](#)” on page 1-89.

---

## Defining Power Supply Groups

To specify a technology library with multiple power supplies, use the `power_supply` group at the library level, as shown here:

```
library(name) {  
    power_supply() {  
        ... power supply information ...  
    }  
}
```

Note:

If the `power_supply` group is not present in a library, it is assumed that the library has one default power supply.

You use the `power_rail` and `default_power_rail` attributes to define a `power_supply` group in a library.

### **power\_rail Complex Attribute**

You can define one or more `power_rail` attributes in a `power_supply` group. The `power_rail` attributes identify the power supplies that have the nominal operating conditions (defined in the `operating_conditions` group) and the nominal voltage values.

### **Syntax**

```
power_rail (power_supply_name_string, voltage_value_float);
```

*power\_supply\_name*

Specifies a power supply name that can be used later for reference. You can refer to it by assigning a string to the `input_signal_level` or `output_signal_level` attribute.

### *voltage\_value*

Identifies the voltage value associated with the string. The value is a floating-point number in the unit you define within the library group `voltage_unit` attribute.

#### **Example**

```
power_rail (VDD1, 5.0) ;
```

### **default\_power\_rail Simple Attribute**

Specifies a default voltage value for nominal operating conditions.

#### **Syntax**

```
default_power_rail : power_supply_name_string ;
```

#### **Example**

```
default_power_rail : VDD1 ;
```

---

## **Defining Cells With Multiple Power Supplies**

To define a cell with multiple power supplies, use the `rail_connection` attributes in the `cell` group. To define a cell with multiple power supplies at the pin level, use the `input_signal_level` and `output_signal_level` attributes.

### **rail\_connection Complex Attribute**

Specifies the presence of multiple power supplies in the cell. A cell with multiple power supplies contains two or more `rail_connection` attributes.

## Syntax

```
cell (name) {  
    ...  
    rail_connection (connection_name, power_supply_name) ;  
    ...  
}
```

*connection\_name*

Specifies a power connection for the pin name.

*power\_supply\_name*

Specifies the power supply group already defined in the `power_supply` group at the library level to be used as the value for the `power_level` attribute in the `internal_power` group (defined in the pin group).

[Example 1-9](#) shows a cell with `rail_connection` attributes.

### *Example 1-9 cell with rail\_connection Attributes*

```
cell (IBUF1) {  
    ...  
    rail_connection (PV1, VDD1) ;  
    rail_connection (PV2, VDD2) ;  
    ...  
}
```

For more information about syntax for multiple power supplies, see the *Library Compiler Reference Manual: Technology and Symbol Libraries*.

## **input\_signal\_level and output\_signal\_level Simple Attributes**

At the pin level, all input and I/O pins of a cell that has more than one power supply require an `input_signal_level` attribute and an `output_signal_level` attribute to define which voltage level to apply to the pins.

You can define various voltage levels in the `pin` group of a cell with multiple power supplies. You can also define various voltage levels in the `pin` group of a cell with the `input_signal_level` or `output_signal_level` attribute. If the `input_signal_level` or `output_signal_level` is missing, you can apply the default power supply name to the cell.

These attributes assign the `power_supply_name` already defined with the `power_rail` attribute in the `power_supply` group.

If the `pin` group does not contain an `input_signal_level` attribute or an `output_signal_level` attribute, Library Compiler applies the power supply name already defined by the `default_power_rail` attribute in the `power_supply` group at the library level.

## Syntax

```
input_signal_level: power_supply_name_string ;  
output_signal_level: power_supply_name_string ;  
power_supply_name
```

Name of the power supply already defined at the library level.

## Example

```
input_signal_level: VDD1 ;  
output_signal_level: VDD2 ;
```

Example 1-10 shows a library with a `power_supply` group, nominal operating conditions, and an `operating_conditions` group.

### Example 1-10 Library With a `power_supply` Group With Nominal Operating Conditions and an `operating_conditions` Group

```
library (multiple_power_supply) {  
    power_supply ( ) { /* Define before operating
```

```

        conditions and cells. */
default_power_rail : VDD0 ;
power_rail (VDD1, 5.0) ;
power_rail (VDD2, 3.3) ;
...
}
nom_process : 1.0 ;
nom_temperature : 25.0 ;
nom_voltage : 5.0 ;
...
operating_conditions (MPSS) {
    process : 1.5 ;
    temperature : 70 ;
    voltage : 4.75 ;
    tree_type : "worse_case_tree" ;
    power_rail (VDD1, 4.8) ;
    power_rail (VDD2, 2.9) ;
}
}

```

## Multiple-Power-Supply Library Requirements

Library Compiler confirms that a library with multiple power supplies meets the following requirements:

- The `power_supply` group is defined before the `operating_conditions` group and before the `cell` groups.
- The `power_supply` names associated with the `default_power_rail` and `power_rail` attributes are unique within a library.
- A `power_rail` attribute in an `operating_conditions` group has a power supply string name predefined in the `power_supply` group.
- All the power supply names defined by the `power_rail` attribute in the `power_supply` group exist in the `operating_conditions` group.

- An `input_signal_level` attribute and an `output_signal_level` attribute in a `pin` group have predefined power supply string names in the `power_supply` group.
- The `rail_connection` names within a cell are unique.
- A `rail_connection` attribute in a `cell` group has a predefined power supply string name in the `power_supply` group.
- In any cell with two or more power supplies, all the pins, buses, and bundles have an `input_signal_level` for input type and an `output_signal_level` attribute for output type.
- In any cell with two or more power supplies, all the I/O pins, buses, and bundles have an `input_signal_level` and an `output_signal_level` attribute.
- An output pin within a cell that has multiple power supplies requires `internal_power` tables with the `power_level` attribute for all the `rail_connection` attributes of the cell.
- An input pin within a cell that has multiple power supplies requires an `internal_power` table with the `power_level` value matching the `input_signal_level` value.

## Example Libraries With Multiple Power Supplies

[Example 1-11](#) shows a library with multiple power supplies defined at the library, cell, and pin levels.

*Example 1-11 Library With a `power_supply` Group and Power Supplies Defined at the Library, Cell, and Pin Levels*

```
library(pow) {
  ...
}
```

```

voltage_unit : "1V";
power_supply() {
    default_power_rail : VDD0;
    power_rail(VDD1, 5.0);
    power_rail(VDD2, 3.3);
}
operating_conditions(WCCOM) {
    process : 1.5 ;
    temperature : 70 ;
    voltage : 4.75 ;
    tree_type : "worst_case_tree" ;
    power_rail(VDD1, 4.8);
    power_rail(VDD2, 2.9);
}
cell(IBUF1) {
    ...
    rail_connection(PV1, VDD1);
    rail_connection(PV2, VDD2);
    pin(A) {
        direction : input;
        ...
        input_signal_level : VDD1;
    }
    pin(EN) {
        direction : input;
        ...
        input_signal_level : VDD1;
    }
    pin(Z) {
        direction : output;
        output_signal_level : VDD2;
        ...
    }
    pin(Z1) {
        direction : inout;
        ...
        input_signal_level : VDD1;
        output_signal_level : VDD2;
    }
}
}

```

*Example 1-12 Library Defining All Power Supplies With Nominal Operating Conditions*

```
power_supply()  {
    default_power_rail : VDD0;
    power_rail : (VDD1, 5.0);
    power_rail : (VDD2, 3.3) ;
}
operating_conditions (MPSCOM) {
    process : 1.5;
    temperature : 70;
    voltage : 4.75;
    tree_type : worst_case_tree;
    power_rail : (VDD1, 5.8);
    power_rail : (VDD2, 3.3);
}
```

*Example 1-13 Cell With Default Power Supply*

```
cell(with_no_power_supply) {
    area : 10;
    pin (A) {
        direction: input;
        capacitance: 1.0;
    }
    pin (Z) {
        direction: output;
        function : "!A";
    }
} /* end cell */
```

*Example 1-14 Cell With One Power Supply*

```
cell(with_one_power_supply) {
    area : 10;
    rail_connection (PV1, VDD1) ;
    pin (A) {
        direction: input;
        capacitance: 1.0;
    }
    pin (Z) {
        direction: output;
        function : "!A";
    }
}
```

```
    } /* end cell */
```

*Example 1-15 Cell With Two Power Supplies*

```
cell(with_2_power_supplies) {
    area : 0;
    rail_connection (PV1, VDD1) ;
    rail_connection (PV2, VDD2) ;
    pin (A) {
        direction: input;
        capacitance: 1.0;
        input_signal_level: VDD1;
    }
    pin (EB) {
        direction: input;
        capacitance: 1.0;
        input_signal_level: VDD1;
    }
    pin (Z) {
        direction: output;
        function : "A";
        three_state: "EB";
        output_signal_level: VDD2;
    }
} /* end cell */
```

*Example 1-16 Cell With Two Power Supplies and a Bidirectional Pin*

```
cell(bidir_with_2_power_supplies) {
    area : 0
    rail_connection (PV1, VDD1) ;
    rail_connection (PV2, VDD2) ;
    pin(PAD) {
        direction : inout;
        function : "A";
        three_state : "EN";
        input_signal_level: VDD1;
        output_signal_level: VDD2;
    }
    pin(A) {
        direction : input;
        capacitance : 3;
        input_signal_level: VDD1;
    }
}
```

```

pin(EN) {
    direction : input;
    capacitance : 3;
    input_signal_level: VDD1;
}
pin(X) {
    direction : output;
    function : "PAD";
    output_signal_level: VDD2;
}
} /* end cell */

```

---

## **Input and Output Voltage Information**

To enable PrimeTime *SI* to perform signal-level checking, Library Compiler allows the input and output voltage levels defined in the technology library for the pins of cells.

### **Syntax**

```

library (namestring) {
    input_voltage (namestring) {
        vil : float | expression ;
        vih : float | expression ;
        vimin : float | expression ;
        vimax : float | expression ;
    }
}

```

**vil**

The maximum input voltage for which the input to the core is guaranteed to be a logic 0.

**vih**

The minimum input voltage for which the input to the core is guaranteed to be a logic 1.

**vimin**

The minimum acceptable input voltage.

`vimax`

The maximum acceptable input voltage.

You can use the `input_voltage` and `output_voltage` groups to define a set of input or output voltage ranges for your cells. You can then assign this set of voltage ranges to the input or output pin of a cell.

For example, you can define an `input_voltage` group with a set of high and low thresholds and minimum and maximum voltage levels. Then you can use the `input_voltage` attribute in the `pin` group to assign those ranges to the cell pin.

```
input_voltage : my_input_voltage ;
```

## **input\_voltage Group**

This group captures a set of voltage levels at which an input cell is driven.

### **Example**

```
library (my_library) {
...
    input_voltage(CMOS) {
        vil : 0.3 * VDD;
        vih : 0.7 * VDD;
        vimin : -0.5;
        vimax : VDD + 0.5;
    }
}
```

The default values 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 cell is driven.

The default values represent nominal operating conditions. These values fluctuate with the voltage range defined in the operating conditions groups.

All voltage values are in the unit 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;  
    }  
}
```

PrimeTime *SI* reports either of the following conditions as an “incompatible voltage” error:

- Driver VOmax > Load VImax
- Driver Vomin < Load Vilmin

PrimeTime *SI* reports either of the following conditions with a “mismatching driver-load voltage” warning:

- Driver VOH < Load VIH
- Driver VOL > Load VIL

---

## 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

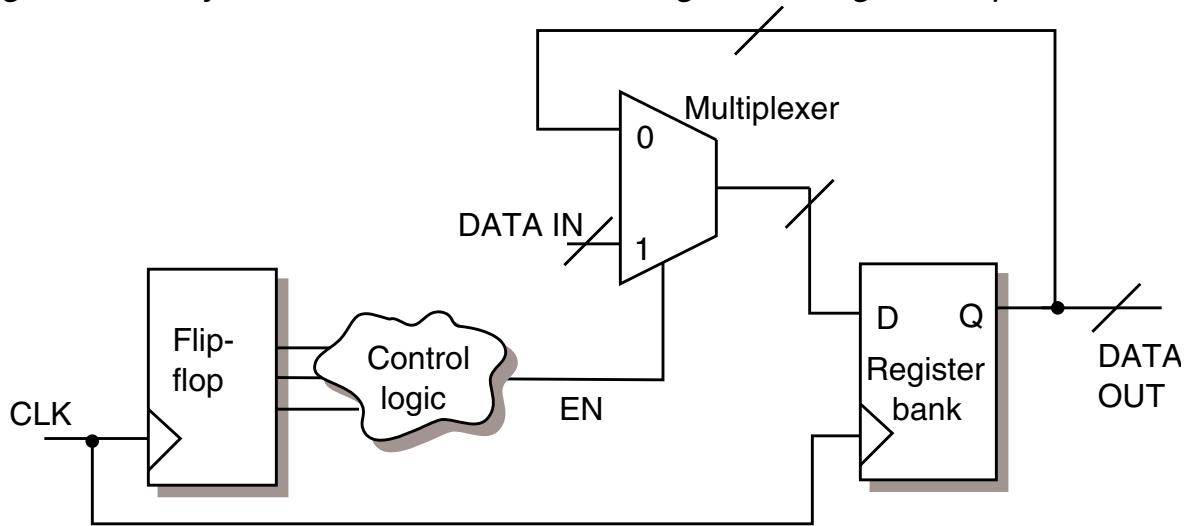
Clock gating provides a power-efficient implementation of register banks that are disabled during some clock cycles.

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 1-7](#) shows a simple implementation of a register bank using a multiplexer and a feedback loop.

*Figure 1-7 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 1-101.

---

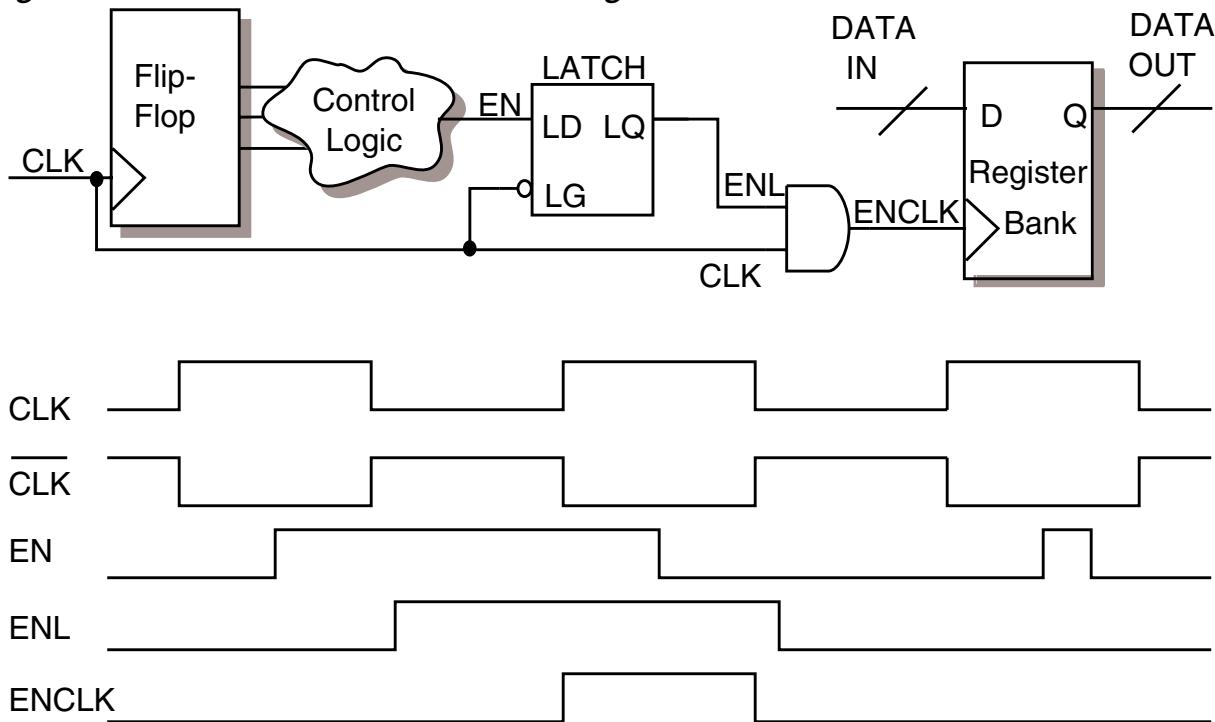
## Looking at a Gated Clock

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 1-7](#), 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 1-8 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 1-8, the waveforms of the signals are shown with respect to the clock signal, CLK.

*Figure 1-8 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 1-7](#). 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.

Library Compiler 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 1-3](#). Setting some of the pin attributes, such as those for test and observability, is optional.

*Table 1-3 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 more details about the `clock_gating_integrated_cell` attribute and the corresponding pin attributes, see the *Library Compiler User Guide: Methodology and Modeling Functionality in Technology Libraries*, the *Library Compiler Reference Manual: Technology and Symbol Libraries*, and the *Power Compiler Reference Manual*.

### **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 clock-gating cells, see the *Library Compiler User Guide: Methodology and Modeling Functionality in Technology Libraries*.

For more information about identifying pins on integrated clock-gating cells, see “[Using an Integrated Clock-Gating Cell](#)” on [page 1-100](#).

For additional information about integrated clock gating, see the *Power Compiler Reference Manual*.

### **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. Design Compiler uses the `clock_gate_out_pin` attribute when it compiles a design containing gated clocks that were introduced by Power Compiler.

## 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

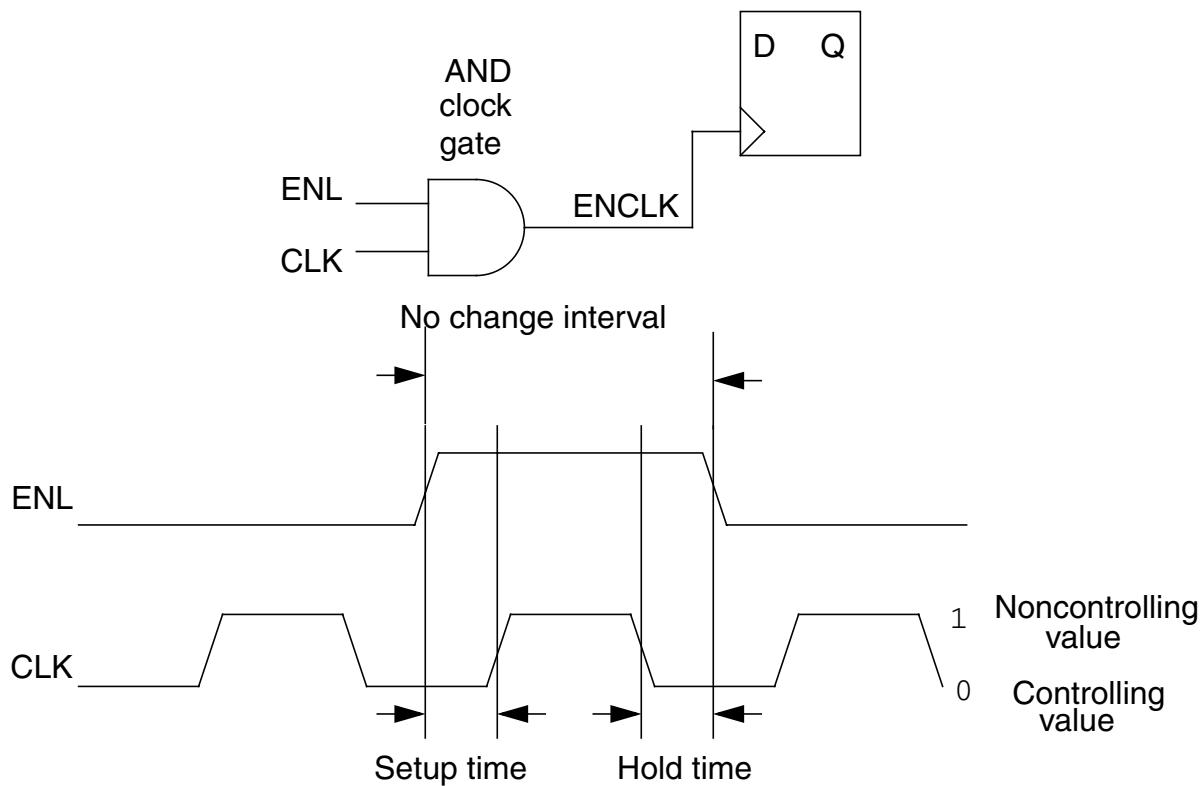
```
clock_gate_out_pin : true;
```

### **Clock-Gating Timing Considerations**

The clock gate must not alter the waveform of the clock—other than turning the clock signal on and off.

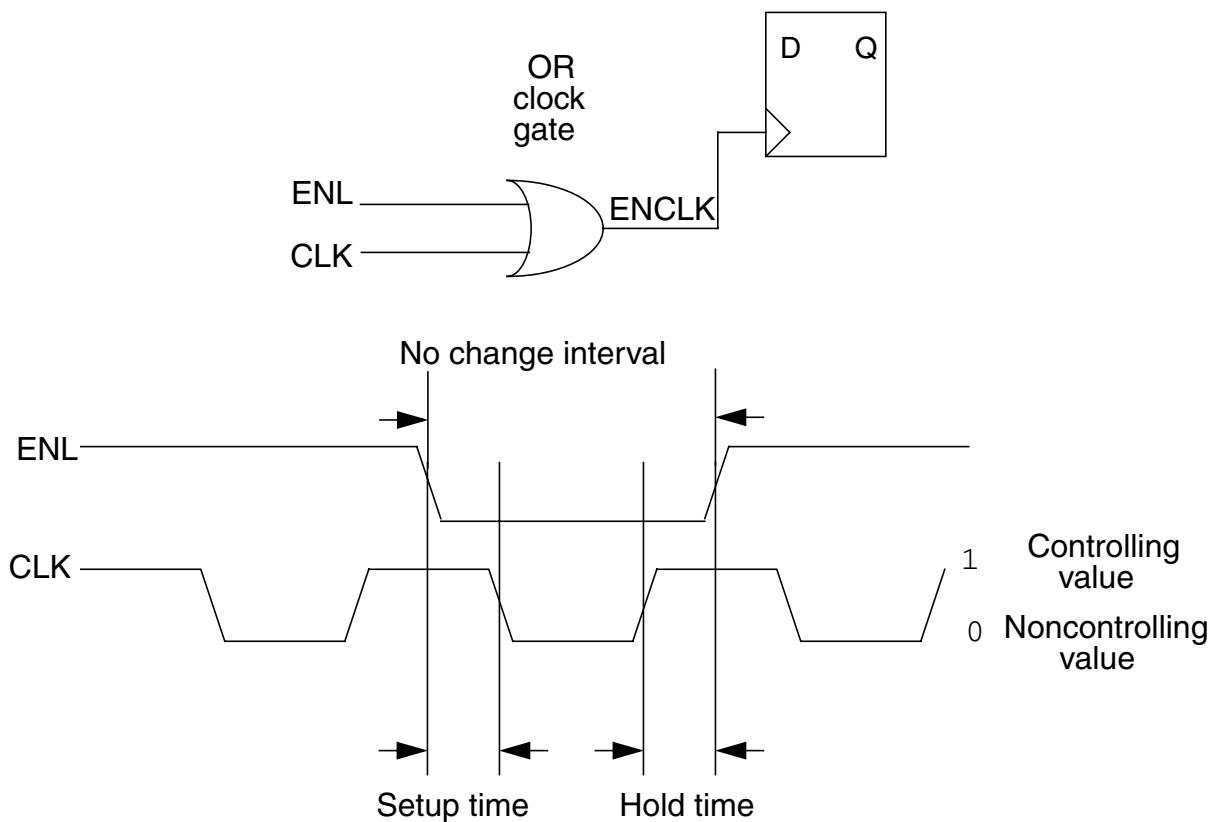
[Figure 1-9](#) and [Figure 1-10](#) show the relationship of setup and hold times to a clock waveform. [Figure 1-9](#) shows the relationship when an AND gate is the clock-gating element. [Figure 1-10](#) shows the relationship when an OR gate is the clock-gating element.

*Figure 1-9 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 1-10 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 1-4](#) 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 1-4 Values of the `clock_gating_integrated_cell` Attribute*

<b>Value of <code>clock_gating_integrated_cell</code> attributes</b>	<b>Setup arc</b>	<b>Hold arc</b>
latch_posedge	rising	rising
latch_negedge	falling	falling
none_posedge	falling	rising
none_negedge	rising	falling
ff_posedge	falling	falling
ff_negedge	rising	rising

In checking the timing arcs on integrated clock-gating cells, Library Compiler does the following:

- If a combinational integrated clock-gating cell or simple clock gating cell has sequential setup and hold arcs, Library Compiler issues no warning.
- If a sequential integrated clock-gating cell has combinational arcs from the inputs to the outputs, Library Compiler issues no warning.
- If there is no setup or hold on the enable pin from the clock pin, Library Compiler issues an error.
- If there is no combinational arc from the clock to the output, Library Compiler 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, Library Compiler issues an error message. This arc prevents propagation of clock properties to the output.

Note:

Library Compiler supports only the gated cells defined in Appendix A of the *Library Compiler User Guide: Methodology and Modeling Functionality in Technology Libraries*.

### Integrated Clock-Gating Cell Example

[Example 1-17](#) 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.

For a listing of all the value options you can enter for the `clock_gating_integrated_cell` attribute and the circuitry schematics and examples for each, see Appendix A in the *Library Compiler User Guide: Methodology and Modeling Functionality in Technology Libraries*.

### *Example 1-17 Integrated Clock-Gating Cell*

```
cell(CGLPCO) {
    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;
            intrinsic_rise : 0.4;
            intrinsic_fall : 0.4;
            related_pin : "CLK";
        }
        timing() {
            timing_type : hold_rising;
            intrinsic_rise : 0.4;
            intrinsic_fall : 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;
            intrinsic_rise : 0.48;
            intrinsic_fall : 0.77;
            rise_resistance : 0.1443;
            fall_resistance : 0.0523;
            slope_rise : 0.0;
            slope_fall : 0.0;
            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

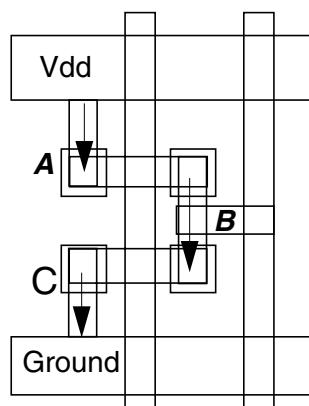
When high-density current passes through a thin metal wire, the high-energy electrons exert forces upon the atoms that cause the electromigration of the atoms. Electromigration can drastically reduce the lifetime of the silicon, by causing increased resistivity of the metal wire or by creating a short circuit between adjacent lines.

---

### Overview

[Figure 1-11](#) shows an inverter. The connections between both drains of the NMOS and PMOS transistor (section B), as well as between the sources and Vdd and ground (sections A and C), are typical examples of lines exposed to unidirectional current with a pulse rate equal to the output toggle rate.

*Figure 1-11 Electromigration in an Inverter*



The 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

$$MTTF \equiv AJ^{-n} \exp(E_a/kT)$$

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<sub>a</sub>

Activation energy.

k

Boltzmann's constant.

T

Temperature in Kelvin.

In order to meet electromigration requirements for a cell, each output rate can be constrained with maximal output toggle rate.

Good library characterization is an essential first step toward solving the electromigration problem. For the purposes of verification, each cell output should be 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 will be 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 the `variable_1` and `variable_2` string attributes and the `index_1` and `index_2` complex attributes.

- Use the `variable_1` string attribute to specify the first dimensional variable and the `variable_2` string attribute to specify the second dimensional variable used by the library

developer to characterize cells within the library for electromigration. You can assign `input_transition_time` or `total_output_net_capacitance` to `variable_1` and `variable_2`.

- Use the `index_1` complex attribute to specify the breakpoints along the `variable_1` table axis, and use the `index_2` complex attribute to specify the breakpoints along the `variable_2` table axis.

The `electromigration` pin-level group attribute 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.
- Assign the same name for the `em_max_toggle_rate` pin-level group you specified for the `em_lut_template` library-level group whose template you want `em_max_toggle_rate` to use.

The `em_max_toggle_rate` pin-level 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 `nindex_1 x nindex_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_string) {
  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.

```
library (name) {
    cell (name) {
        pin (name) {
            electromigration () {
                em_max_toggle_rate(em_template_name) {
                    ... em_max_toggle_rate description ...
                }
            }
        }
    }
}
```

## 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 ;
```

---

## Scalable Polynomial Electromigration Model

At the library level, use the `poly_template` group to specify cell electromigration polynomial equation variables, variable ranges, voltage mapping, and piecewise data. You can specify the following parameters in the `poly_template` group variables: `input_transition_time`, `total_output_net_capacitance`, `temperature`, and `voltages`.

## Syntax

```
library(em_sample) {
    delay_model : polynomial;
    ...
    poly_template(template_namestring) {
        variables(variable_1, variable_2, ..., variable_n);
        variable_1_range : (float, float);
        variable_2_range : (float, float);
        ...
        variable_n_range : (float, float);
        mapping(voltage, power_rail_name);
        mapping(voltage1, power_rail_name);
        domain(domain_namestring) {
            calc_mode: calc_mode_1_namestring;
            variable_1_range : (float, float)
            ...
            variable_n_range : (float, float);
            mapping(voltage, power_rail_name);
            mapping(voltage1, power_rail_name);
        }
    }
}
```

```
        }
    }
}
```

Similarly, you can define an electromigration group within the pin group to provide specific parameters of the polynomial representation. The syntax is as follows:

## Syntax

```
pin(namestring) {
    ...
    /* em_temp_degradation_factor: float; */

    electromigration() {
        when : "boolean expression";
        em_max_toggle_rate(template_namestring) {
            /* variable ranges are optional for overwriting */
            variable_1_range : (float, float);
            /* optional for overwriting*/
            variable_2_range : (float, float);
            /* optional for overwriting*/
            variable_n_range : (float, float);
            /* optional for overwriting */
            orders("float,..., float");
            coefs("float,..., float");
            ...
            domain(domain_namestring) {
                variable_i_range : (float, float); /* optional */
                ...
                variable_n_range : (float, float); /* optional */
                orders("float,..., float");
                coefs("float,..., float");
            }
        }
    }
}
```

# 2

## Advanced Low Power Modeling

---

Advanced low power design methodologies such as multivoltage and MTCMOS require new library cells. These new cells need additional modeling attributes to drive implementation tools during library cell selection. Library Compiler is continually enhancing Liberty syntax with attributes and statements to support tool features for low power designs.

This chapter describes the power and ground (PG) pin syntax and gives examples for level-shifter, enable level-shifter, isolation cells (special cells used to connect the netlist in the different voltage domains in order to meet design constraints), and switch cells (a type of cell that drives power to other logic cells). Going forward, all low power syntax modeling enhancements will be based on the new power and ground pin syntax.

This chapter includes the following sections:

- Power and Ground (PG) Pins
- Level Shifter and Isolation Cells in a Multivoltage Design
- Switch Cell Modeling

---

## Power and Ground (PG) Pins

Library Compiler provides support for power and ground library pins. In Liberty, a power pin is defined as a current source pin, and a ground pin is defined as a current sink pin. This section provides an overview of the Library Compiler support for power and ground pins.

### **Important:**

Library Compiler version Y-2006.06 or later is required to compile a library based on power and ground pin syntax. All future Liberty syntax updates will be based on the power and ground pin Liberty syntax version Y-2006.06 and later.

The syntax in Library Compiler versions earlier than Y-2006.06 has limitations in describing the functionalities and characteristics of the new standard cells, such as level-shifter cells, isolation cells, and power switches.

The following section provides the general syntax for power and ground pin libraries.

---

## Syntax

The power and ground pin syntax for general 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_namei0>) {
        voltage_name : <voltage_namei0>;
        pg_type : <type _valuei0>
    }
    ...
    leakage_power() {
        related_pg_pin : <pg_pin_namej0>;
        ...
    }
    ...
    pin (<pin_namem0>) {
        direction : input/inout;
        power_down_function : !VDD + VSS;
        related_power_pin : <pg_pin_namem0p>;
        related_ground_pin : <pg_pin_namem0g>;
        input_signal_level_low : <voltage_valuem0l>;
        input_signal_level_high : <voltage_valuem0l>;
        ...
    }
    ...
    pin (<pin_namen0>) {
        direction : inout/output;
        related_power_pin : <pg_pin_namen0p>;
        related_ground_pin : <pg_pin_namen0g>;
        output_signal_level_low : <voltage_valuem0l>;
        output_signal_level_high : <voltage_valuem0l>;
        internal_power() {
            related_pg_pin : <pg_pinn0i>;
            ...
        }/* end internal_power group */
        ...
    }/* end pin group*/
    ...
}/* end cell group*/
...
}/* end library group*/

```

---

## **Library-Level Attributes**

This section describes library-level attributes.

### **voltage\_map Attribute**

The library-level `voltage_map` attribute associates the voltage name with the relative voltage values. These defined voltage names are referenced by the `pg_pin` groups defined at the cell level.

### **default\_operating\_conditions Attribute**

The `default_operating_conditions` attribute is required to explicitly specify 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. At least one voltage map in the library should have a value of 0, which will become the reference value to which other voltage map values relate.

---

## **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.

## **voltage\_name Attribute**

The `voltage_name` string attribute is mandatory in all `pg_pin` groups. The `voltage_name` attribute specifies the associated voltage name of the power and ground pin.

## **pg\_type Attribute**

The `pg_type` attribute specifies the type of power and ground pin. The attribute is optional in `pg_pin` groups. The `pg_type` attribute includes five possible values: `primary_power`, `primary_ground`, `backup_power`, `backup_ground`, and `internal_power`.

**Table 2-1** describes the `pg_type` values.

*Table 2-1 pg\_type Values*

<b>Value</b>	<b>Description</b>
<code>primary_power</code>	Specifies that <code>pg_pin</code> is a primary power source (the default). If the <code>pg_type</code> attribute is not specified, <code>primary_power</code> is the <code>pg_type</code> value.
<code>primary_ground</code>	Specifies that <code>pg_pin</code> is a primary ground source.
<code>backup_power</code>	Specifies that <code>pg_pin</code> is a backup (secondary) power source (for retention registers, always-on logic, and so on).
<code>backup_ground</code>	Specifies that <code>pg_pin</code> is a backup (secondary) ground source (for retention registers, always-on logic, and so on).
<code>internal_power</code>	Specifies that <code>pg_pin</code> is an internal power and ground source (for MTCMOS cells).

---

## **Pin-Level Attributes**

This section describes pin-level attributes.

### **power\_down\_function Attribute**

The `power_down_function` string attribute is used to identify the condition when an output pin is switched off by `pg_pin` and to specify the Boolean condition under which the cell's 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.

### **related\_power\_pin and related\_ground\_pin Attributes**

The `related_power_pin` and `related_ground_pin` attributes are defined at the pin level for output, input, and inout pins. The attributes are used to associate a predefined power and ground pin with the corresponding signal pins under which they are defined.

If you do not specify the `related_power_pin` and `related_ground_pin` attribute values, the following defaults are used:

- The first `pg_pin` that has the `pg_type` attribute set to `primary_power` becomes the default value for `related_power_pin`.
- The first `pg_pin` that has the `pg_type` attribute set to `primary_ground` becomes the default value for `related_ground_pin`.

In a library based on power and ground pin syntax, the `pg_pin` groups are mandatory for each cell, and a cell must have at least one `primary_power` power and ground pin and at least one

`primary_ground` power and ground pin. Therefore, a default `related_power_pin` and `related_ground_pin` will always exist in a cell.

### **`output_signal_level` and `input_signal_level` Simple Attributes**

The `input_signal_level` and `output_signal_level` attributes are defined at the pin level and are used to specify which signal is driving the input pin or the output pin, respectively. The attributes define special overdrive cells that do not have a physical relationship with the power and ground on input pins or any cell that does not have a power pin relationship with the output pin.

If the `input_signal_level` and the `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.

### **`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).

[Table 2-2](#) lists the power and ground pin attributes and groups in the old syntax and maps them to the attributes and groups in the new syntax, introduced in the Y-2006.06 release.

*Table 2-2 Power and Ground Pin Syntax Changes*

Old syntax	New syntax
nom_voltage simple attribute	no change
power_supply group	voltage_map complex attribute
rail_connection complex attribute	pg_pin group
power_level simple attribute	related_pg_pin simple attribute
input_voltage / output_voltage groups	no change
input_voltage / output_voltage simple attributes	no change
input_signal_level simple attribute	related_power_pin and related_ground_pin attributes (plus input_signal_level to model overdrive cells)

**Table 2-2 Power and Ground Pin Syntax Changes**

Old syntax	New syntax
output_signal_level simple attribute	related_power_pin and related_ground_pin simple attributes
input_signal_level_low / input_signal_level_high simple attributes	no change
output_signal_level_low / output_signal_level_high simple attributes	no change
operating_condition with power_rail attributes	The power_rail attribute no longer needs to be defined in the operating_conditions group

---

## Library Checking Rules

Library Compiler issues an error message if the new power and ground pin syntax and old syntax are both specified in a library. For example, the rail\_connection and power\_level attributes, which use the old syntax, and the pg\_pin and related\_pg\_pin attributes, which use the new power and ground pin syntax, are not compatible. Any new power and ground pin enhancements will be based on the new power and ground pin syntax, introduced in the Y-2006.06 release.

---

## **Level Shifter and Isolation Cells in a Multivoltage Design**

Level-shifter cells and isolation cells are cells used to connect the netlist in the different voltage domains in order to meet the design constraint. Level-shifter insertion is one of the most important aspects of multivoltage optimization flow.

To speed automation of a Synopsys multivoltage design flow, complete information about the level-shifter characteristics is required. Implementation tools need the following information from the cell library:

- Which power and ground pin of the level shifter is used for voltage boundary hookup during level shifter insertion. This information allows the optimization tools to determine on which side of the voltage boundary a particular level shifter is allowed.
- Which voltage conversions the particular level shifter can handle. Specifically, does the level shifter 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 under all operating conditions.

---

## **Operating Voltages**

In a multivoltage design, each design instance can operate at its specified operating voltage. Therefore, it is important that each voltage 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.

---

## **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.

Although the functionality of a level shifter is that of a buffer, it has two unique properties:

- 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

---

## Syntax

The syntax for level-shifter cells and isolation cells is as follows:

```
cell(level_shifter) {
    is_level_shifter : true ;
    ...
    pin (enable) {
        level_shifter_enable_pin : true ;
        ...
    }/* End pin group */
    ...
}/* End Cell group */

cell(isolation_cell) {
    is_isolation_cell : true ;
    ...
    pin (enable) {
        isolation_cell_enable_pin : true ;
        ...
    }/* End pin group */
    ...
}/* End Cell group */
```

---

## Cell-Level Attributes

This section describes cell-level attributes for level-shifter cells and isolation cells.

### **is\_level\_shifter Attribute**

The `is_level_shifter` 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 the same as any ordinary standard cell.

### **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 the same as any ordinary standard cell.

### **level\_shifter\_type Attribute**

The `level_shifter_type` 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 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 are used to specify the partial voltage swing of an input pin (that is, to specify only partial swings rather than the full rail-to-rail swing). Note that `input_voltage_range` is not related to the voltage swing.

Note:

The `input_voltage_range` and `output_voltage_range` attributes should always be defined together. If they are not, Library Compiler 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 are used to specify the partial voltage swing of an output pin (that is, to specify only partial swings rather than the full rail-to-rail swing). Note that `output_voltage_range` is not related to the voltage swing.

Note:

The `input_voltage_range` and `output_voltage_range` attributes should always be defined together. If they are not, Library Compiler issues an error message.

### **`std_cell_main_rail` Attribute**

The `std_cell_main_rail` Boolean attribute is defined in a `primary_power` power pin. When the attribute is set to true, the power pin is used to determine which side of the voltage boundary the power pin is connected to.

## **Pin-Level Attributes**

This section describes pin-level attributes for level-shifter cells and isolation cells.

### **`level_shifter_enable_pin` Attribute**

The `level_shifter_enable_pin` attribute identifies an enable pin of any level-shifter cell. The valid values are true or false. If not specified, the default is false, meaning that the pin is a regular signal pin of the level-shifter cell.

### **`isolation_cell_enable_pin` Attribute**

The `isolation_cell_enable_pin` attribute identifies the enable pin of any isolation cell. The valid values are true or false. If not specified, the default is false, meaning that the pin is a regular signal pin of the isolation cell.

---

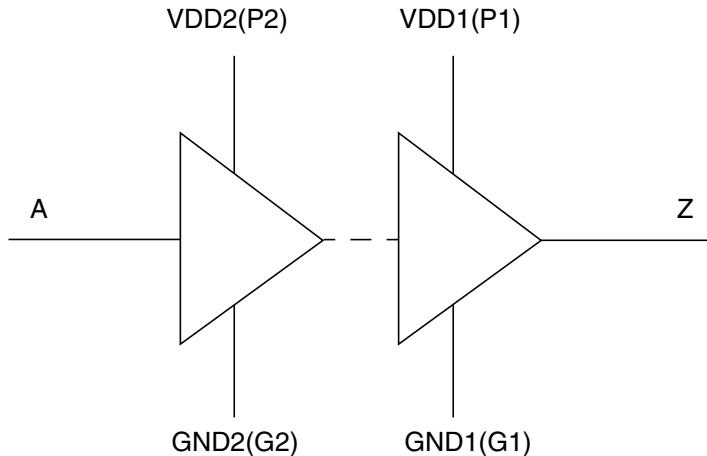
## Level-Shifter Modeling Examples

The following sections provide examples of a buffer-type level-shifter cell, an overdrive level-shifter cell, an enable level-shifter cell, and an enable Isolation cell.

### Simple Buffer-Type Level-Shifter Example

[Figure 2-1](#) shows a simple level-shifter cell model using the power and ground pin syntax and level-shifter modeling attributes.

*Figure 2-1 Buffer Level-Shifter Cell Model*



### Example

```
library(Level_Shifter_Library) {  
    voltage_map(VDD1, 3.0);  
    voltage_map(VDD2, 3.1);  
    voltage_map(GND1, 0.3);  
    voltage_map(GND2, 0.0);  
    operating_conditions(XYZ) {  
        voltage : 3.0;  
        ...  
    }  
    default_operating_conditions : XYZ;  
  
    cell(Buffer_Type_Level_shifter) {  
        is_level_shifter : true;  
        level_shifter_type : HL_LH ;
```

```

input_voltage_range ( 0.9 , 3.1);
output_voltage_range (0.9 , 3.1);

pg_pin(P1) {
    voltage_name : VDD1;
    pg_type : primary_power;
    std_cell_main_rail : true;
}
pg_pin(P2) {
    voltage_name : VDD2;
    pg_type : primary_power;
}
pg_pin(G1) {
    voltage_name : GND1;
    pg_type : primary_ground;
}
pg_pin(G2) {
    voltage_name : GND2;
    pg_type : primary_ground;
}

leakage_power() {
    when : "!A";
    value : 1.5;
    related_pg_pin : P1;
}
leakage_power() {
    when : "!A";
    value : 2.7;
    related_pg_pin : P2;
}
leakage_power() {
    when : "!A";
    value : -0.1;
    related_pg_pin : G1;
}

pin(A) {
    direction : input;
    related_power_pin : P2;
    related_ground_pin : G2;
    input_signal_level_low : 2.15;
    input_signal_level_high : 2.85;
}
pin(Z) {
    direction : output;
    related_power_pin : P1;
}

```

```

related_ground_pin : G1;
output_signal_level_low : 3.0;
output_signal_level_high : 3.1;

timing() {
    cell_rise(delay_template) {
        ...
    }
    cell_fall(delay_template) {
        ...
    }
}

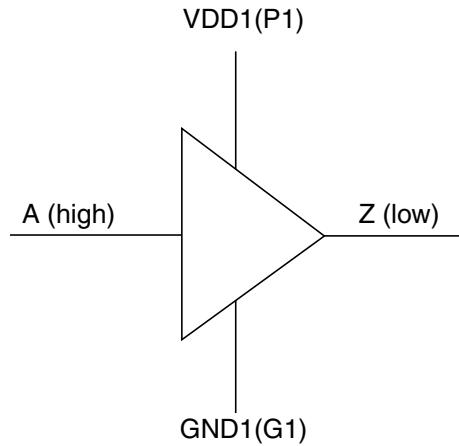
internal_power() {
    related_pin : A;
    when : "A";
    related_pg_pin : P1;
    ...
}
internal_power() {
    related_pin : A;
    when : "A";
    related_pg_pin : P2;
    ...
}
internal_power() {
    related_pin : A;
    when : "A";
    related_pg_pin : G1;
    ...
}
} /* end pin group*/
} /*end cell group*/
...
} /*end library group*/

```

## Overdrive Level-Shifter Example

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 2-2](#) shows an overdrive level-shifter cell schematic.

*Figure 2-2 Overdrive Level-Shifter Cell Schematic*



## Example

```
library(Over_Drive_Library) {

    voltage_map(VDD1, 3.0);
    voltage_map(GND1, 0.3);

    operating_conditions(XYZ) {
        voltage : 3.0;
        ...
    }

    default_operating_conditions : XYZ;

    cell(Over_Drive_Level_Shifter) {
        is_level_shifter : true;
        level_shifter_type : HL ;
        input_voltage_range ( 3.0 , 3.1);
        output_voltage_range (0.9 , 3.1);

        pg_pin(P1) {
            voltage_name : VDD1;
            pg_type : primary_power;
            std_cell_main_rail : true;
        }
        pg_pin(G1) {
            voltage_name : GND1;
            pg_type : primary_ground;
        }
    }

    leakage_power() {
```

```

when : "!A";
value : 1.5;
related_pg_pin : P1;
}
leakage_power() {
when : "!A";
value : -0.1;
related_pg_pin : G1;
}

pin(A) {
direction : input;
input_signal_level : VDD1; /* defining the input_signal_level attribute
identifies an overdrive cell */
input_signal_level_low : 3.0;
input_signal_level_high : 3.1;
}

pin(Z) {
direction : output;
related_power_pin : P1;
related_ground_pin : G1;
output_signal_level_low : 3.0;
output_signal_level_high : 3.1;
timing() {
cell_rise(delay_template) {
...
}
cell_fall(delay_template) {
...
}
internal_power() {
related_pin : A;
when : "A";
related_pg_pin : P1;
...
}
internal_power() {
related_pin : A;
when : "A";
related_pg_pin : G1;
...
}
} /* end pin group*/
} /*end cell group*/
...
} /*end library group*/

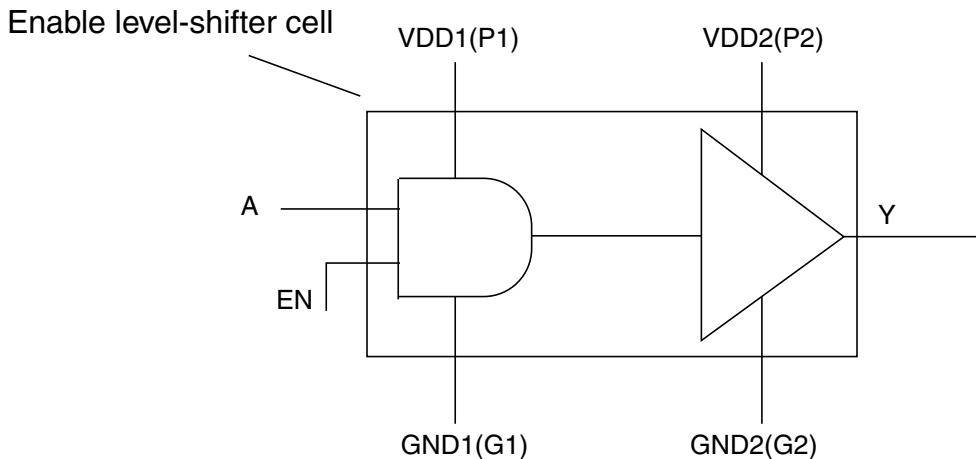
```

## Enable Level-Shifter Example

This section provides an example of a library containing an enable level shifter. This library describes only the portion related to the power and ground pin syntax. Note that the enable level shifter shifts voltage levels between (and locks the output to) a logic value.

Figure 2-3 shows an enable level-shifter cell schematic.

Figure 2-3 Enable Level-Shifter Cell Schematic



## Example

```
library(Over_Drive_Library) {
    voltage_map(VDD1, 3.0);
    voltage_map(VDD2, 4.0);
    voltage_map(GND1, 0.3);
    voltage_map(GND2, 0.0);

    operating_conditions(XYZ) {
        voltage : 3.0;
        ...
    }

    default_operating_conditions : XYZ;

    cell(Enable_Level_Shifter) {
        is_level_shifter : true;
        level_shifter_type : HL_LH ;
        input_voltage_range ( 3.0 , 3.1);
        output_voltage_range (0.9 , 3.1);
    }
}
```

```

pg_pin(P1) {
    voltage_name : VDD1;
    pg_type : primary_power;
    std_cell_main_rail : true;
}
pg_pin(G1) {
    voltage_name : GND1;
    pg_type : primary_ground;
}
pg_pin(P2) {
    voltage_name : VDD2;
    pg_type : primary_power;
}
pg_pin(G2) {
    voltage_name : GND2;
    pg_type : primary_ground;
}

leakage_power() {
    when : "!A";
    value : 1.5;
    related_pg_pin : P1;
}
leakage_power() {
    when : "!A";
    value : 0.1;
    related_pg_pin : G1;
}
leakage_power() {
    when : "!A";
    value : 1.5;
    related_pg_pin : P2;
}

pin(A) {
    direction : input;
    related_power_pin : P1;
    related_ground_pin : G1;
    level_shifter_data_pin : true; /* Optional to specify */
    input_signal_level_low : 2.0;
    input_signal_level_high : 2.1;
}

pin(EN) {
    direction : input;
    related_power_pin : P1;
    related_ground_pin : G1;
}

```

```

level_shifter_enable_pin : true; /* Must specify */
input_signal_level_low : 2.0;
input_signal_level_high : 2.1;
}

pin(Y) {
    direction : output;
    related_power_pin : P2;
    related_ground_pin : G2;
    input_signal_level_low : 3.0;
    input_signal_level_high : 3.1;

    timing() {
        cell_rise(delay_template) {
            ...
        }
        cell_fall(delay_template) {
            ...
        }
    }
    internal_power() {
        related_pin : A;
        when : "A";
        related_pg_pin : P1;
        ...
    }
    internal_power() {
        related_pin : A;
        when : "A";
        related_pg_pin : P2;
        ...
    }
    internal_power() {
        related_pin : A;
        when : "A";
        related_pg_pin : G1;
        ...
    }
}/* end pin group*/
}/*end cell group*/
...
}/*end library group*/

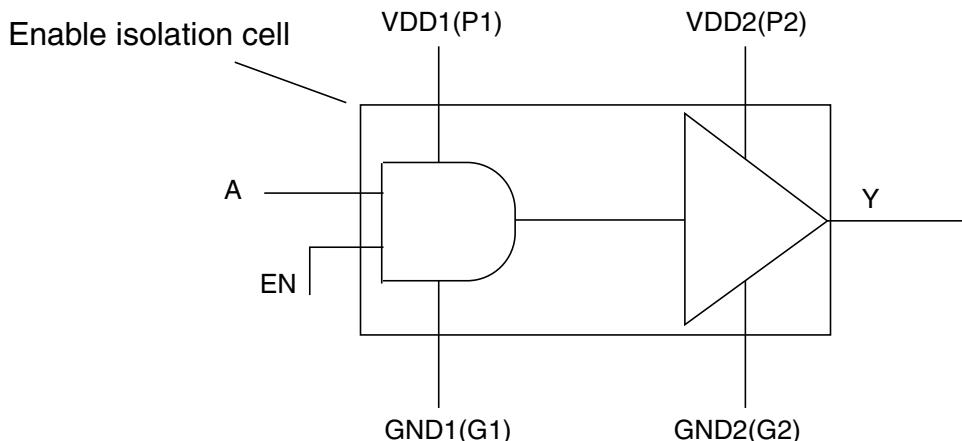
```

## Enable Isolation Cell Example

This section provides an example of a library containing an enable isolation cell. Note that an enable isolation cell cannot shift the voltage levels like a level-shifter cell can. All other characteristics are the same between a level-shifter cell and an enable isolation cell.

[Figure 2-4](#) shows an enable isolation cell schematic. The library in this example describes only the portion related to the power and ground pin syntax.

*Figure 2-4 Enable Isolation Cell Schematic*



### Example

```
library(Over_Drive_Library) {  
  
    voltage_map(VDD1, 3.0);  
    voltage_map(VDD2, 3.0);  
    voltage_map(GND1, 0.3);  
    voltage_map(GND2, 0.0);  
  
    operating_conditions(XYZ) {  
        voltage : 3.0;  
        ...  
    }  
  
    default_operating_conditions : XYZ;
```

```

cell(Enable_Level_Shifter) {
    is_isolation_cell : true;

    pg_pin(P1) {
        voltage_name : VDD1;
        pg_type : primary_power;
        std_cell_main_rail : true;
    }
    pg_pin(G1) {
        voltage_name : GND1;
        pg_type : primary_ground;
    }
    pg_pin(P2) {
        voltage_name : VDD2;
        pg_type : primary_power;
    }
    pg_pin(G2) {
        voltage_name : GND2;
        pg_type : primary_ground;
    }

    leakage_power() {
        when : "!A";
        value : 1.5;
        related_pg_pin : P1;
    }
    leakage_power() {
        when : "!A";
        value : 0.1;
        related_pg_pin : G1;
    }
    leakage_power() {
        when : "!A";
        value : 1.5;
        related_pg_pin : P2;
    }

    pin(A) {
        direction : input;
        related_power_pin : P1;
        related_ground_pin : G1;
        level_shifter_data_pin : true; /* Optional to specify */
        input_signal_level_low : 2.0;
        input_signal_level_high : 2.1;
    }

    pin(EN) {
        direction : input;

```

```

related_power_pin : P1;
related_ground_pin : G1;
isolation_cell_enable_pin : true; /* Must specify */
input_signal_level_low : 2.0;
input_signal_level_high : 2.1;
}

pin(Y) {
    direction : output;
    related_power_pin : P2;
    related_ground_pin : G2;
    input_signal_level_low : 3.0;
    input_signal_level_high : 3.1;

    timing() {
        cell_rise(delay_template) {
            ...
        }
        cell_fall(delay_template) {
            ...
        }
    }
}

internal_power() {
    related_pin : A;
    when : "A";
    related_pg_pin : P1;
    ...
}
internal_power() {
    related_pin : A;
    when : "A";
    related_pg_pin : P2;
    ...
}
internal_power() {
    related_pin : A;
    when : "A";
    related_pg_pin : G1;
    ...
}
}/* end pin group*/
}/*end cell group*/
...
}/*end library group*/

```

---

## Library Checking Rules

Library Compiler checks the following conditions on level-shifter cells and isolation cells and quits with an error if the conditions are not met:

- The `level_shifter_type`, `input_voltage_range`, `output_voltage_range`, and `std_cell_main_rail` attributes must be specified only in a level-shifter cell that has the `is_level_shifter` attribute set to true.
- The `std_cell_main_rail` attribute must satisfy the following properties:
  - The attribute must be specified as the `related_power_pin` value of one of the level-shifter signal pins.
  - The attribute must be defined in a `primary_power` power and ground pin.
- The `input_voltage_range` and `output_voltage_range` attributes should always be defined together. If they are not, Library Compiler issues an error message.
- The `input_voltage_range` and `output_voltage_range` `lower_bound` value must be less than or equal to the `upper_bound` value, as shown:

```
input_voltage_range (1.0, 2.0);
```

Library Compiler issues an error if `lower_bound` (the first value) is greater than `upper_bound` (the second value).

- Pin-level attributes, `level_shifter_data_pin` and `isolation_cell_data_pin`, can appear only in a level shifter or an isolation cell indicated by the `is_level_shifter` and `is_isolation_cell` cell attributes, respectively.

- The `input_voltage_range` and `output_voltage_range` values must be consistent with the type of level shifter.

In the following example,

```
cell <level_shifter_cell_name> {
    is_level_shifter : true;
    level_shifter_type : <level_shifter_type_value>;;
    input_voltage_range (<float11>, <float12>);
    output_voltage_range( <float21>, <float22>);
}
```

your tool must ensure that the following constraints are met:

- If the `level_shifter_type` value is HL, then `float12` cannot be less than or equal to `float21`.
- If the `level_shifter_type` value is LH, then `float11` cannot be greater than or equal to `float22`.
- The pin-level attributes `level_shifter_data_pin` and `isolation_cell_data_pin` can be defined only on an input pin.
- The pin-level attributes `level_shifter_enable_pin` and `isolation_cell_enable_pin` can be defined only on an enable input pin.
- A level-shifter or an isolation cell must be a cell satisfying one of the following conditions. If none of the conditions is met, Library Compiler issues an error.
  - Has two pins, including one input pin and one output pin
  - Has three pins, including one input pin, one `level_shifter_enable_pin` or `isolation_cell_enable_pin`, and one output pin

- For a level shifter, each of the input pins should define the `input_signal_level` value or the `related_power_pin` and `related_ground_pin` as a pair, or both. Similarly, the output pins should define the `related_power_pin` and `related_ground_pin` value as a pair.
- For a level shifter, the `input_signal_level` value of the data input pin and `output_signal_level` value of the output pin should be different. For an isolation cell, the `input_signal_level` value of the data input pin and the `output_signal_level` value of the output pin should be the same.

This also applies if the `related_power_pin` and the `related_ground_pin` attributes only have been specified. In this case, the `related_power_pin` value of the input side cannot be equal to the `related_power_pin` value of the output pin for a level shifter and should be the same for an isolation cell.

#### Note:

For lever shifters and isolation cells, Library Compiler does not check the `input_signal_level` value of the enable input pin against the `input_signal_level` value of the data input pin, the `output_signal_level` value of the output pin, or the `related_power_pin` and `related_ground_pin` pairs that are specified on the input and the output pin.

---

## Switch Cell Modeling

Switch cells are used to reduce power. They are divided into the following two classes:

- Coarse grain

This type of 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

This type of cell has an embedded switch pin that can be used to turn off the cell when it is inactive in order to reduce the leakage power.

Currently, Library Compiler supports only coarse-grain switch 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).
- 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.

- 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/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 Z.

## 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;
direction : input ;
```

```

...
}

/* virtual PG 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>; /* source */
related_internal_pg_pin : <Virtual VDD>; /* drain */
values("<float>, ...");
}

pin (<input_pin_name>) {
direction : input;
switch_pin : true;
...
}

/*
 * "acknowledge output pin uses "function" to
 * represent the propagated switching signal */
pin(<acknowledge_output_pin_name>) {
...
function : "<function_string>";
direction : output;
...
} /* end pin group */
} /* end cell group */

```

---

## **Cell-Level Attribute**

The following attribute is a cell-level attribute for switch cells.

### **switch\_cell\_type Attribute**

The `switch_cell_type` attribute provides a complete description of the switch cell so that Library Compiler does not need to infer the switch cell type from the cell modeling description.

---

## **Pin-Level Attributes**

The following attributes are pin-level attributes for 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.

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 a Z 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` syntax is modified from the Boolean `function` attribute. In addition to its existing usage for signal output pins, it is used for the coarse-grain switch cells' virtual VDD output pins to represent the propagated power level through the switch as a function of the input power and ground pins. This is usually a logical buffer and is useful in cases where the VDD and VSS connectivity might be erroneously reversed.

---

## **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.

---

## **lu\_table\_template Group**

The library-level `lu_table_template` defines the variables and the size of the `dc_current` tables specified at the cell level.

---

## **dc\_current Group**

The cell-level `dc_current` group is used to model the steady state current information, similar to the `lu_table_template` group. It 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 is used to specify 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.

---

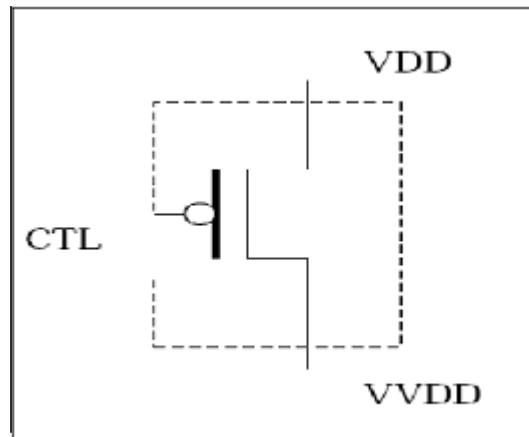
## Switch-Cell Modeling Examples

The following sections provide examples for a simple coarse-grain switch cell and a complex coarse-grain switch cell.

### Simple Coarse-Grain Switch Cell Example

Figure 2-5 shows a simple coarse-grain switch cell.

Figure 2-5 Simple Coarse-Grain Switch Cell



### Example

```
Library (sample_lib) {
    ...
lu_template ( ivt1 ) {
variable_1 : input_voltage;
/* "input_voltage" is used to specify the input_voltage values of the switch pin */
variable_2 : output_voltage;
/* "output_voltage" is used to specify the voltage difference values between the
related_pg_pin and related_internal_pg_pin */
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 ( 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;
direction : output ;
switch_function : "CTL" ;
pg_function : "VDD" ;
}

pin ( CTL ) {
switch_pin : true;
capacitance: 0.034;
}

/* I/V curve information */
dc_current ( ivt1 ) {
related_switch_pin : CTL; /* control pin */
related_pg_pin : VDD; /* source */
related_internal_pg_pin : VVDD; /* drain */
/* index_1 is used to specify the input_voltage values of the switch pin */
/* index_2 is used to specify the voltage difference values between the
related_pg_pin and related_internal_pg_pin */
values("0.010, 0.020, 0.030, 0.030, 0.030", \
       "0.011, 0.021, 0.031, 0.031, 0.031", \
       "0.012, 0.022, 0.032, 0.032, 0.032", \
       "0.013, 0.023, 0.033, 0.033, 0.033", \
       "0.014, 0.024, 0.034, 0.034, 0.034");
}

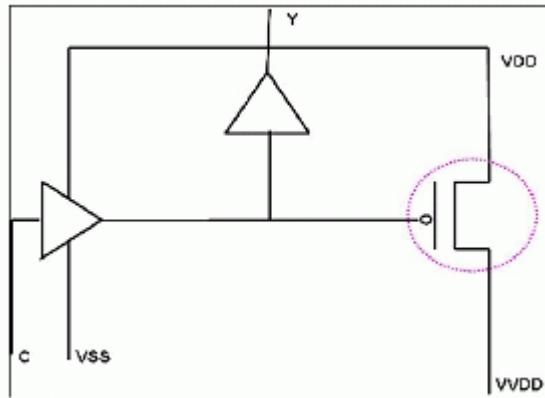
...
}
}

```

## Complex Coarse-Grain Switch Cell Example

Figure 2-6 shows a complex coarse-grain switch cell.

Figure 2-6 Complex Coarse-Grain Switch Cell



### Example

```
library (sample_lib) {
...
lu_template ( ivt1 ) {
variable_1 : input_voltage;
/* "input_voltage" specifies the input_voltage values of the switch pin */
variable_2 : output_voltage; /* "output_voltage" specifies the voltage difference
values between the related_pg_pin and related_internal_pg_pin */

index_1 ( "0.1, 0.2, 0.4, 0.8, 1.0" ); /* Note that in this example, the
index_1(input_voltage) models the voltage values of internal node which is
directly attached to the transistor that has the "VWDD" pg_pin as the current
sink pin. */

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 ;
switch_function : "C";
pg_function : "VDD" ;
}

pg_pin ( VSS ) {
pg_type : primary_ground;
voltage_name : VSS;
direction : output ;
}

pin ( C ) {
direction : input;
switch_pin : true;
capacitance: 0.034;
...
}

/* I/V curve information */
dc_current ( iv1 ) {
related_switch_pin : C; /* 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, 0.031, 0.031", \
       "0.012, 0.022, 0.032, 0.032, 0.032", \
       "0.013, 0.023, 0.033, 0.033, 0.033", \
       "0.014, 0.024, 0.034, 0.034, 0.034");
...
}

...
}

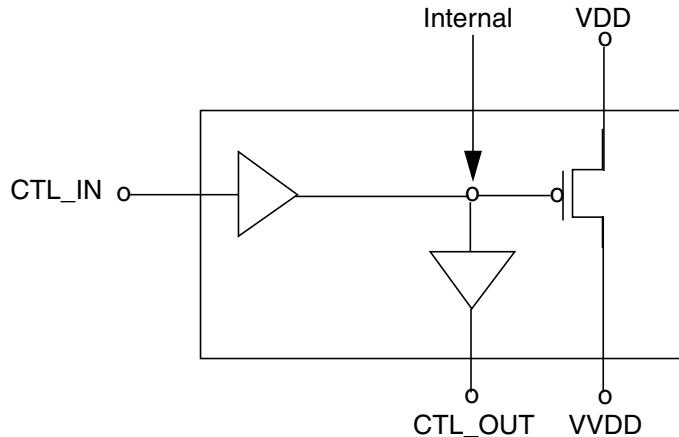
pin ( Y) {
direction : output;
function : "C";
...
}
} /* end cell group*/
} /* end library group*/

```

## Complex Coarse-Grain Switch Cell With an Internal Switch Pin Example

Figure 2-7 shows a complex coarse-grain switch cell with an internal switch pin.

Figure 2-7 Complex Coarse-Grain Switch Cell With an Internal Switch Pin



### Example

```
library (Complex(CG_lib) {  
  ...  
  define(always_on, pin, float);  
  voltage_map(VDD, 3.0);  
  voltage_map(VVDD, 3.1);  
  voltage_map(VSS, 0.0);  
  
  operating_conditions(XYZ) {  
    process : 1.0;  
    voltage : 3.0;  
    temperature : 25.0;  
  }  
  default_operating_conditions : XYZ;  
  
  lu_template ( ivt1 ) {  
    variable_1 : input_voltage; /* "input_voltage" specifies the  
      input_voltage values of the switch pin */  
  
    variable_2 : output_voltage; /* "output_voltage" specifies  
      the difference between the related_pg_pin and  
      related_internal_pg_pin voltage values */  
  }  
}
```

```

index_1 ( "0.1, 0.2, 0.4, 0.8, 1.0" ); /* In this example,
the index_1(input_voltage) models the voltage values of the
internal node, which is directly attached to the transistor
that has the "VVDD" pg_pin as the current sink pin. */

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 : "CTL_IN" ;
    }
    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;
    index_1 ("0.0000, 0.1000");
    index_2 ("0.0000, 0.1000");
    values ("-0.0050, -0.0049", \
            "-0.0048, -0.0047");
}
pin(CTL_IN) {
    switch_pin : true;
    direction : input;
    capacitance : 0.002035;
    always_on : true;
}

...
}

```

```

pin(CTL_OUT) {
    switch_pin : true;
    direction : output;
    function : "CTL_IN";
    capacitance : 0.002035;
timing() {
    related_pin : "CTL_IN"
    cell_rise( scalar ) {
        values("0.0");
    }
    rise_transition( scalar ) {
        values("0.0");
    }
    cell_fall( scalar ) {
        values("0.0");
    }
    fall_transition( scalar ) {
        values("0.0");
    }
}
pin(INTERNAL) {
    direction : internal;
    capacitance : 0.002035;
timing() {
    related_pin : "CTL_IN"
    cell_rise( scalar ) {
        values("0.0");
    }
    rise_transition( scalar ) {
        values("0.0");
    }
    cell_fall( scalar ) {
        values("0.0");
    }
    fall_transition( scalar ) {
        values("0.0");
    }
}
}
}

```

---

## Library Checking Rules

Library Compiler performs the following checks for switch cells and quits with an error if the conditions are not satisfied:

- The switch function can contain only switch pins whose input pin (set by the `switch_pin` attribute) is set to true. (The function attribute can also contain switch pins for a coarse-grain switch cell.)

Note:

Library Compiler does not check whether the timing should be modeled with composite current source (CCS) timing or nonlinear delay model (NLDM) timing, and it does not check whether the power should be modeled with CCS power or NLDM power.

A coarse-grain switch cell must satisfy the following rules:

- It must define `switch_cell_type` with the `coarse_grain` value.
- It must define the cell-level `switch_function` attribute to identify the control logic of its switch pins.
- It must have at least one switch pin.
- It must have at least one controlled power and ground pin and one regular power and ground pin (that is, a virtual VSS `pg_pin` and a VSS `pg_pin` or a virtual VDD `pg_pin` and a VDD `pg_pin`). Each of these output power pins must have a `pg_function` attribute Boolean expression containing input power pins.
- The `pg_function` attribute must contain only the input power and ground pins.

- The `related_switch_pin` value can be either an internal pin or a switch pin. If the `switch_pin` attribute is set to true within a pin, it is identified as a switch pin of the cell. If the `direction` attribute is set to internal within a pin group, it is identified as an internal pin of the cell. It is permitted to have the `related_pin` value within an internal pin in a valid `switch_pin` of the cell.

# 3

## Delay Models

---

Design Compiler uses 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 used in the timing delay calculations are dependent on the delay model used. This chapter describes the timing parameters and environment attributes used by the various delay models, in the following sections:

- [CMOS Generic Delay Model](#)
- [CMOS Nonlinear Delay Model](#)
- [Scalable Polynomial Delay Model](#)
- [CMOS Piecewise Linear Delay Model](#)
- [Delay Calculation Module \(DCM\) Delay Model](#)

The delay model you choose applies to all cells in the library; you cannot mix delay models within a single library.

Each component of each delay model equation is affected differently by variations in the manufacturing process, operating temperature, and supply voltage. Consequently, separate scaling factors are applied to each component of each delay equation.

You define the appropriate delay model with the following syntax:  
(table\_lookup applies to the CMOS nonlinear delay model):

```
delay_model : generic_cmos | table_lookup | piecewise_cmos | cmos2 | dcm |
    polynomial ;
```

---

## CMOS Generic Delay Model

To understand the effects of the timing parameters and library environment attributes on the timing delay calculations when you use a CMOS Generic Delay Model, you need to know about the concepts discussed in the following sections:

- Total Delay Equation
- Total Delay Scaling
- Slope Delay
- Intrinsic Delay
- Transition Delay
- Connect Delay
- Interconnect Delay
- Delay Calculation Example

For additional information about how Design Compiler uses timing delay information, see the Design Compiler documentation.

---

### Total Delay Equation

Delay analysis involves calculating the delay between the input pin of a gate and the input pin of the next gate. This timing delay includes the connect delay from the driving pins to the load pins.

The following delay model divides the total delay of a network into four physical components whose sum is the total delay through a circuit element:

$$D_{\text{total}} = D_I + D_S + D_C + D_T$$

$D_I$

Intrinsic delay inherent in the gate and independent of any particular instantiation.

$D_S$

Slope delay caused by the ramp time of the input signal.

$D_C$

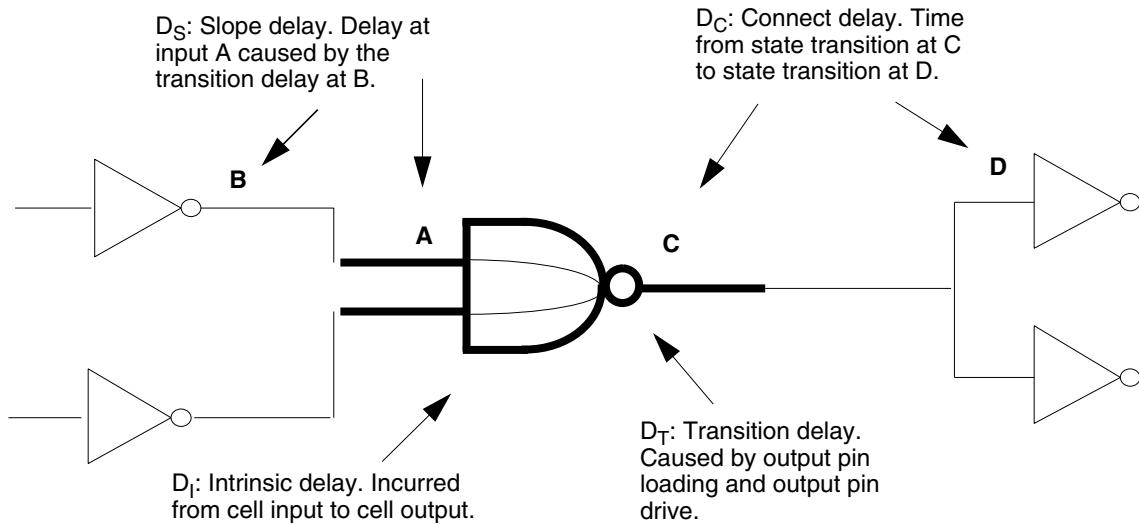
Connect media delay to an input pin (wire delay).

$D_T$

Transition delay caused by loading of the output pin.

[Figure 3-1](#) shows the components of the total delay.

*Figure 3-1 Delay Equation Components for CMOS Generic Delay Model*



Total delay is a monotonically increasing function of the four components. Each component monotonically increases its primary parameters. All timing parameters use the same units in a library description.

You do not need to designate the units you used to perform the delay calculations, but you must use a consistent unit throughout your library to ensure accurate results.

You can use several different delay models with the same analysis method. For example, setting the slope and interconnect components to zero effect yields a simple rise or fall model. The rise and fall delay parameters are separated in the formulated equation.

All four components of the delay equation are affected in different degrees by variations in manufacturing process, operating temperature, and supply voltage. Consequently, separate scaling factors are applied to each component of the delay equation.

---

## Total Delay Scaling

When calculating total delay, Design Compiler scales each parameter of  $D_{total}$  individually.

Each component of the total delay has its own global parameters to model the effects on the nominal case of variations in process, temperature, and voltage.

The actual delays used for analysis are scaled versions of the four components of the total delay equation, as shown here:

$$D_{total} = D_I + D_S + D_C + D_T$$

The equation for the scaled versions of the individual components' total delay is

$$(D_{\text{scaled}} = D(1 + \Delta_{\text{process}} \times K_{\text{process}})(1 + \Delta_{\text{temp}} \times K_{\text{temp}})(1 + \Delta_{\text{voltage}} \times K_{\text{voltage}}))$$

D

A component of the total delay equation:  $D_I$ ,  $D_S$ ,  $D_C$ , or  $D_T$ .

$\Delta_{\text{process}}$ ,  $\Delta_{\text{temp}}$ , and  $\Delta_{\text{voltage}}$

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. Here is one illustration:

$$\Delta_{\text{process}} = \text{process} - \text{nom\_process}$$

$K_{\text{process}}$ ,  $K_{\text{temp}}$ , and  $K_{\text{voltage}}$

Represent the scaling factors of the individual components of each member of the total delay equation.

The total scaled delay equation takes this form:

$$D_{\text{total(scaled)}} = D_I(\text{scaled}) + D_S(\text{scaled}) + D_C(\text{scaled}) + D_T(\text{scaled})$$

Scaling factors for process, temperature, and voltage are called *k-factors* (scaling factor attributes that begin with k\_) and are defined at the library group level in the technology library.

Nominal operating conditions and operating\_conditions groups are also defined at the library level. For a description of these statements, see “[Defining Operating Conditions](#)” on page 12-15 of this guide and the *Library Compiler Reference Manual: Technology and Symbol Libraries*.

[Example 3-2 on page 3-17](#) shows a complete scaled delay equation for a rise delay.

---

## Slope Delay

The slope delay of an element ( $D_S$ ) is the incremental time delay caused by slowly changing input signals. Normally, library cells are characterized with a nominal ramp time (1.5 ns is common) to determine the delay from input pin to output pin.

In some technologies, this delay is a strong function of the ramp time. In other technologies, the delay does not vary—even over a wide range of ramp values.

$D_S$  is included in the delay equation to allow additional accurate modeling of technologies that are sensitive to input ramp time (see “[Total Delay Equation](#)” on page 3-3).

$D$  is calculated with the transition delay at the previous output pin, plus a slope sensitivity factor, as shown here:

$$D_S = S_S \times D_{T(\text{prevstage})}$$

This equation calculates both the rise and fall delays. Where applicable, use the `_rise` parameter to calculate the rise delay and the `_fall` parameter to calculate the fall delay.

$D_S$

Transition delay is calculated at the previous stage of logic. Therefore, the calculation of  $D_S$  enforces a global order on local analysis. If you assume that all delay calculations are performed

with a minimum achievable ramp time (possibly zero), omitting the slope in timing analysis prevents overestimation of the total delay.

### $S_S$

Slope sensitivity factor. This 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. The attributes that define it in the `timing` group of the driving pin are `slope_rise` and `slope_fall`.

### $D_T(\text{prevstage})$

The transition delay calculated at the previous output pin.

The  $D_T$  (transition delay) component of the equation must be calculated according to the standard delay model equation of  $D_T$ .

The total slope delay is calculated by scaling of the constant values by their corresponding k-factors (see “[Total Delay Scaling](#)” on [page 3-5](#)).

---

## Intrinsic Delay

The intrinsic delay of a circuit element ( $D_I$ ) is the portion of the total delay that is independent of the circuit element’s usage. This portion is the fixed (or zero load) delay from the input pin to the output pin of a circuit element.

Constant values for the intrinsic delay are stored in the `timing` group of the driving pin as floating-point numbers in the `intrinsic_rise` and `intrinsic_fall` attributes:

Total intrinsic delay is calculated by scaling of these constant values by their corresponding k-factors (see “[Total Delay Scaling](#)” on page 3-5).

---

## Transition Delay

The transition delay of a circuit element is the time it takes the driving pin to change state. The transition time of the output pin on a net is a function of the capacitance of all pins on the net and the capacitance of the interconnect network that ties the pins together.

Here is the equation for transition delay:

$$D_T = R_{\text{driver}} (C_{\text{wire}} + C_{\text{pins}}) / \text{number\_non\_three\_state\_drivers}$$

This equation calculates the rise and fall delays. The components of this equation are described below. Where applicable, use the `_rise` parameter to calculate the rise delay and the `_fall` parameter to calculate the fall delay.

`Rdriver`

The resistance of the timing arc is stored in the `timing` group of the driving pin as floating-point numbers in the `rise_resistance` and `fall_resistance` attributes:

Resistance values in the piecewise linear delay model (`rise_pin_resistance` and `fall_pin_resistance`) are not valid in the standard delay model.

`Cwire`

The estimated wire capacitance for the net attached to the head of the timing arc. Wire length is computed with the actual number of fanout pins on the net and the `fanout_length` specifications

in the `wire_load` group. The estimated value is scaled by the capacitance factor, which is defined in the `wire_load` group as capacitance. If no `wire_load` group is designated at runtime, the value of  $C_{\text{wire}}$  is 0.

$C_{\text{pins}}$

The sum of all pin capacitances on the net. In each `pin` group on the net, the pins' capacitance value appears as a floating-point number in the capacitance attribute:

`number_non_three_state_drivers`

Parallel drivers are handled by division of the arc transition time by the number of non-three-state drivers on the output net.

Wire resistance and wire capacitance are defined in the `wire_load` groups in the technology library.

The total transition delay is calculated by scaling of the constant values by their corresponding k-factors (see “[Total Delay Scaling](#)” on page 3-5).

---

## Connect Delay

The connect delay of an element ( $D_C$ ) 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 known as time-of-flight delay, which is the time it takes a waveform to travel along a wire.

Library Compiler and Design Compiler support three cases for an estimated interconnect topology. These three cases represent the three possible values of the `operating_conditions` group attribute `tree_type`. See “[operating\\_conditions Group](#)” on [page 12-15](#) for details.

#### `best_case_tree`

The best case models the load pin as physically adjacent to the driver. All the wire capacitance is incurred, but none of the wire resistance must be overcome. The best-case connect delay is calculated from the following equation. Because  $R_{\text{wire}}$  is always zero in this case, the resulting  $D_C$  is always zero.

$$D_{C_{\text{best}}} = R_{\text{wire}}(C_{\text{wire}} + C_{\text{pin}}) = 0$$

#### `worst_case_tree`

The worst case models the load pin at the extreme end of the wire. Each load pin incurs both the full wire capacitance and the full wire resistance, as shown in the following equation.

$$D_{C_{\text{worst}}} = R_{\text{wire}} \left( C_{\text{wire}} + \sum_{\text{pins}} C_{\text{pin}} \right)$$

#### `balanced_tree`

The balanced case models all the load pins on separate, equal branches of the interconnect wire. In the balanced case, each load pin incurs an equal portion of the wire capacitance and wire resistance, as shown in the following equation.

$$D_{C_{\text{balanced}}} = \frac{R_{\text{wire}}}{N} \left( \frac{C_{\text{wire}}}{N} + C_{\text{pin}} \right)$$

Connect delay equations calculate the rise and fall delays. The components of these equations are described as follows:

$R_{\text{wire}}$

Estimated wire resistance on the net, determined by the wire load model. Wire length is computed with a global estimation function whose parameter is the number of fanout pins on the net being estimated. The estimated value is scaled by the resistance factor, which is defined in the `wire_load` group as resistance.

$C_{\text{wire}}$

Estimated wire capacitance on the net attached to the head of the timing arc. Wire length is computed with the actual number of fanout pins on the net and the `fanout_length` specifications in the `wire_load` group. The estimated value is scaled by the capacitance factor, which is defined in the `wire_load` group as capacitance. If no `wire_load` group is specified at runtime, the value of  $C_{\text{wire}}$  is 0.

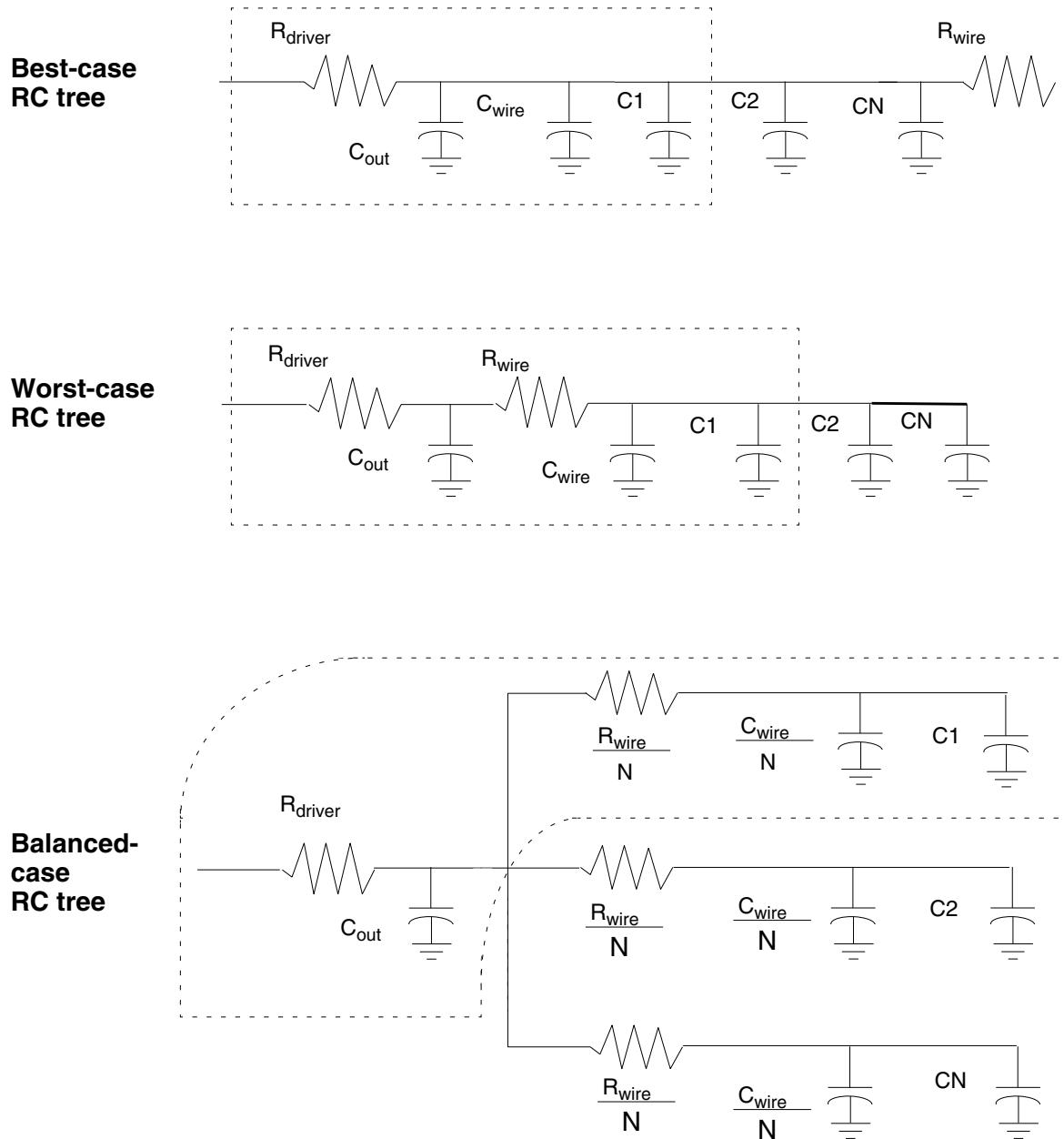
$C_{\text{pin}}$

Capacitance values for the load pin, defined in the `pin` group of the load pin as capacitance.

Scaling the constant values by their corresponding k-factors calculates the total connect time (see “[Total Delay Scaling](#)” on [page 3-5](#)).

In [Figure 3-2](#), dotted lines surround those parameters that are included in the connect delay calculation for one load pin.

*Figure 3-2 Resistance Capacitance Interconnect Tree Topologies for Fanout of N*



The following equations show the expanded  $D_C$  equations for each tree topology and the three calculations for  $D_C$ .

Best-case  $D_C = 0.0$

Worst-case  $D_C =$

$$\begin{aligned} & \text{wire\_resistance} (1 + \Delta_{\text{voltage}} k_{\text{volt\_wire\_res}}) \\ & (1 + \Delta_{\text{temp}} k_{\text{temp\_wire\_res}})(1 + \Delta_{\text{process}} k_{\text{process\_wire\_res}}) \\ & [ (C_{\text{wire}})(1 + \Delta_{\text{voltage}} k_{\text{volt\_wire\_cap}}) \\ & (1 + \Delta_{\text{temp}} k_{\text{temp\_wire\_cap}})(1 + \Delta_{\text{process}} k_{\text{process\_wire\_cap}}) \\ & + \sum_{\text{pins}} \{ (C_{\text{pin}})(1 + \Delta_{\text{voltage}} k_{\text{volt\_pin\_cap}}) \\ & (1 + \Delta_{\text{temp}} k_{\text{temp\_pin\_cap}})(1 + \Delta_{\text{process}} k_{\text{process\_pin\_cap}}) \} ] \end{aligned}$$

Balanced-case  $D_C =$

$$\begin{aligned} & \text{wire\_resistance}/N (1 + \Delta_{\text{voltage}} k_{\text{volt\_wire\_res}}) \\ & (1 + \Delta_{\text{temp}} k_{\text{temp\_wire\_res}})(1 + \Delta_{\text{process}} k_{\text{process\_wire\_res}}) \\ & [ (C_{\text{wire}}/N)(1 + \Delta_{\text{voltage}} k_{\text{volt\_wire\_cap}}) \\ & (1 + \Delta_{\text{temp}} k_{\text{temp\_wire\_cap}})(1 + \Delta_{\text{process}} k_{\text{process\_wire\_cap}}) \\ & + (C_{\text{pin}})(1 + \Delta_{\text{voltage}} k_{\text{volt\_pin\_cap}}) \\ & (1 + \Delta_{\text{temp}} k_{\text{temp\_pin\_cap}})(1 + \Delta_{\text{process}} k_{\text{process\_pin\_cap}})] \end{aligned}$$

Note:

$N$  is the number of load pins on the net.

---

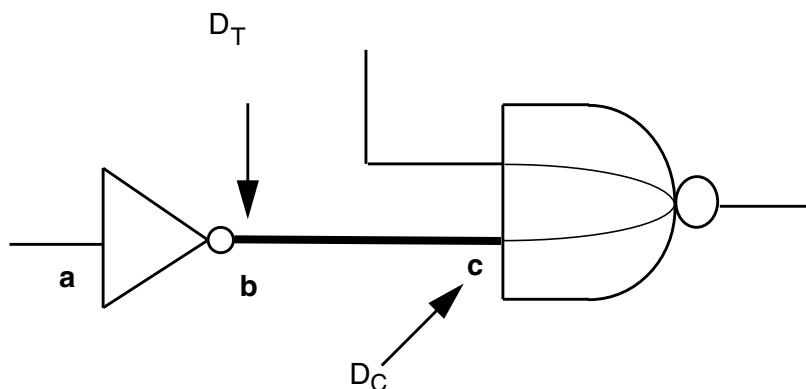
## Interconnect Delay

Interconnect delay is defined as the delay caused by connect delay and fanout. It is calculated as the sum of  $D_T$  and  $D_C$ .

$$D_{\text{interconnect}} = D_T + D_C$$

The interconnect delay is calculated in several different ways. If you take into consideration the path delay computation Design Compiler performs, the most efficient modeling technique is to attribute the transition delay to the output of the driving gate and the connect delay to the input of the driven gate (see [Figure 3-3](#)).

*Figure 3-3 Interconnect Delay Diagram*



In terms of cell modeling, include the capacitance attribute in the `pin` group of the input pin. Give zero capacitance to the `pin` group of the output pin. Resistance is attributed entirely to the output pin. [Example 3-1](#) shows how the interconnect delay is modeled.

*Example 3-1 Modeling Interconnect Delay*

```
cell (inv1) {  
    area : 1 ;  
    pin (a) {  
        direction : input ;
```

```

        capacitance : 1 ;
    }
pin (b) {
    direction : output ;
    capacitance : 0 ;
    timing () {
        related_pin : "a" ;
        rise_resistance : 1.2 ;
        fall_resistance : 0.8 ;
        intrinsic_rise : 1.5 ;
        intrinsic_fall : 1.2 ;
    }
}
cell (NAND1) {
    area : 1.5 ;
    pin (in1) {
        direction : input ;
        capacitance : 1.5 ;
    }
    ...
}
```

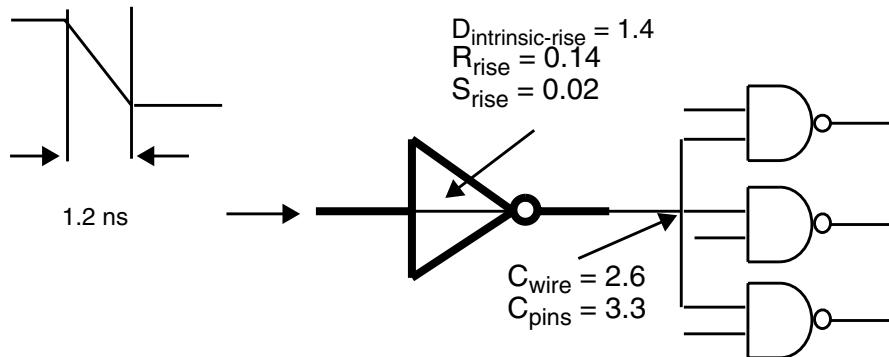
Total interconnect delay is the sum of the scaled versions of  $D_T$  and  $D_C$  (see “Total Delay Scaling” on page 3-5).

---

## Delay Calculation Example

Example 3-2 shows the rise delay parameters for the inverter in Figure 3-4.

Figure 3-4 Rise Delay Diagram



The input pin is driven by a falling signal with a transition delay ( $D_T$ ) of 1.2 ns. This inverter fans out to three NAND gates, each with an input pin capacitance of 1.1. The inverter has an intrinsic rise delay of 1.4, a rise slope sensitivity of 0.02, and an output rise resistance of 0.14. Example 3-2 is the computation for the rise delay of the inverter.

Example 3-2 Rise Delay Computation

```
intrinsic_rise = 1.4
rise_resistance = 0.14
slope_rise = 0.02
Cpins = 3 * (1.1) = 3.3
Cwire = 2.6
DT(fall_previous_stage) = 1.2
DC = 0.0          /* for a best-case RC tree */
Rise Delay = Intrinsic + Slope + Transition + Connect
= 1.4 + (1.2 * 0.02) + (0.14)(3.3 + 2.6) + 0.0
= 1.4 + 0.024 + 0.826 + 0.0
= 2.25
```

**Example 3-2** assumes that the output pin capacitance of the inverter is not modeled as a separate attribute but is included as part of the intrinsic rise value. Assuming a best-case RC-interconnect tree type and an estimated interconnect wire capacitance of 2.6, the rise delay is 2.25.

The following equation shows the complete scaled delay equation for a rise delay (assuming a worst-case interconnect tree).

Rise delay =

$$D_T = + \text{rise\_resistance} (1 + \Delta_{\text{voltage}} k_{\text{volt\_drive\_rise}}) \\ (1 + \Delta_{\text{temp}} k_{\text{temp\_drive\_rise}})(1 + \Delta_{\text{process}} k_{\text{process\_drive\_rise}}) \\ [ (C_{\text{wire}})(1 + \Delta_{\text{voltage}} k_{\text{volt\_wire\_cap}}) \\ (1 + \Delta_{\text{temp}} k_{\text{temp\_wire\_cap}})(1 + \Delta_{\text{process}} k_{\text{process\_wire\_cap}}) ]$$

$$+ \sum_{\text{pins}} \{ (C_{\text{pin}})(1 + D_{\text{voltage}} k_{\text{volt\_pin\_cap}}) \\ (1 + \Delta_{\text{temp}} k_{\text{temp\_pin\_cap}})(1 + \Delta_{\text{process}} k_{\text{process\_pin\_cap}}) \} ]$$

---


$$D_I = \frac{\text{intrinsic\_rise} (1 + \Delta_{\text{voltage}} k_{\text{volt\_intrinsic\_rise}})}{(1 + \Delta_{\text{temp}} k_{\text{temp\_intrinsic\_rise}})(1 + \Delta_{\text{process}} k_{\text{process\_intrinsic\_rise}})}$$

On examination, the equation reveals a subtle modeling point to consider when you develop a library. In many technology modeling schemes, it is the overall delay ( $D_{\text{total}}$ ) that is scaled, rather than the individual components ( $D_I$ ,  $D_S$ ,  $D_T$ , and  $D_C$ ).

$$D_S = \frac{+ (D_{T_{\text{prevstage}}}) (\text{slope\_rise}) (1 + \Delta_{\text{voltage}} k_{\text{volt\_slope\_rise}})}{(1 + \Delta_{\text{temp}} k_{\text{temp\_slope\_rise}}) (1 + \Delta_{\text{process}} k_{\text{process\_slope\_rise}})}$$


---

$$D_C = \frac{+ \text{wire\_resistance} (1 + D_{\text{voltage}} k_{\text{volt\_wire\_res}})}{(1 + \Delta_{\text{temp}} k_{\text{temp\_wire\_res}}) (1 + \Delta_{\text{process}} k_{\text{process\_wire\_res}})} \\ [ (C_{\text{wire}}) (1 + \Delta_{\text{voltage}} k_{\text{volt\_wire\_cap}}) (1 + \Delta_{\text{temp}} k_{\text{temp\_wire\_cap}}) \\ (1 + \Delta_{\text{process}} k_{\text{process\_wire\_cap}}) \\ + \sum_{\text{pins}} \{ (C_{\text{pin}}) (1 + \Delta_{\text{voltage}} k_{\text{volt\_pin\_cap}}) \\ (1 + \Delta_{\text{temp}} k_{\text{temp\_pin\_cap}}) (1 + \Delta_{\text{process}} k_{\text{process\_pin\_cap}}) \} ]$$

In the equation, this effect is achieved by use of only nonzero k-factors for intrinsic delay, pin resistance, wire resistance, and slope-sensitivity factors. Using the same process, temperature, and voltage variations for the respective components of the delay equation produces the same results as overall scaling.

---

## CMOS Nonlinear Delay Model

The CMOS nonlinear delay model enhances Library Compiler and the timing analyzer to improve the accuracy of delay predictions. The CMOS nonlinear delay model uses lookup tables and interpolation to compute delays. As a result, the model is flexible enough to provide close timing correlation with a wide variety of submicron delay modeling schemes.

To create a CMOS nonlinear delay model, you need to understand the following concepts and tasks:

- The total delay equation
- Cell delay ( $D_{cell}$ )
- Propagation delay ( $D_{propagation}$ )
- Transition delay ( $D_{transition}$ )
- Connect delay ( $D_C$ )
- CMOS nonlinear delay model calculation
- Process, voltage, and temperature scaling

---

## Total Delay Equation

Delay analysis involves calculating the total delay of a logic stage—the delay between the input pin of a gate and the input pin of the next gate.

The total delay has two major components:  $D_{cell}$  and  $D_C$ .

$$D_{\text{total}} = D_{\text{cell}} + D_C$$

$D_{\text{cell}}$

The delay contributed by the gate itself, typically defined as the 50 percent input pin voltage to 50 percent output voltage.  $D_{\text{cell}}$  is computed in two ways, depending on the timing data provided. See “[Cell Delay](#)” on page 3-23 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. Connect delay is calculated by the same method used in other delay equations.

In the absence of a  $D_{\text{cell}}$  component,  $D_{\text{transition}}$ , which corresponds to the time required for the output pin to change state, is used. This is sometimes referred to as the output ramp time.

$D_{\text{transition}}$

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  $D_{\text{transition}}$  involves performing table lookup and interpolation.

The CMOS nonlinear delay model supports two methods of computing  $D_{\text{cell}}$ . 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:

$$D_{\text{cell}} = D_{\text{propagation}} + D_{\text{transition}}$$

The typical measurement for  $D_{\text{propagation}}$  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  $D_{\text{transition}}$  value of 10 percent to 50 percent output voltage added to  $D_{\text{propagation}}$ , 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

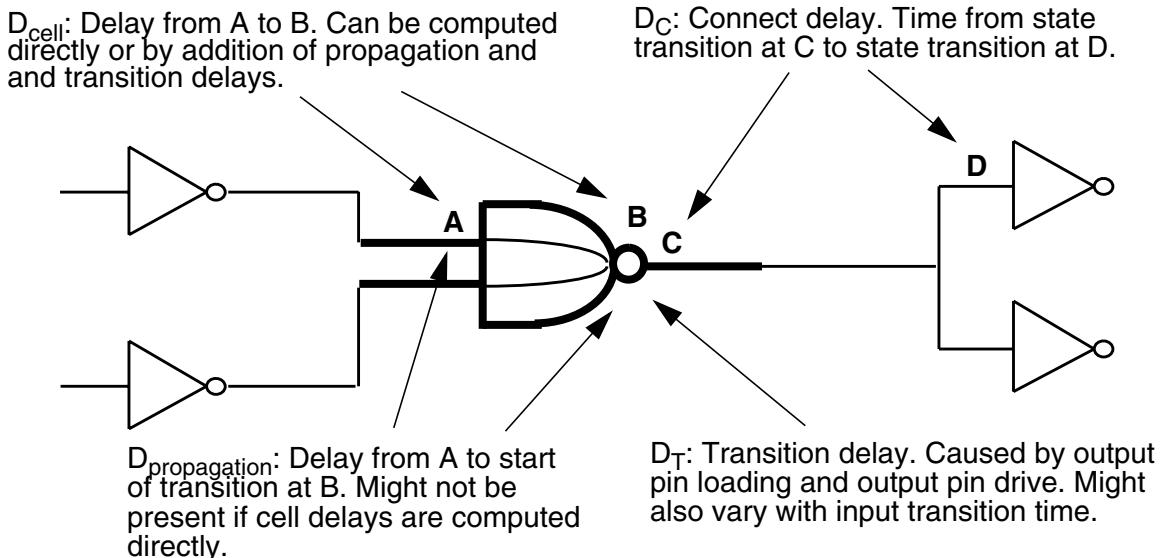
$$D_{\text{total}} = D_{\text{cell}} + D_C$$

If propagation delay tables are provided instead, the total delay equation is

$$D_{\text{total}} = D_{\text{propagation}} + D_{\text{transition}} + D_C$$

[Figure 3-5](#) shows the delay equation components.

*Figure 3-5 Delay Equation Components for CMOS Nonlinear Delay Model*



## Cell Delay

Cell delay ( $D_{cell}$ ) is typically defined as the 50 percent input pin voltage to 50 percent 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 are for a timing arc, do not specify propagation delay tables for that arc.

For more information about how to define delay tables, see Chapter 3, “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 4, “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 4, “Timing Arcs.”](#)

---

## Connect Delay

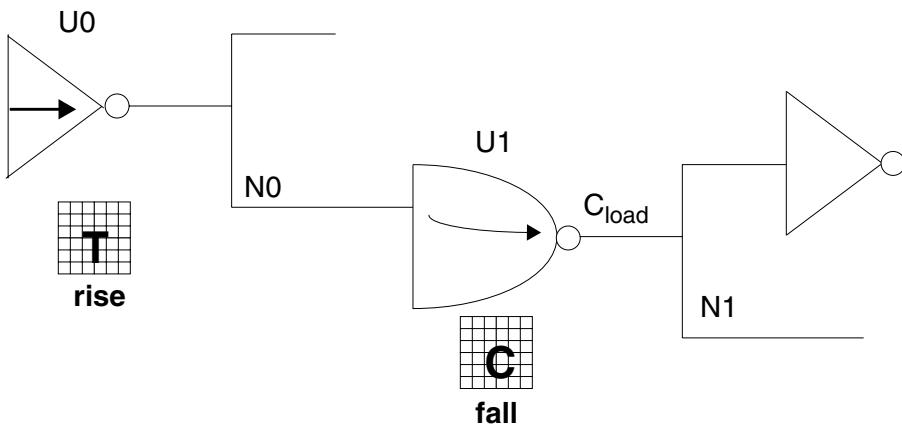
$D_C$  is the time it takes the voltage at an input pin to charge after the driving output pin has made a transition. This delay is also called the time-of-flight delay—the time it takes for a waveform to travel along a wire. The CMOS nonlinear delay model computes connect delay by using the same equations as the generic delay model.

---

## CMOS Nonlinear Delay Model Calculation

[Figure 3-6](#) is a schematic representation of the total delay through a cell. [Figure 3-7](#) 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 3-6](#), 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 3-6 Nonlinear Delay Calculation Schematic



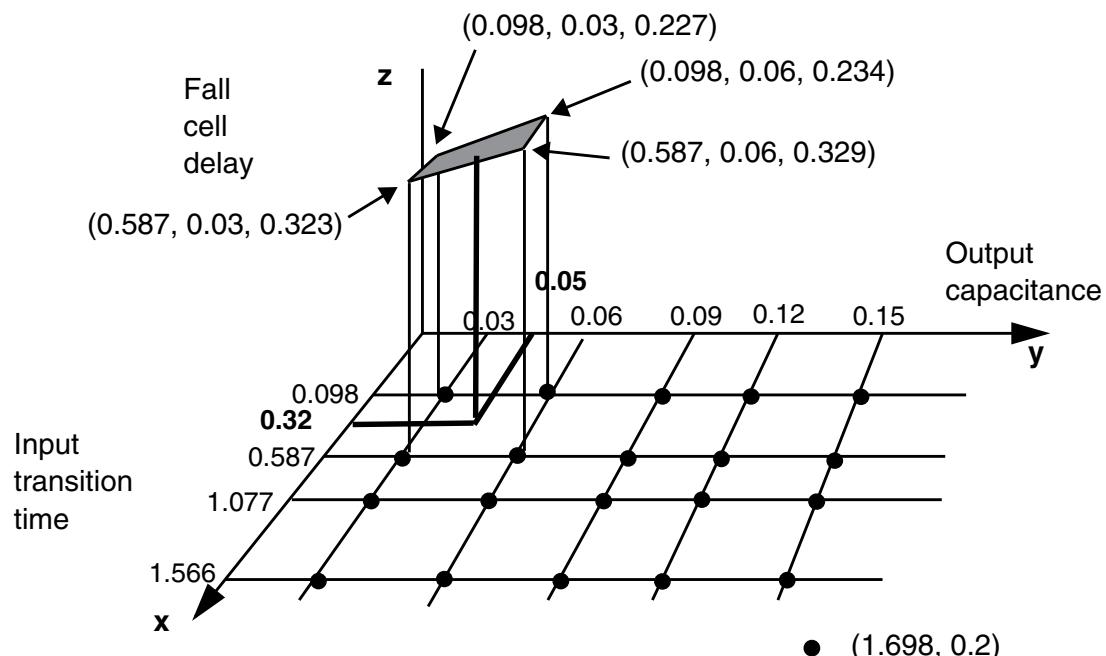
Input transition time is determined by evaluation of the transition delay at the previous gate, U0. Because the timing sense in U1 is negative unate, use the rise transition table for U0 to determine U1's input transition time. If multiple timing arcs are present at gate U0, the maximum of the rise transition 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, however, 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 3-7](#). The corresponding z values are also shown in [Figure 3-7](#).

*Figure 3-7 Result of Delay Calculation*



You determine the approximation for the surface described by the three coordinates ( $x, y, z$ ) in [Figure 3-7](#) 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 3-3](#)).

$$Z = A + B_x + C_y + D_{xy}$$

You can derive the coefficients A, B, C, and D with common mathematical methods, such as Gaussian elimination, as shown in [Example 3-3](#).

*Example 3-3 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 3-1](#) shows the coefficient values for the fall cell delay.

*Table 3-1 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 3-7](#) represent points you defined in the library source file for the 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 3-7](#)—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 3-3](#).

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  $D_{\text{total}}$  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.

$\Delta_{\text{process}}$ ,  $\Delta_{\text{temp}}$ , and  $\Delta_{\text{voltage}}$

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,

$$\Delta_{\text{process}} = \text{process} - \text{nom\_process}$$

$K_{\text{process}}$ ,  $K_{\text{temp}}$ , and  $K_{\text{voltage}}$

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. Library Compiler assigns a 0 (zero) as the default value for each scaling 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 “[Defining Operating Conditions](#)” on page 12-15 and the *Library Compiler Reference Manual: Technology and Symbol Libraries*.

The k-factors specific to the CMOS nonlinear delay model apply to the table delay values, not to  $D_{\text{transition}}$ ,  $D_{\text{propagation}}$ , or  $D_{\text{cell}}$ . This presents a subtle modeling consideration when you develop a library.

In the nonlinear delay model (as in other Design Compiler models), delay parameters are scaled before delay calculation is performed. 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  $D_{total}$  by using the nonlinear delay model, 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.

---

## Scalable Polynomial Delay Model

Scalable polynomials provide a smaller and faster alternative to nonlinear lookup tables.

To create a scalable polynomial delay model, you need to be familiar with the following concepts and tasks:

- Polynomial representation
- Model description
- Scalable polynomial delay calculation
- Using the conversion feature

The term *scalable* indicates that the form and the order of the polynomials are determined by, or scaled to, the given data. Given a predefined accuracy, most delay data can be modeled with polynomials.

The advantages of using a predefined set of polynomials, instead of an arbitrary equation that attempts to fit all cases, are efficiency and controllability. Also, by allowing for the inclusion of temperature and voltage (besides slope and load) as additional dimensions for timing arcs, you can develop a single library instead of multiple libraries (that is, one for each operating condition).

Note:

The scalable polynomial model does not alter the process of library construction. You still need to run a circuit-level simulator such as SPICE to collect timing data, and you need to use curve-fitting methods to convert the characterization data into computationally efficient polynomial equations with sufficient user-defined accuracy.

---

## Polynomial Representation

Polynomials play a key role in numerical computations. The fundamental theory is Taylor expansions, where an analytical function can be expressed as a finite series of polynomials.

The complete decomposed polynomial form represents the scalable polynomial delay model syntax. The following example shows two variable functions, but it is easy to extend to the case of more variables. A two-variable polynomial function  $D(x,y)$  can be written as  $D(x,y)=P_m(x)Q_n(y)$ , where  $x$  and  $y$  are variables and  $P_m$  and  $Q_n$  are the  $m$ th- and  $n$ th-order polynomials, respectively. There are  $(m+1)(n+1)$  coefficients for any given  $m$  or  $n$ , as the following equation illustrates.

$$\begin{aligned} P_1 Q_2 &= (a_0 + a_1 x_1) (b_0 + b_1 x_2 + b_2 x_2^2) = \\ A_{00} + A_{10}x_1 + A_{01}x_2 + A_{11}x_1x_2 + A_{02}x_2^2 + A_{12}x_1x_2^2 \end{aligned}$$

---

## Model Description

The scalable polynomial delay model syntax allows you to specify up to variable  $n$  polynomials. The dimensions are slew, load, related load, temperature, voltage, and voltage $i$ . Most cells reference a single voltage rail, specified by voltage. In the case of level shifter cells, both voltage and voltage1 are used for voltage rails. The model considers analytical parameters (physical parameters) that affect the delay calculation results to be variables.

For a large data set with abrupt changes, a single polynomial equation might not fit the entire operating region of interest. In this case, use the piecewise (adaptive domain) syntax to specify the breakpoints over the characterization data domains.

As with lookup tables, you describe a template specifying the polynomial form before specifying the values (in this case, polynomial orders and coefficients). Orders of coefficients can be included in the cell delay, propagation delay, and transition delay.

You specify polynomials in the expanded decomposed form. Assume  $m$ ,  $n$ ,  $p$ ,  $q$ ,  $r$ , and  $s$  as the orders of each of six dimensions ( $x_1$ — $x_6$ ). The number of coefficients is

$$(m+1)(n+1)(p+1)(q+1)(r+1)(s+1).$$

The coefficients are expressed as

$$\sum_{f=0}^s \sum_{e=0}^r \sum_{d=0}^q \sum_{c=0}^p \sum_{b=0}^n \sum_{a=0}^m A_{abcdef} x_1^a x_2^b x_3^c x_4^d x_5^e x_6^f$$

## Examples

**Example 3-4** shows the sequence of the coefficients in the case of a three-dimensional polynomial ( m=3, n=1, p=2 ):

*Example 3-4 Sequence of Coefficients in the Case of a Three-Dimensional Polynomial*

$$\begin{aligned} A_{000} + A_{100}x + A_{200}x^2 + A_{300}x^3 + A_{010}y + A_{110}xy + A_{210}x^2y + \\ A_{310}x^3y + A_{001}z + A_{101}xz + A_{201}x^2z + A_{301}x^3z + A_{011}yz + \\ A_{111}xyz + A_{211}x^2yz + A_{311}x^3yz + A_{002}z^2 + A_{102}xz^2 + \\ A_{202}x^2z^2 + A_{302}x^3z^2 + A_{012}yz^2 + A_{112}xyz^2 + A_{212}x^2yz^2 + A_{312}x^3yz^2 \end{aligned}$$

**Example 3-5** shows how the sequence of the coefficients is represented in the timing group:

*Example 3-5 Sequence of Coefficients in the timing Group*

$$\begin{aligned} A_{000}, A_{100}, A_{200}, A_{300}, A_{010}, A_{110}, A_{210}, A_{310}, A_{001}z, A_{101}, A_{201}, \\ A_{301}, A_{011}, A_{111}, A_{211}, A_{311}, A_{002}, A_{102}, A_{202}, A_{302}, A_{012}, \\ A_{112}, A_{212}, A_{312} \end{aligned}$$

With the above approach, memory usage is significantly improved, because

- Relatively few polynomial coefficients are stored, compared to table values
- Only one library is required instead of one library per operating condition

The following features are also supported in the delay model:

- If it is not possible to curve-fit a complex table with sufficient accuracy and at a reasonable computation cost, you can still retain the lookup table in the library.

- You can specify piecewise polynomials and their range by using the piecewise polynomial template. To ensure continuity and completeness, specify piecewise polynomials to the entire specified variable domain.
- To ensure that the polynomial is used within the intended range, specify its range by using the `variable_n_range` attribute. This helps avoid explosions of the calculated slew or load due to the higher-order terms in the polynomial equation. The scalable polynomial delay model linearly extrapolates outside the range of the equation, using the closest boundary values.

---

## Scalable Polynomial Calculation

Library Compiler supports the automatic conversion of a nonlinear delay model to a polynomial delay model.

A curve-fitting algorithm dictates the error distribution pattern between the data set and the fitted equation. There are two curve-fitting methods you can use to find the polynomial coefficients for a given set of characterized data points. The two curve-fitting methods are the least square error (lse) and the least square relative error (lsre).

As with nonlinear delay models, you can determine the number and values of the characterization points by selecting the polynomial curve-fitting algorithm that matches the accuracy requirements of the given technology.

The conversion feature implemented in Library Compiler incorporates the scalable polynomial delay modeling techniques and reports the results of the two polynomial curve-fitting techniques (lse and lsre). The conversion feature, however, is not intended for use as a complete library characterization tool.

**Note:**

Currently, the conversion feature generates only polynomials depending on the variables supported by the nonlinear delay model tables. These variables are the input transition time and the output capacitance load. Temperature and voltage effects are not currently supported by the conversion utility. Multiple scalable polynomial delay model libraries can be created from existing nonlinear delay-model-based libraries for each operating point of interest. However, temperature and voltage parameters can be modeled in the scalable polynomial delay model format by use of vendor-selected external polynomial curve fitters.

---

## Using the Conversion Feature

To convert a nonlinear delay model to a polynomial delay model, use the following procedure:

1. Enable the conversion.

```
lc_shell/dc_shell > libgen_spdm_extract_enable = true
```

2. Choose the conversion algorithm (lse or lsre).

```
lc_shell/dc_shell > libgen_spdm_extract_algorithm = lsre
```

3. Control the error margin between the two models. (This example specifies the sigma setting as 5 percent).

```
lc_shell/dc_shell > libgen_spdm_extract_sigma = 5
```

4. Control the number of coefficients relative to the table size.

```
lc_shell/dc_shell > libgen_spdm_extract_ratio = 0.5
```

5. Convert the lookup table model into a polynomial delay model during library compilation. Use either the source .lib or the compiled .db as the input to the conversion feature. Use the compiled .db with polynomials instead of tables as the output.

```
lc_shell/dc_shell > read_lib non_linear_delay_model_library.lib
```

or

```
lc_shell/dc_shell > read non_linear_delay_model_library.db  
...  
lc_shell/dc_shell > write_lib libr_name -o poly_delay_model_lib.db
```

---

## CMOS Piecewise Linear Delay Model

Design Compiler uses the timing parameters and environment attributes described in a technology library to calculate the timing delays for designs. To understand the effects of the timing parameters and the library environments on the timing delay calculations, you need to know about the concepts discussed in the following sections of this chapter:

- Total Delay Equation
- Total Delay Scaling
- Piecewise Linear Model
- Intrinsic Delay
- Slope Delay
- Transition Delay
- Connect Delay
- Interconnect Delay

For additional information about how Design Compiler uses timing delay information, see the Design Compiler documentation.

---

## Total Delay Equation

Delay analysis involves calculating the delay between the input pin of a gate and the input pin of the next gate, which includes the connect delay from the driving pins to the load pins.

This delay modeling divides the total delay in a network into four physical components whose sum is the total delay through a circuit element:

$$D_{\text{total}} = D_I + D_S + D_C + D_T$$

Following are the components of the delay model:

$D_I$

Intrinsic delay inherent in the gate and independent of any particular instantiation.

$D_S$

Slope delay caused by the ramp time of the input signal.

$D_C$

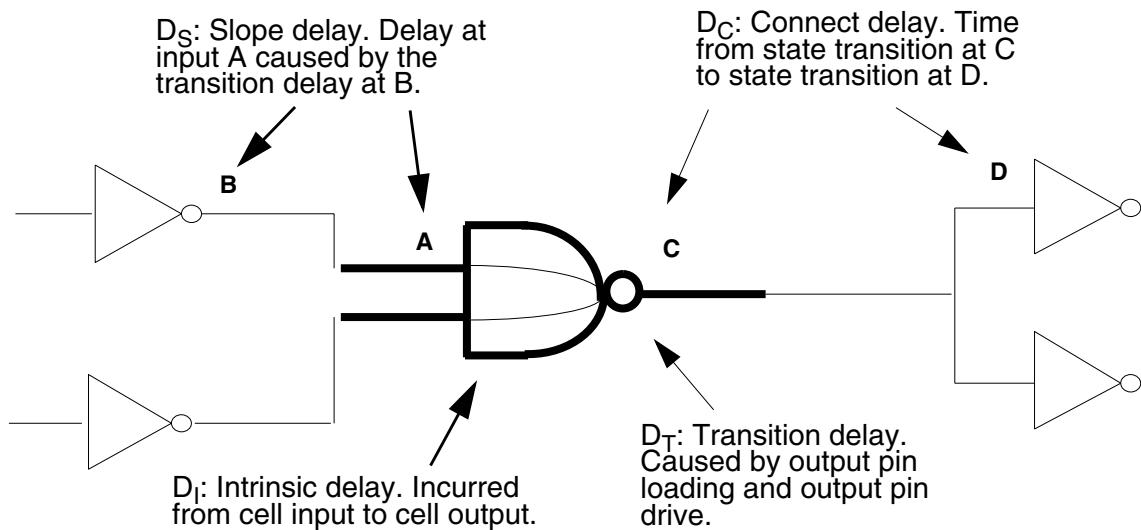
Connect media delay to an input pin (wire delay).

$D_T$

Transition delay caused by loading of the output pin.

Figure 3-8 shows the components of the total delay.

**Figure 3-8 Delay Equation Components for CMOS Piecewise Linear Delay Model**



Total delay is a monotonically increasing function of the four components. Each component monotonically increases its primary parameters.

All timing parameters are defined in the same unit in a library description.

You do not need to designate the unit you use to perform the delay calculations; however, you must use a consistent unit throughout your library to ensure accurate results.

You can use several different delay models with the same analysis method. For example, setting the slope and interconnect components to zero effect yields a simple rise or fall model. The rise and fall delay parameters are separated in the formulated equation.

All four components of the delay equation are affected in different degrees by variations in manufacturing process, operating temperature, and supply voltage. Consequently, separate scaling factors are applied to individual components of the delay equations.

---

## Total Delay Scaling

When calculating the total delay, Design Compiler scales each parameter of  $D_{\text{total}}$  individually. Each component of the total delay has its own global parameters to model the effects on the nominal case of variations in process, temperature, and voltage. The actual delays used for analysis are scaled versions of the four components of the general delay equation:

$$D_{\text{total}} = D_I + D_S + D_C + D_T$$

The equation for the scaled versions of the individual components' total delay is

$$\left( D_{\text{scaled}} = D(1 + \Delta_{\text{process}} \times K_{\text{process}})(1 + \Delta_{\text{temp}} \times K_{\text{temp}})(1 + \Delta_{\text{voltage}} \times K_{\text{voltage}}) \right)$$

D

A component of the total delay equation:  $D_I$ ,  $D_S$ ,  $D_C$ , or  $D_T$ .

$\Delta_{\text{process}}$ ,  $\Delta_{\text{temp}}$ , and  $\Delta_{\text{voltage}}$

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. Here is one illustration:

$$\Delta_{\text{process}} = \text{process} - \text{nom\_process}$$

$K_{process}$ ,  $K_{temp}$ , and  $K_{voltage}$

Represent the scaling factors of the individual components of each member of the total delay equation.

The total scaled delay equation takes this form:

$$D_{total(scaled)} = D_{I(scaled)} + D_{S(scaled)} + D_{C(scaled)} + D_{T(scaled)}$$

Scaling factors for process, temperature, and voltage are called k-factors (scaling factors that begin with  $k_$ ) and are defined at the library level in the technology library.

Nominal operating conditions and `operating_conditions` groups are also defined at the library level. For a complete description of these statements, see “[Defining Operating Conditions on page 12-15](#) and the *Library Compiler Reference Manual: Technology and Symbol Libraries*.

For an illustration of total delay scaling, see [Example 3-6 on page 3-52](#).

---

## Piecewise Linear Model

When Design Compiler calculates timing delays in a generic delay model, it overlooks the effect different wire lengths have on the net. The equations used by a piecewise linear model to calculate timing delays take into consideration delay effects the actual length of wire has on various components of the total delay equation.

To use a piecewise linear model, first assign the `piece_define` attribute in the `library` group. This attribute defines the ranges of capacitance values or wire lengths in terms of different attribute values.

The syntax of the `piece_define` attribute is

```
piece_define ("length0 [length1 length2 ...]" ) ;  
piece_define ("cap0 [cap1 cap2 ...]" ) ;
```

Each *length* value is a positive floating-point number defining the lower limit of the respective range. A piece of wire whose length is between *length0* and *length1* is defined as piece 0, a length between *length1* and *length2* is piece 1, and so on.

For example, consider the following `piece_define` specification: In this example, a piece of wire of length 5 is referred to as piece 0, a piece of wire of length 10 is piece 1, length 20 is piece 2, length 30 or more is piece 3.

```
piece_define ( "0 10 20 30" ) ;
```

If the capacitance form is used, each capacitance is a positive floating-point number defining the lower limit of the respective range. A piece of wire whose capacitance is between *cap0* and *cap1* is defined as piece 0, a capacitance between *cap1* and *cap2* is piece 1, and so on.

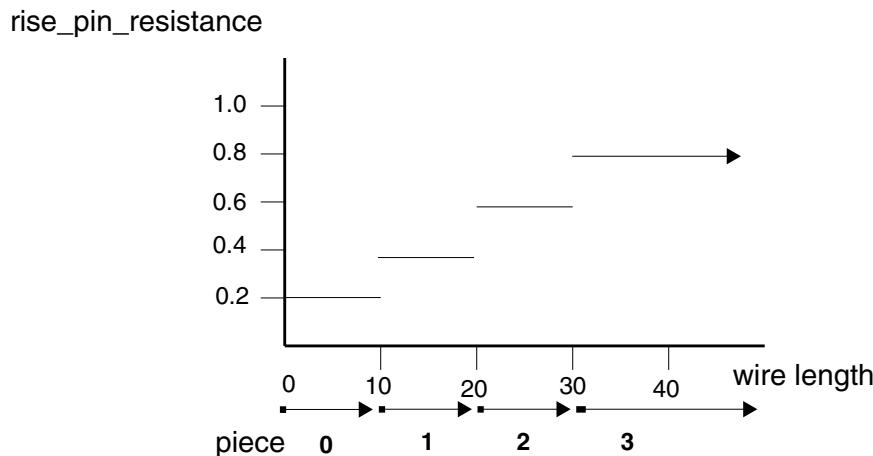
In this model, you must include all ranges of wire length or capacitance you want to associate with unique attribute values.

Some components of the transition delay equation require that you define the range (piece) associated with the attribute value. Define these components as constant (linear) for any single range. The following example defines a unique `rise_pin_resistance` value for each of the ranges in the previous `piece_define` statement.

```
rise_pin_resistance ( 0, 0.2 ) ; /* piece 0 */  
rise_pin_resistance ( 1, 0.4 ) ; /* piece 1 */  
rise_pin_resistance ( 2, 0.6 ) ; /* piece 2 */  
rise_pin_resistance ( 3, 0.8 ) ; /* piece 3 */
```

Figure 3-9 shows a graph of the `rise_pin_resistance` value as previously defined.

Figure 3-9 Graph of `rise_pin_resistance` Values



You do not have to designate attributes for each piece defined in the `piece_define` statement. For example, if you designate a `rise_pin_resistance` attribute for only pieces 0 and 2, Library Compiler interpolates the value of `rise_pin_resistance` for the other pieces defined by the `piece_define` statement.

---

## Intrinsic Delay

The intrinsic delay of an element ( $D_I$ ) is the portion of the total delay that is independent of the element's usage. This is the fixed (or zero load) delay from the input pin to the output pin of an element.

Constant values for the intrinsic delay are stored in the timing group of the load pin as floating-point numbers in the `intrinsic_rise` and `intrinsic_fall` attributes.

The total intrinsic delay is calculated by scaling of these constant values by their corresponding k-factors (see “[Total Delay Scaling](#)” on page 3-40).

---

## Slope Delay

The slope delay of an element ( $D_S$ ) is the incremental time delay caused by slowly changing input signals. Normally, library cells are characterized with a nominal ramp time (1.5 ns is common) to determine the delay from the input pin to the output pin.

In some technologies, this delay is a strong function of the ramp time. In other technologies, this delay does not vary—even over a wide range of ramp values.  $D_S$  is included in the delay equation to allow more-accurate modeling of technologies that are sensitive to input ramp time. See “[Total Delay Scaling](#)” on page 3-40.

$D_S$  is calculated with the transition delay at the previous output pin, plus a slope-sensitivity factor, as shown here:

$$D_S = S_S \times D_{T(\text{prevstage})}$$

This equation calculates the rise and fall delays. The components of the equation are described next. Where applicable, use the `rise_` parameter to calculate the rise delay and the `fall_` parameter to calculate the fall delay.

$S_S$

Slope sensitivity factor. This 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. The attributes are defined in the `timing` group of the driving pin as

slope\_rise  
slope\_fall

$D_T(\text{prevstage})$

The transition delay calculated at the previous output pin.

The  $D_T$  (transition delay) component of this equation must be calculated according to the piecewise linear delay model equation of  $D_T$ .

$D_S$

The transition delay is calculated at the previous stage of logic. Therefore, the calculation of  $D_S$  enforces a global order on local analysis. If you assume that all delay calculations are performed with a minimum achievable ramp time (possibly zero), omitting the slope in timing analysis prevents overestimation of the total delay.

The total slope delay is calculated by scaling of the constant values by their corresponding k-factors. See “[Total Delay Scaling](#)” on page 3-40.

---

## Transition Delay

The transition delay represents the time it takes the output pin to change state. The transition time of the output pin on a net is a function of both the drive resistance and the total load capacitance on the output pin. The equation for the transition delay component is

$$D_T = R_{\text{tdrive}_i} \left( C_{\text{pins}} + C_{\text{wire}} \right) + Y_{\text{adj}_i}$$

This equation calculates the rise and fall delays. The components of this equation are described below. Where applicable, use the `rise_` parameter to calculate the rise delay and the `fall_` parameter to calculate the fall delay.

$R_{t\text{drive}_i}$

The drive (pin) resistance. This complex attribute defines a value for each range (piece) of wire length and is defined in the `timing` group of the driving pin as

```
rise_pin_resistance  
fall_pin_resistance
```

Constant resistance values used in the generic delay model (`rise_resistance` and `fall_resistance`) are not valid in the piecewise model.

$C_{\text{pins}}$

The sum of the pin capacitance values for all loads on the net in the `pin` groups of each pin on the net. Pin capacitance values appear to be measured as capacitance.

$C_{\text{wire}}$

The estimated wire capacitance for the net attached to the head of the timing arc. The wire length is computed with the actual number of fanout pins on the net and the `fanout_length` definitions in the `wire_load` group. The estimated value is scaled by the capacitance factor, which is defined in the `wire_load` group as `capacitance`. If no `wire_load` group is designated at runtime, the value of  $C_{\text{wire}}$  is 0.

$Y_{adj_i}$

This parameter corresponds to the Y-intercept of a slope or intercept timing equation. The complex attributes designate a value for each range (piece) of wire length. The attributes are defined in the `timing` group of the driving pin as

```
rise_delay_intercept  
fall_delay_intercept
```

The total transition delay is calculated by scaling of the constant values by their corresponding k-factors. See “[Total Delay Scaling](#)” on [page 3-40](#).

---

## Connect Delay

The connect delay of an element ( $D_C$ ) is the time the voltage at an input pin takes to charge after the driving output pin has made a transition. This delay is also known as time-of-flight delay, which is the time it takes a waveform to travel along a wire.

Connect delay equations calculate the rise and fall delays. The components of these equations are described next.

$R_{wire}$

Estimated wire resistance on the net, determined by the `wire_load` model. Wire length is computed with a global estimation function whose parameter is the number of fanout pins on the net being estimated. The estimated value is scaled by the resistance factor, which is defined in the `wire_load` group as resistance.

### $C_{\text{wire}}$

Estimated wire capacitance on the net attached to the head of the timing arc. Wire length is computed with the actual number of fanout pins on the net being estimated and the `fanout_length` definitions in the `wire_load` group. The estimated value is scaled by the capacitance factor defined in the `wire_load` group as capacitance. If no `wire_load` group is defined at runtime, the value of  $C_{\text{wire}}$  is 0.

### $C_{\text{pin}}$

Capacitance values for the load pin, defined in the `pin` group of the load pin as capacitance.

Library Compiler and Design Compiler support three cases for an estimated interconnect topology. These three cases represent the three possible values of the `operating_conditions` group attribute `tree_type` (see “[Defining Operating Conditions](#)” on [page 12-15](#)).

#### `best_case_tree`

The best case models the load pin as physically adjacent to the driver. All the wire capacitance is incurred, but none of the wire resistance must be overcome. The following equation calculates the best-case connect delay. Because  $R_{\text{wire}}$  is always 0 (zero) in this case, the resulting  $D_C$  is always 0.

$$D_{C_{\text{best}}} = R_{\text{wire}}(C_{\text{wire}} + C_{\text{pin}}) = 0$$

#### `worst_case_tree`

The worst case models the load pin at the extreme end of the wire. Each load pin incurs both the full wire capacitance and the full wire resistance. The following equation calculates the worst-case connect delay:

$$D_{C_{\text{worst}}} = R_{\text{wire}} \left( C_{\text{wire}} + \sum_{\text{pins}} C_{\text{pin}} \right)$$

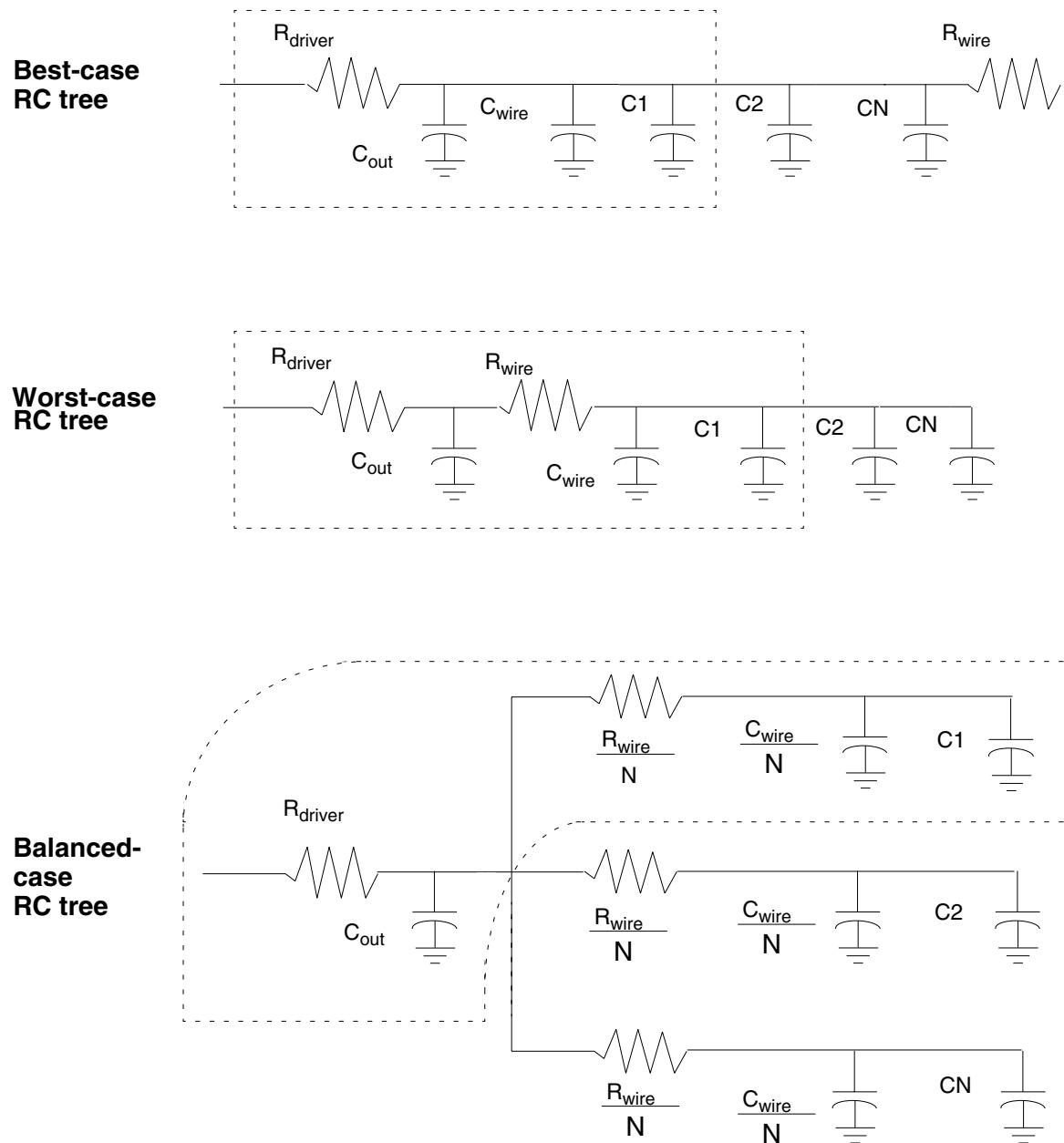
### balanced\_tree

The balanced case models all the load pins on separate, equal branches of the interconnect wire. Each load pin incurs an equal portion of the wire capacitance and wire resistance. The following equation calculates the balanced-case connect delay:

$$D_{C_{\text{balanced}}} = \frac{R_{\text{wire}}}{N} \left( \frac{C_{\text{wire}}}{N} + C_{\text{pin}} \right)$$

[Figure 3-10](#) shows the electrical relationships of the topologies. Dotted lines surround the parameters that are included in the connect delay calculation for one load pin.

*Figure 3-10 Resistance Capacitance Interconnect Tree Topologies for Fanout of  $N$*



See the following three equations for the expanded  $D_C$  equation for each tree topology.

Best-case  $D_C = 0.0$

Worst-case  $D_C =$

$$\begin{aligned} & \text{wire\_resistance} (1 + \Delta_{\text{voltage}} k_{\text{volt\_wire\_res}}) \\ & (1 + \Delta_{\text{temp}} k_{\text{temp\_wire\_res}})(1 + \Delta_{\text{process}} k_{\text{process\_wire\_res}}) \\ & [ (C_{\text{wire}})(1 + \Delta_{\text{voltage}} k_{\text{volt\_wire\_cap}}) \\ & (1 + \Delta_{\text{temp}} k_{\text{temp\_wire\_cap}})(1 + \Delta_{\text{process}} k_{\text{process\_wire\_cap}}) \\ & + \sum_{\text{pins}} \{ (C_{\text{pin}})(1 + \Delta_{\text{voltage}} k_{\text{volt\_pin\_cap}}) \\ & (1 + \Delta_{\text{temp}} k_{\text{temp\_pin\_cap}})(1 + \Delta_{\text{process}} k_{\text{process\_pin\_cap}}) \} ] \end{aligned}$$

Balanced-case  $D_C =$

$$\begin{aligned} & \text{wire\_resistance}/N (1 + \Delta_{\text{voltage}} k_{\text{volt\_wire\_res}}) \\ & (1 + \Delta_{\text{temp}} k_{\text{temp\_wire\_res}})(1 + \Delta_{\text{process}} k_{\text{process\_wire\_res}}) \\ & [ (C_{\text{wire}}/N)(1 + \Delta_{\text{voltage}} k_{\text{volt\_wire\_cap}}) \\ & (1 + \Delta_{\text{temp}} k_{\text{temp\_wire\_cap}})(1 + \Delta_{\text{process}} k_{\text{process\_wire\_cap}}) \\ & + (C_{\text{pin}})(1 + \Delta_{\text{voltage}} k_{\text{volt\_pin\_cap}}) \\ & (1 + \Delta_{\text{temp}} k_{\text{temp\_pin\_cap}})(1 + \Delta_{\text{process}} k_{\text{process\_pin\_cap}}) ] \end{aligned}$$

$N$  is the number of load pins on the net.

The total connect delay is calculated by scaling of the constant values by their corresponding k-factors. See “[Total Delay Scaling](#)” on page 3-40.

---

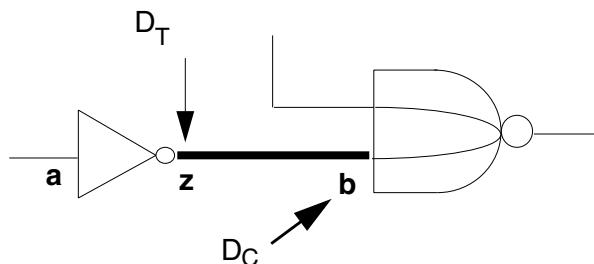
## Interconnect Delay

Interconnect delay is defined as the delay caused by the connect delay and fanout.

$$D_{\text{interconnect}} = D_T + D_C$$

Interconnect delay can be calculated in several ways. If you take into consideration the path delay computation performed by Design Compiler, the most efficient modeling technique is to attribute the transition delay to the output of the driving gate and the connect delay to the input of the driven gate (see [Figure 3-11](#)).

*Figure 3-11 Interconnect Delay Diagram*



Include the `capacitance` attribute in the `pin` group of the input pin. Give zero capacitance to the `pin` group of the output pin. Resistance is attributed entirely to the output pin. [Example 3-6](#) shows how the interconnect delay of [Figure 3-11](#) is modeled.

*Example 3-6 Modeling Interconnect Delay*

```
library (example) {
    date : "November 12, 2002" ;
    revision : 2.2 ;
    piece_define ( "0 10" ) ;
    cell (inv1) {
        area : 1 ;
        pin (a) {
            direction : input ;
            capacitance : 0.0 ; // Input pin
        }
        pin (y) {
            direction : output ;
            resistance : 1.0 ; // Output pin
        }
    }
}
```

```

        capacitance : 1 ;
    }
    pin (z) {
        direction : output ;
        capacitance : 0 ;
        timing () {
            related_pin : "a" ;
            intrinsic_rise : 1.5 ;
            intrinsic_fall : 1.2 ;
            rise_pin_resistance (0, 1.2) ;
            fall_pin_resistance (0, 0.8) ;
            rise_delay_intercept (0, 0.89) ;
            fall_delay_intercept (0, 0.71) ;
            rise_pin_resistance (1, 1.4) ;
            fall_pin_resistance (1, 0.98) ;
            rise_delay_intercept (1, 1.02) ;
            fall_delay_intercept (1, 0.89) ;
        }
    }
}
cell (NAND1) {
    area : 1.5 ;
    pin (b) {
        direction : input ;
        capacitance : 1.5 ;
    }
    ...
}
```

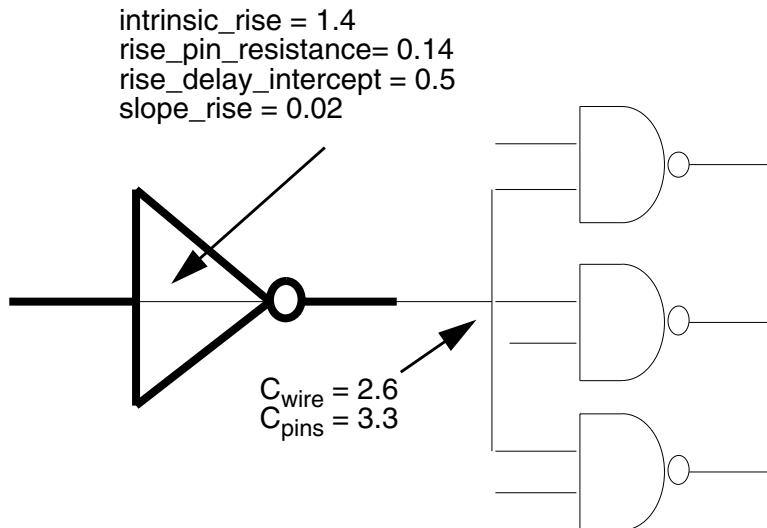
Total interconnect delay is defined as the sum of the scaled versions of  $D_T$  and  $D_C$ . See “[Total Delay Scaling](#)” on page 3-40.

---

## Delay Calculation Example

Figure 3-12 shows the rise delay parameters for an inverter.

Figure 3-12 Rise Delay Diagram



The input pin is driven by a falling signal with a transition delay ( $D_T$ ) of 1.2 ns. This inverter fans out to three NAND gates, each with an input pin capacitance of 1.1. The inverter has an intrinsic rise delay of 1.4 ns and an output rise resistance of 0.14.

The computation in [Example 3-7](#) assumes that the output pin capacitance of the inverter is not modeled as a separate attribute but is included as part of the intrinsic rise value. Assuming a best-case RC-interconnect tree type and an estimated interconnect wire capacitance of 2.6, the rise delay is 2.75 ns.

[Example 3-7](#) Rise Delay Computation

```
intrinsic_rise = 1.4
rise_pin_resistance = 0.14
rise_delay_intercept = 0.5
slope_rise = 0.02
```

```

Cpins = 3 * (1.1) = 3.3
Cwire = 2.6
    DT(fall_previous_stage) = 1.2
DC = 0.0          /* for a best-case RC tree */
Rise Delay = Intrinsic + Slope + Transition + Connect
= 1.4 + (1.2 * 0.02) + [(0.14)(3.3 + 2.6) + 0.5] + 0.0
= 1.4 + 0.024      + 1.326           + 0.0
= 2.75

```

[Figure 3-8 on page 3-39](#) provides details about the components of the total delay equation.

Examining the following equation reveals a subtle modeling point to consider when you develop a library. In many technology modeling schemes, the overall delay ( $D_{Total}$ )—rather than the individual components ( $D_I$ ,  $D_S$ ,  $D_T$ , and  $D_C$ )—is scaled. This effect is achieved by use of only nonzero k-factors for intrinsic delay, pin resistance, wire resistance, and slope-sensitivity factors. With the same process, temperature and voltage variations for the respective components of the delay equation produce the same result as overall scaling.

Rise delay =

$$\begin{aligned}
D_I & \frac{\text{intrinsic\_rise} (1 + \Delta_{\text{voltage}} \text{k\_volt\_intrinsic\_rise})}{(1 + \Delta_{\text{temp}} \text{k\_temp\_intrinsic\_rise})(1 + \Delta_{\text{process}} \text{k\_process\_intrinsic\_rise})} \\
& + (D_T \text{prevstage})(\text{slope\_rise})(1 + \Delta_{\text{voltage}} \text{k\_volt\_slope\_rise}) \\
D_S & \quad (1 + \Delta_{\text{temp}} \text{k\_temp\_slope\_rise})(1 + \Delta_{\text{process}} \text{k\_process\_slope\_rise})
\end{aligned}$$


---

$$\begin{aligned}
& + \text{rise\_pin\_resistance} (1 + \Delta_{\text{voltage}} k_{\text{volt\_rise\_pin\_resistance}}) \\
& \quad (1 + \Delta_{\text{temp}} k_{\text{temp\_rise\_pin\_resistance}})(1 + \Delta_{\text{process}} k_{\text{process\_rise\_pin\_resistance}}) \\
D_T & \quad [ (C_{\text{wire}})(1 + \Delta_{\text{voltage}} k_{\text{volt\_wire\_cap}}) \\
& \quad (1 + \Delta_{\text{temp}} k_{\text{temp\_wire\_cap}})(1 + \Delta_{\text{process}} k_{\text{process\_wire\_cap}}) \\
& \quad + \sum_{\text{pins}} \{(C_{\text{pin}})(1 + D_{\text{voltage}} k_{\text{volt\_pin\_cap}}) \\
& \quad (1 + \Delta_{\text{temp}} k_{\text{temp\_pin\_cap}})(1 + \Delta_{\text{process}} k_{\text{process\_pin\_cap}})\} ] \\
& + \text{rise\_delay\_intercept} (1 + \Delta_{\text{voltage}} k_{\text{volt\_rise\_delay\_intercept}}) \\
& \quad (1 + \Delta_{\text{temp}} k_{\text{temp\_rise\_delay\_intercept}})(1 + \Delta_{\text{process}} k_{\text{process\_rise\_delay\_intercept}}) \\
& + \text{wire\_resistance} (1 + D_{\text{voltage}} k_{\text{volt\_wire\_res}}) \\
& \quad (1 + \Delta_{\text{temp}} k_{\text{temp\_wire\_res}})(1 + \Delta_{\text{process}} k_{\text{process\_wire\_res}}) \\
D_C & \quad [ (C_{\text{wire}})(1 + \Delta_{\text{voltage}} k_{\text{volt\_wire\_cap}})(1 + \Delta_{\text{temp}} k_{\text{temp\_wire\_cap}}) \\
& \quad (1 + \Delta_{\text{process}} k_{\text{process\_wire\_cap}}) \\
& \quad + \sum_{\text{pins}} \{(C_{\text{pin}})(1 + \Delta_{\text{voltage}} k_{\text{volt\_pin\_cap}}) \\
& \quad (1 + \Delta_{\text{temp}} k_{\text{temp\_pin\_cap}})(1 + \Delta_{\text{process}} k_{\text{process\_pin\_cap}})\} ]
\end{aligned}$$

## Delay Calculation Module (DCM) Delay Model

The Delay Calculation Module (DCM) model is an IEEE STD 1481.1 delay model.

Note:

See IEEE STD 1481.1 for details about setting up and using the DCM model.

If you select DCM, Library Compiler ignores all the timing-related attributes and groups except for `cell_degradation` and those attributes that identify timing and power, design rules, wire load, and pad.

If the `cell_degradation` attribute is selected for your library, its corresponding timing and `related_pin` information is also recognized by Library Compiler.

Selecting the DCM model has the following consequences:

- Because it ignores timing-related attributes and groups, Library Compiler cannot run any time-related consistency checks on your library.
- The application tools supporting DCM get the timing-related information from the DCM files generated by the Delay Calculation Language (DCL) compiler program and get nontiming information from the .db files generated by Library Compiler.

Note:

You can use both DCM and non-DCM technology libraries in the same design.

- An application tool that does not support DCM can use the .db file generated by Library Compiler if the application tool does not need to access any timing-related information during runtime.

Note:

You can use the `report_lib -timing` or the `report_lib-timing_arc` command to verify which delay model you selected. If you selected DCM as the delay model, these commands do not produce any timing information.



# 4

## Timing Arcs

---

Timing is divided into two major areas: describing delay (the actual circuit timing) and specifying constraints (boundaries). This chapter discusses timing concepts and describes the `timing group` attributes for setting constraints and defining delay with the different delay models.

For information on describing delay, see these sections:

- Understanding Timing Arcs
- Modeling Method Alternatives
- Defining timing Groups
- 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

For information on describing constraints, see these sections:

- 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:

- Timing Arc Restrictions
- Examples of Libraries Using Delay Models
- Describing a Transparent Latch Clock Model
- Checking Timing Library Data
- Driver Waveform Support
- Sensitization Support

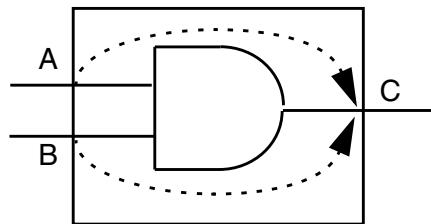
---

## 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 4-1 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 4-1 Timing Arcs



You must distinguish between combinational and sequential timing types, because they serve different purposes.

Design Compiler 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.

Design Compiler uses sequential timing arc information to determine rule-based design optimization constraints. See the *Design Compiler Reference Manual* for more information on 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 on describing combinational timing types, see “[timing Group Attributes](#)” on page 4-32.

---

## 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)

If you use Library Compiler to generate VHDL simulation models, a sequential timing arc can also be of this type:

- Skew (skew\_rising or skew\_falling)

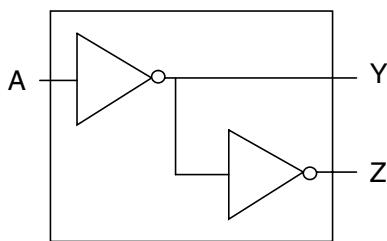
For information on describing sequential timing types, see “[timing Group Attributes](#)” on page 4-32.

---

## Modeling Method Alternatives

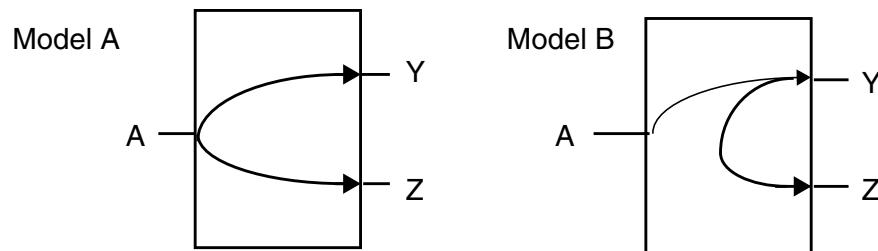
Timing information for combinational cells such as the one in [Figure 4-2](#) can be modeled in one of two ways, as [Figure 4-3](#) shows.

*Figure 4-2 Two-Inverter Cell*



In [Figure 4-3](#), 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 4-3 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 timing Groups

timing groups contain the information Design Compiler needs in order to model timing arcs and trace paths. timing groups define the timing arcs through a cell and the relationships between clock and data input signals.

timing groups describe

- 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

Note:

See the following section for a description of naming timing arcs by use of the timing group.

timing groups describe setup and hold information when the constraint information refers to an input-to-input pin pair.

A timing group is defined in a 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 4-1 on page 4-3](#).

Note:

Library Compiler allows you to define multiple timing arcs between two pins. However, other tools may have difficulty interpreting multiple timing arcs for pin pairs. The VHDL library generator supports one timing arc only. If you specify multiple arcs, the VHDL library generator uses the worst case.

---

## 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) identified with the `related_bus_pins` attribute.
- Between all the bits of a related-bus-equivalent group identified with the `related_bus_equivalent` attribute and an internal pin, and 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. You identify the multiple timing arcs on a name list entered with the timing group 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";  
        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.

Library Compiler 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;
        intrinsic_rise : 0.99;
        intrinsic_fall : 0.96;
        rise_resistance : 0.1458;
        fall_resistance : 0.0653;
        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 we assume that 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.

Library Compiler 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;
        intrinsic_rise : 0.99;
        intrinsic_fall : 0.96;
        rise_resistance : 0.1458;
        fall_resistance : 0.0653;
        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 we assume that 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.

Library Compiler 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 we assume that 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

Library Compiler 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 we assume that 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.

## Timing Arcs Between a Bus and Related Bus Pins

This section describes the timing arcs created when a timing group is within a bus group that has several bits that have to be matched with several related bus pins of a required width.

You identify the resulting multiple timing arcs by entering a name list with the timing group.

Library Compiler assumes that the first name in the name list is the arc going from the first pin of the related bus pins of the necessary width to the MSB in the bus group, the second name in the name list is the arc going from the second pin of the related bus pins to the second MSB in the bus group, and so on. See the following example:

## 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'";
        timing (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 timing arcs 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

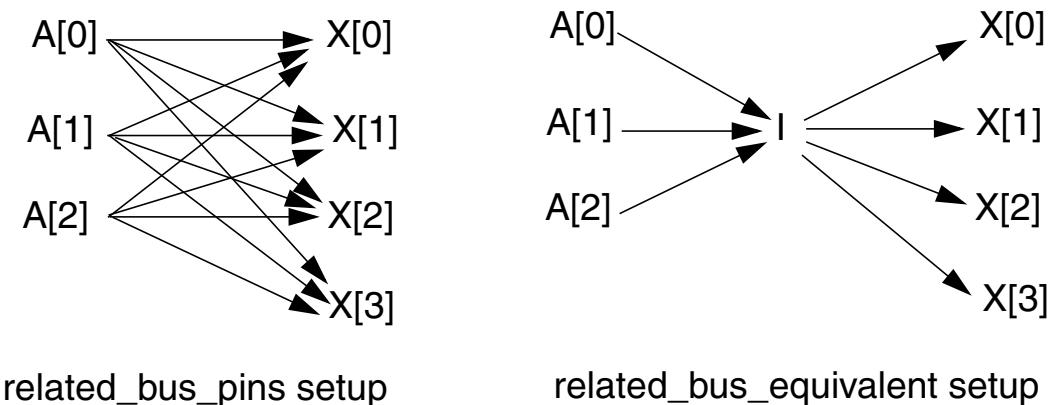
---

## Timing Arcs Between a Bus and a Related Bus Equivalent

You can generate an arc from each element in a starting bus to each element of an ending bus, such as when you create arcs from each related bus pin defined with the `related_bus_pins` attribute to each endpoint.

Instead of using this approach, you can use the `related_bus_equivalent` attribute to generate a single timing arc for all paths from points in a group through an internal pin (I) to given endpoints. [Figure 4-4](#) compares the setup created with the `related_bus_pins` attribute with a setup created with the `related_bus_equivalent` attribute.

*Figure 4-4 Comparing `related_bus_pins` Setup With `related_bus_equivalent` Setup*



This section describes the timing arcs created from all the bits of a related bus equivalent group, which you define with the `related_bus_equivalent` attribute, to an internal pin (`I`) and all the timing arcs created from the same internal pin to all the bits of the endpoint `bus` group.

You identify the resulting multiple timing arcs by entering a name list, using the `timing` group.

Library Compiler assumes that the first name in the name list is the arc going from the first bit (`A[0]`) of the related `bus` group to the internal pin (`I`), the second name in the name list is the arc going from the second bit (`A[1]`) to the internal pin (`I`), and so on in order until all the related `bus` group bits are used.

The next name on the name list is of the timing arc going from the internal pin (`I`) to the first bit (`X[0]`) in the endpoint `bus` group, the following name in the name list is of the arc going from the internal join pin (`I`) to the second bit (`X[1]`) of the `bus` group, and so on in order until all the bits in the `bus` group are used. See the following example.

**Note:**

The widths of bus A and bus X do not need to be identical.

## **Example**

```
bus (X) {...  
    bus_type : bus4;  
    direction : output;  
    capacitance : 1;  
    timing (A0_I, A1_I, A2_I, I_X0,I_X1, I_X2, I_X3,) {...  
        related_bus_equivalent : A;  
    }...  
}
```

The following is a list of the timing arcs and their labels:

From pin	To pin	Label
A[0]	I	A0_I
A[1]	I	A1_I
A[2]	I	A2_I
I	X[0]	I_X0
I	X[1]	I_X1
I	X[2]	I_X2
I	X[3]	I_X3

---

## Delay Models

Timing groups are defined by the timing group attributes. The delay model you use determines which 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 these delay models (see Chapter 3, “Delay Models,” for detailed information):

CMOS generic delay model

This is the standard delay model.

## CMOS nonlinear delay model

The nonlinear delay model 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.

## CMOS piecewise linear delay model

The equations this model uses to calculate timing delays consider the delay effect of different wire lengths.

## Scalable polynomial delay model

The scalable polynomial delay model is characterized by scalable polynomial equations that define the timing arcs. To describe delay and constraint arcs with this model,

- Use the `poly_template` group to set up a template of common polynomials to use in the `timing` group.
- Use the templates and the `timing` groups described in this chapter to specify equations.

Multiterm and multiorder equations can specify up to variable *n* variables.

## **delay\_model Attribute**

To specify the CMOS 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 should follow the `technology` attribute.

### **Example**

```
library (demo) {  
    delay_model : table_lookup ;  
}
```

## **Defining the CMOS Nonlinear Delay Model 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");  
}
```

Library Compiler 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 value `input_net_transition` from set 1, then you must assign `variable_2` with one of the values from set 2. [Table 4-1](#) lists the combinations of values you can assign to the different variables for the varying dimensional tables specifying timing delays.

*Table 4-1 Variable Values for Timing Delays*

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 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 value `constrained_pin_transition` from set 1, then you must assign `variable_2` with one of the values from set 2.

**Table 4-2** lists the combination of values you can assign to the different variables for the varying dimensional tables specifying load-dependent constraints.

*Table 4-2 Variable Values for Load-Dependent Constraint Tables*

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 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 4-68](#) 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 4-5 on page 4-72](#).
- 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 x 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 4-5 on page 4-72](#) shows a library that uses the CMOS nonlinear delay model to describe the delay.

## Defining the Scalable Polynomial Delay Model Template

Polynomial templates store common format information that equations can use.

### **poly\_template Group**

You use a `poly_template` group to specify the equation variables, the variable ranges, the voltage mapping, and the piecewise data. Assign each `poly_template` group a unique name, so that equations in the `timing` group can refer to it.

### **Syntax**

```
poly_template(name_id) {  
    variables(variable_i_enum, ..., variable_n_enum)  
    variable_i_range(float, float)  
    ...  
    variable_n_range(float, float)  
    mapping(voltage_enum, power_rail_id)  
    domain(domain_name_id) {  
        calc_mode : name_id ;  
        variables(variable_i_enum)..., variable_n_enum)  
        variable_i_range(float, float)  
        ...  
        variable_n_range(float, float)  
        mapping(voltage_enum, power_rail_id)  
    }  
}
```

### **poly\_template Variables**

The `poly_template` group that defines timing delays can have up to *n* variables (`variable_i`, ..., `variable_n`), which you specify in the `variables` complex attribute. The variables you specify represent the following in the equation:

- The input transition time of a constrained pin

- The output net length and capacitance
- The output loading of a related pin
- The default power supply voltage
- The voltage  $i$  for multivoltage cells
- The temperature
- User parameters ( parameter1...parameter5 )

The following list shows the valid values, divided into four sets, that you can assign to variables in an equation:

- Set 1:

`input_net_transistion`  
`constrained_pin_transition`

- Set 2:

`total_output_net_capacitance`  
`output_net_length`  
`output_net_wire_cap, output_net_pin_cap`  
`related_pin_transistion`

- 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`

- Set 4:

temperature  
voltage/*i*  
parameter/*n*

### **delay\_model Simple Attribute**

Use the `delay_model` attribute in the `library` group to specify the scalable polynomial delay model.

```
library (demo) {
    delay_model : polynomial ;
}
```

### **timing Group Attributes**

The delay model you use determines which set of attributes for delay model calculation you specify in a `timing` group. [Table 4-3](#) shows the available delay models and the supported `timing` group attributes for each. See [Chapter 3, “Delay Models,”](#) for detailed information about the delay models.

*Table 4-3 timing Group Attributes in the Delay Models*

Purpose	CMOS generic	CMOS piecewise linear	CMOS nonlinear/ scalable polynomial
To specify a default timing arc			default_timing
To identify timing arc startpoint	related_pin related_bus_pins	related_pin related_bus_pins	related_pin related_bus_pins
To describe logical effect of input pin on output pin	timing_sense	timing_sense	timing_sense
To identify an arc as combinational or sequential	timing_type	timing_type	timing_type

**Table 4-3 timing Group Attributes in the Delay Models (Continued)**

Purpose	CMOS generic	CMOS piecewise linear	CMOS nonlinear/ scalable polynomial
To describe intrinsic delay on an output pin	intrinsic_rise intrinsic_fall	intrinsic_rise intrinsic_fall	( <i>inherent</i> )
To specify transition delay. (Used with propagation delay in CMOS nonlinear delay model.)	rise_resistance fall_resistance	rise_pin_resistance rise_delay_intercept fall_pin_resistance fall_delay_intercept	rise_transition fall_transition
To specify propagation delay in total cell delay. (Used with transition delay in CMOS nonlinear delay model.)			rise_propagation fall_propagation
To specify cell delay independent of transition delay			cell_rise cell_fall
To specify retain delay within the delay arc			retaining_rise retaining_fall
To describe incremental delay added to slope of input waveform	slope_rise slope_fall	slope_rise slope_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;  
}
```

Design Compiler 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 said to be *unate* if a rising (or falling) change on a positive (or negative) unate input variable causes the output function variable to rise (or fall) or not change. For a non-unate 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.

Design Compiler 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.

Design Compiler uses the sequential timing arcs information to determine rule-based design optimization constraints. More information on optimization constraints is available in the Design Compiler documentation.

## Example

```
timing () {
    timing_type : rising_edge;
}
```

[Table 4-4](#) shows the valid values for the `timing_type` attribute:

*Table 4-4 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_fall	combinational	Use these timing type values to describe 0-to-1 or 1-to-0 timing arcs, respectively.
three_state_disable three_state_enable three_state_disable_rise three_state_disable_fall three_state_enable_rise three_state_enable_fall	combinational	Use these values to describe three-state output pins.
rising_edge falling_edge	sequential	Use these values to specify which edge of the input signal causes a transition on the output pin.
preset clear	sequential	Use these values to affect fall or rise arrival time on a timing arc's endpoint pin.

**Table 4-4 Valid Values for the *timing\_type* Attribute (Continued)**

Value	Cells where used	Function
hold_rising hold_falling	sequential	Use these values to designate the edge of the related pin for the hold check.
setup_rising setup_falling	sequential	Use these values to designate the edge of the related pin for the setup check on clocked elements.
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 (not for VITAL models).
removal_rising removal_falling	sequential	Use these values to designate the edge that defines the removal time.
minimum_period min_pulse_width	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 `intrinsic_rise` value is the maximum skew time between the reference pin rising and the parent pin rising. The `intrinsic_fall` 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 `intrinsic_rise` value is the maximum skew time between the reference pin falling and the parent pin rising. The `intrinsic_fall` 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 `intrinsic_rise` attribute. For active-high asynchronous control signals, define the removal time with the `intrinsic_fall` attribute.

### `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 `intrinsic_rise` attribute. For active-high asynchronous control signals, define the removal time with the `intrinsic_fall` attribute.

### `minimum_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.

### `minimum_period`

This value, together with the `minimum_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. Library Compiler 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. Library Compiler 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 standard CMOS and nonlinear CMOS delay models. 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);
```

Library Compiler 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 4-1 shows a mode instance description.

#### *Example 4-1 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 4-2 shows multiple mode descriptions.**

### *Example 4-2 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 `intrinsic_rise` statement.
3. Define the 1-to-Z propagation time with the `intrinsic_fall` statement.
4. Include the `timing_type : three_state_disable` statement.

### Example

```

timing () {
    related_pin : "OE" ;

```

```

    timing_type : three_state_disable ;
    intrinsic_rise : 1.0 ; /* 0 to Z time */
    intrinsic_fall : 1.0 ; /* 1 to Z time */
}

```

**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 4-36](#) for more information.

[Example 4-3](#) describes the timing model for an inverter with a three-state output and active-high enable in a CMOS generic library.

## 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 `intrinsic_rise` statement.
3. Define the Z-to-0 propagation time with the `intrinsic_fall` statement.
4. Include the `timing_type : three_state_enable` statement.

### Example

```

timing () {
    related_pin : "OE" ;
    timing_type : three_state_enable ;
    intrinsic_rise : 1.0 ; /* Z-to-1 time */
    intrinsic_fall : 1.0 ; /* Z-to-0 time */
}

```

```
}
```

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 4-36 for more information.

**Example 4-3** shows a three-state cell with both disable and enable timing arcs.

*Example 4-3 Three-State Cell With Disable and Enable Timing Arcs*

```
library(example){  
    date : "January 14, 2002";  
    revision : 2000.01;  
    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() {  
                intrinsic_rise : 0.39;  
                intrinsic_fall : 0.75;  
                rise_resistance : 0.15;  
                fall_resistance : 0.06;  
                related_pin : "A";  
            }  
            timing() {
```

```

        timing_type : three_state_enable;
        intrinsic_rise : 0.39;
        intrinsic_fall : 0.75;
        rise_resistance : 0.15;
        fall_resistance : 0.06;
        related_pin : "E";
    }
    timing() {
        timing_type : three_state_disable;
        intrinsic_rise : 0.25;
        intrinsic_fall : 0.35;
        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 4-37 for information about the `timing_type` attribute.

The following example shows the timing arc for the QN pin in a JK flip-flop in a CMOS library using a CMOS generic delay model.

### Example

```
pin(QN) {
    direction : output ;
    function : "IQN" ;
    timing() {
        related_pin : "CP" ;
        timing_type : rising_edge ;
        intrinsic_rise : 1.29 ;
        intrinsic_fall : 1.61 ;
        rise_resistance : 0.1723 ;
        fall_resistance : 0.0553 ;
    }
}
```

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. Depending on the delay model defined in the library, the rise time for a preset arc is defined by creation of a rise delay table or by setting of the `intrinsic_rise` value. The fall timing for a clear arc is defined similarly. Generally, fall timing for preset arcs and rise timing for clear arcs should not exist.

In some cases, however, 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 PrimeTime 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 should be defined only for a case similar to one of the two examples described above.

---

## 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.

[Example 4-14 on page 4-131](#) and [Example 4-15 on page 4-133](#) show how to specify preset arcs.

---

## 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.

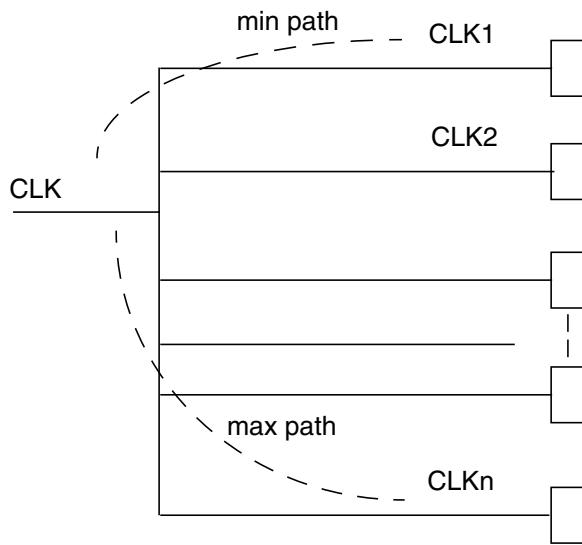
[Example 4-14 on page 4-131](#) and [Example 4-15 on page 4-133](#) show how to specify clear arcs.

---

## 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 4-5](#).

*Figure 4-5 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 either lookup tables or scalable polynomials to model the cell delays. Lookup tables are indexed only by the input transition time of the clock. Polynomials can include only the following variables in piecewise domains: `input_net_transition`, `voltage`, `voltage $i$` , and `temperature`.

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 or polynomials 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. See “[Intrinsic Delay](#)” on page 3-8.

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. Pin D in [Example 4-14 on page 4-131](#) illustrates the intrinsic delay attributes used as timing constraints. Timing constraints are not used in the delay equation.

---

## In the CMOS Generic Delay Model

The `intrinsic_rise` and `intrinsic_fall` attributes describe the intrinsic delay of a pin when you use the CMOS generic delay model. See “[CMOS Generic Delay Model](#)” on page 3-3 for details.

### **intrinsic\_rise Simple Attribute**

On an output pin, this value defines

- The 0-to-1 propagation time for a combinational cell
- The 0-to-Z propagation time for a three-state-disable timing type
- The Z-to-1 propagation time for a three-state-enable timing type

On an input pin, this value defines a setup, hold, or recovery timing requirement for a logic 0-to-1 transition.

#### **Example**

```
intrinsic_rise : 1.0;
```

### **intrinsic\_fall Simple Attribute**

On an output pin, this value defines

- The 1-to-0 propagation time for a combinational cell
- The 1-to-Z propagation time for a three-state-disable timing type
- The Z-to-0 propagation time for a three-state-enable timing type

On an input pin, this value defines a setup, hold, or recovery timing requirement for a logic 1-to-0 transition.

## Example

This example shows the timing group for a 2-input NAND gate with an intrinsic rise delay of 0.58 and an intrinsic fall delay of 0.69.

```
pin (Z) {  
    direction : output ;  
    function : "(A * B)' " ;  
    timing () {  
        related_pin : "A B" ;  
        intrinsic_rise : 0.58 ;  
        intrinsic_fall : 0.69 ;  
        ...  
    }  
}
```

---

## In the CMOS Piecewise Linear Delay Model

You describe the intrinsic delay in the CMOS piecewise linear delay model the same way you describe it in the CMOS generic delay model. See “[intrinsic\\_rise Simple Attribute](#)” on page 4-62 and “[intrinsic\\_fall Simple Attribute](#)” on page 4-62 for details.

---

## In the CMOS Nonlinear Delay Model

The description of intrinsic delay is inherent in the lookup tables you create for this delay model. See “[Defining the CMOS Nonlinear Delay Model Template](#)” on page 4-23 to find out how to create and use templates for lookup tables in a library, using the CMOS nonlinear delay model.

---

## In the Scalable Polynomial Delay Model

The description of intrinsic delay is inherent in the polynomials you create for this delay model. See “[Defining the Scalable Polynomial Delay Model Template](#)” on page 4-30 to learn how to create and use templates for scalable polynomials in a library, using the CMOS scalable polynomial nonlinear delay model.

---

## 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.

---

## In the CMOS Generic Delay Model

Use the following `timing` group attributes exclusively for generic delay models. In the attribute statements, the value is a positive floating-point number for the delay time per load unit.

### **`rise_resistance` Simple Attribute**

This value represents the output resistance, or drive capability, for a logic 0-to-1 transition.

## **fall\_resistance Simple Attribute**

This value represents the output resistance, or drive capability, for a logic 1-to-0 transition.

Note:

You cannot specify a resistance unit in the library. Instead, the resistance unit is derived from the ratio of the `time_unit` value to the `capacitive_load_unit` value.

## **Example**

This example is a combinational timing description of a NAND gate in a library that uses a standard CMOS generic delay model.

```
cell (NAND2) {  
    area : 1 ;  
    pin(A, B) {  
        direction : input ;  
        capacitance : 1 ;  
    }  
    pin(Z) {  
        direction : output ;  
        function : "(A * B)" ;  
        timing() {  
            intrinsic_rise : 0.17 ;  
            intrinsic_fall : 0.10 ;  
            rise_resistance : 0.65 ;  
            fall_resistance : 0.18 ;  
            related_pin : "A" ;  
        }  
        timing() {  
            intrinsic_rise : 0.21 ;  
            intrinsic_fall : 0.12 ;  
            rise_resistance : 0.65 ;  
            fall_resistance : 0.18 ;  
            related_pin : "B" ;  
        }  
    }  
}
```

---

## In the CMOS Piecewise Linear Delay Model

When using a piecewise linear delay model, you must include the `piece_define` attribute statement in the `library` group; the `piece_type` attribute statement is optional (see the *Library Compiler User Guide: Methodology and Modeling Functionality in Technology Libraries*). See “[CMOS Piecewise Linear Delay Model](#)” on page 3-37 for the delay equations.

The transition delay for piecewise linear equations is modeled with delay-intercept attributes and pin-resistance attributes. Delay-intercept attributes define the intercept for vendors that use slope- or intercept-type timing equations. Pin-resistance attributes describe the resistance encountered during logic transitions.

### **`rise_delay_intercept` Complex Attribute**

This attribute defines the intercept for vendors using slope- or intercept-type timing equations. The value of this attribute is added to the rising edge in the delay equation.

### **`fall_delay_intercept` Complex Attribute**

This attribute defines the intercept for vendors using slope- or intercept-type timing equations. The value of this attribute is added to the falling edge in the delay equation.

### **`rise_pin_resistance` Complex Attribute**

This attribute defines the drive resistance applied to pin loads in the rising edge in the transition delay equation.

## **fall\_pin\_resistance Complex Attribute**

This attribute defines the drive resistance applied to pin loads in the falling edge in the transition delay equation.

### **Syntax**

This is the syntax for the following timing group attribute statements in which *piece* is an integer defining the wire segment in accordance with the `piece_define` statement and *value* is a floating-point number.

```
attribute_name (piece, "value");
```

[Example 4-4](#) shows how to define a NOR gate in a library that uses a CMOS piecewise linear delay model.

### *Example 4-4 Combinational Timing Description (Piecewise Linear Delay Model)*

```
library (example) {
    date : "August 14, 2002" ;
    revision : 2000.05;
    piece_define ( "0 15 40" ) ;
    cell(NOR2) {
        area : 1;
        pin(A, B) {
            direction : input;
            capacitance : 1;
        }
        pin(Z) {
            direction : output;
            function : "(A+B)'";
            timing() {
                intrinsic_rise : 0.38;
                intrinsic_fall : 0.15;
                rise_delay_intercept(0,"1.0"); /* piece 0 */
                rise_delay_intercept(1,"0.0"); /* piece 1 */
                rise_delay_intercept(2,"-1.0"); /* piece 2 */
                fall_delay_intercept(0,"1.0"); /* piece 0 */
                fall_delay_intercept(1,"0.0"); /* piece 1 */
            }
        }
    }
}
```

```

    fall_delay_intercept(2, "-1.0"); /* piece 2 */
    rise_pin_resistance(0, "0.25"); /* piece 0 */
    rise_pin_resistance(1, "0.50"); /* piece 1 */
    rise_pin_resistance(2, "1.00"); /* piece 2 */
    fall_pin_resistance(0, "0.25"); /* piece 0 */
    fall_pin_resistance(1, "0.50"); /* piece 1 */
    fall_pin_resistance(2, "1.00"); /* piece 2 */
    related_pin : "A B";
}
}
}

```

## In the CMOS Nonlinear Delay Model

Transition time is the time it takes for an output signal to make a transition between the high and low logic states. With nonlinear delay models, 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 [Chapter 3, “Delay Models,”](#) and the *Library Compiler Reference Manual: Technology and Symbol Libraries* 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

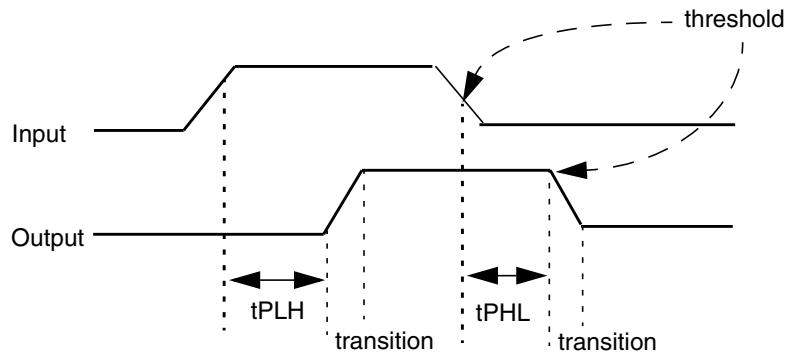
```
cell (general) {  
    pin(a) {  
        direction: output;  
        timing() {  
            cell_rise(basic_template) {  
                values ("0.0, 0.13, 0.17, 0.19", "0.21, 0.23, \  
                        0.30, 0.41", "0.22, 0.31, 0.35, 0.47", \  
                        "0.33, 0.37, 0.45, 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 4-6](#).

If you define propagation delay tables for a timing arc, you cannot specify cell delay tables for that arc.

*Figure 4-6 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(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" ;
    }
}
```

See “[Specifying Delay Scaling Attributes](#)” on page 12-35 for information about calculating delay factors. For information about including propagation delay in total cell delay calculations, see “[Propagation Delay](#)” on page 3-24.

[Example 4-5](#) shows the use of lookup tables and templates for describing cell delay with the CMOS nonlinear delay model.

### *Example 4-5 Using Templates in the CMOS Nonlinear Delay Model*

```

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, \
                            0.35, 0.37, 0.39", "0.38, 0.40, 0.42, 0.44", "0.46, 0.48, 0.50, 0.52", "0.54, 0.56, 0.58, 0.60", "0.62, 0.64, 0.66, 0.68", "0.70, 0.72, 0.74, 0.76", "0.78, 0.80, 0.82, 0.84", "0.86, 0.88, 0.90, 0.92", "0.94, 0.96, 0.98, 1.00");
                }
            }
        }
    }
}

```

```

        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.

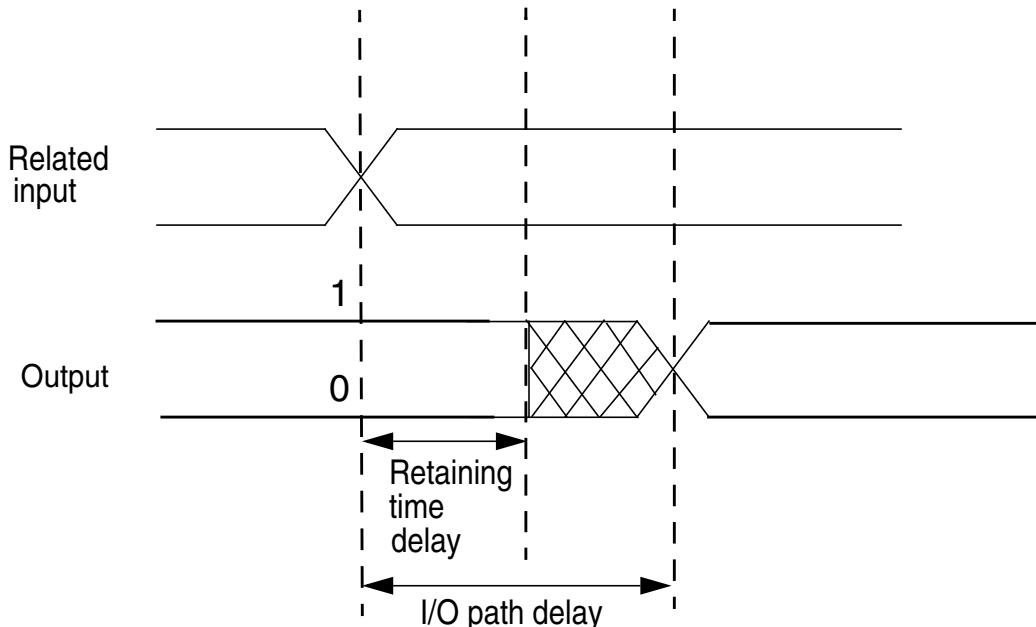
The value you enter for the `retaining_fall` attribute determines how long the output will retain its current value, 1, after the value at the related input port has changed.

Note:

Retaining time works only on a nonlinear delay model.

[Figure 4-7](#) shows retaining delay in regard to changes in a related input port.

*Figure 4-7 Retaining Time Delay*



**Example 4-6** shows how to use the `retaining_rise` and `retaining_fall` attributes.

### *Example 4-6 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 12-35 for information about calculating delay factors. For information about including propagation delay in total cell delay calculations, see “[Propagation Delay](#)” on page 3-24.

### **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 happens not only at a different time than the propagation of the final logical value but also 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 will retain its current value, 0, after the value at the related input port has changed.

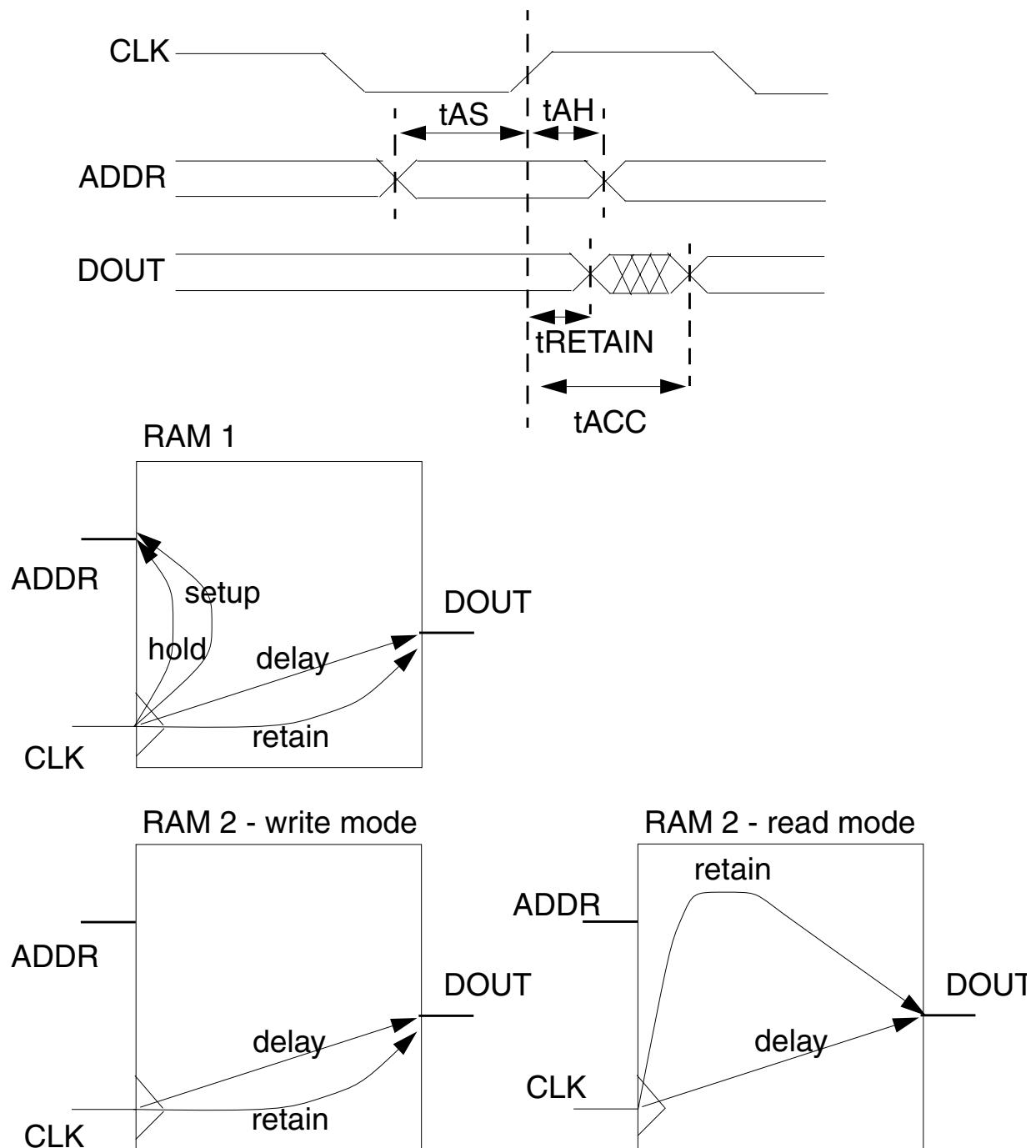
The value you enter for the `retain_fall_slew` attribute determines how long the output will retain its current value, 1, after the value at the related input port has changed.

**Note:**

Retaining time works only on a nonlinear delay model.

[Figure 4-8](#) shows a timing diagram of synchronous RAM.

Figure 4-8 Timing Diagram of Synchronous RAM



## Example

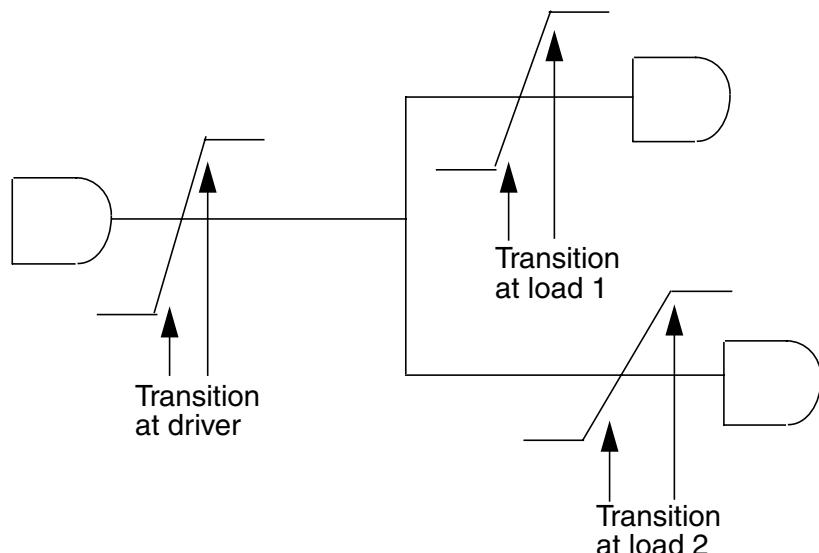
```
library(library_name) {  
    ...  
    lu_table_template (retaining_table_template) {  
        ...  
        variable_1: total_output_net_capacitance;  
        variable_2: input_net_transition;  
        index_1 ("0.0, 1.5");  
        index_2 ("1.0, 2.1");  
    }  
    ...  
    cell (cell_name) {  
        ...  
        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 12-35 for information about calculating delay factors. For information about including propagation delay in total cell delay calculations, see “[Propagation Delay](#)” on page 3-24.

## 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 4-9](#). The higher the interconnect load, the greater the flattening effect and the greater the transition delay.

*Figure 4-9 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, as long as 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))
```

For more information about function f, see the “[CMOS Nonlinear Delay Model Calculation](#)” on page 3-25.

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 4-7** shows the use of the degradation tables. In this example, `trans_deg` is the name of the template for the transition degradation.

### *Example 4-7 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.

See “[CMOS Nonlinear Delay Model](#)” on page 3-20 for more information about calculating CMOS nonlinear timing models.

---

## Modeling Load Dependency

“Describing Transition Delay” on page 4-64 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.

Load-dependent constraints are allowed in the CMOS nonlinear delay model and in the scalable polynomial delay model.

---

### In the CMOS Nonlinear Delay Model

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 4-4.
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 CMOS Nonlinear Delay Model Template” on page 4-23.

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 4-28.) The following groups are valid lookup tables for output load modeling:

- `rise_constraint`
- `fall_constraint`

See “[Setting Setup and Hold Constraints](#)” on page 4-94 for information about these groups.

## **Example of a Load-Dependent Model**

[Example 4-8](#) is an example of a library that includes a load-dependent model.

### *Example 4-8 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");
}
...
}
...
}

```

## In the CMOS Scalable Polynomial Delay Model

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 4-4](#).
2. Specify the output pin used to figure the load dependency with the `related_output_pin` attribute described below.
3. Create a three-dimensional table template that uses two variables to model transition time and a third variable, `poly_template`, 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 Scalable Polynomial Delay Model Template” on page 4-30.](#)

4. Create a three-dimensional lookup table, using the table template and the `index_3` attribute in the lookup table group.

Express the delay equation in terms of scalable polynomial delay delay coefficients, using the `variable_3` variable and the `variable_3_range` attribute in the `poly_template` group.

- rise\_constraint
- fall\_constraint

See “[Setting Setup and Hold Constraints](#)” on page 4-94 for information about these groups.

## **Example of a Load-Dependent Model**

[Example 4-9](#) is an example of a library that includes a load-dependent model.

### *Example 4-9 Load-Dependent Model*

```
library(load_dependent) {
    ...
    technology (cmos) ;
    delay_model : polynomial ;
    ...
    poly_template ( const ) {
        variables (constrained_pin_transition, related_pin_transition, \
                   related_out_total_output_net_capacitance);
        variable_1_range (0.0000, 4.0000);
        variable_2_range (0.0000, 4.0000);
        variable_3_range (0.0000, 4.0000);
    }
    ...
    cell(example) {
        ...
        pin(D) {
            direction : input ;
            capacitance : 1.00 ;
            timing() {
                timing_type : setup_rising ;
                fall_constraint(const) {
                    orders ("2, 1, 1")
                    coefs ("1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0");
                }
                rise_constraint(const){
                    orders ("2, 1, 1")
                    coefs ("1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0");
                }
                related_pin : "CP" ;
                related_output_pin : "Q";
            }
            timing() {
                timing_type : hold_rising ;
            }
        }
    }
}
```

```

        rise_constraint(const) {
            orders ("1, 1, 1")
            coefs ("1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0");
        }
        fall_constraint(const) {
            orders ("1, 1, 1")
            coefs ("1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0");
        }
        related_pin : "CP" ;
        related_output_pin : "Q";
    }
}
...
} /* end cell */
...
} /* end library */

```

## 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. It is defined in the timing group of the driving pin.

The value in the attribute statements for slope sensitivity is a positive floating-point number that results in slope delay when multiplied by the transition delay.

---

## In the CMOS Generic Delay Model and Piecewise Linear Delay Model

Use these slope sensitivity attributes for CMOS generic or piecewise linear technology only.

### **slope\_rise Simple Attribute**

This value represents the incremental delay to add to the slope of the input waveform for a logic 0-to-1 transition.

#### **Example**

```
slope_rise : 0.0;
```

### **slope\_fall Simple Attribute**

This value represents the incremental delay to add to the slope of the input waveform for a logic 1-to-0 transition.

#### **Example**

```
slope_fall: 0.0;
```

---

## Describing State-Dependent Delays

These timing attributes describe the delay values for specified conditions.

In the `timing` group of a technology library, you can specify state-dependent delays that correspond to entries in Open Verilog International Standard Delay Format (OVI SDF 2.1) syntax. See the *Library Compiler Reference Manual: Technology and Symbol Libraries* 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 4-128 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 4-5](#) for the valid Boolean operators.

*Table 4-5 Valid Boolean Operators*

Operator	Description
,	invert previous expression
!	invert following expression
<code>^</code>	logical XOR

*Table 4-5 Valid Boolean Operators (Continued)*

Operator	Description
*	logical AND
&	logical AND
space	logical AND
+	logical OR
	logical OR
1	signal tied to logic 1
0	signal tied to logic 0

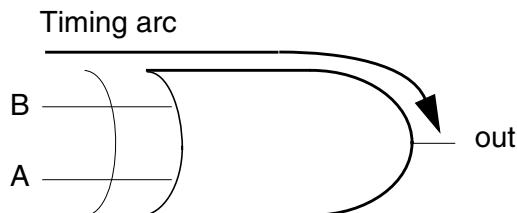
The order of precedence of the operators is left to right, with inversion performed first, then XOR, then AND, then OR.

### Example

```
when : "B" ;
```

Figure 4-10 shows an XOR gate.

*Figure 4-10 XOR Gate With State-Dependent Timing Arc*



Example 4-10 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 4-37 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 should be logically equivalent to the `when` attribute in the same timing arc. If these two Boolean expressions are not equivalent, back-annotation is not performed properly.

Note:

Library Compiler does not parse the `sdf_cond` expression, nor does it check for logical equivalency between these two expressions.

**Example 4-10** is a 2-input XOR gate. It represents a commonly used state-dependent delay case. The intrinsic 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 `intrinsic_rise` 1.4 and `intrinsic_fall` 1.6 when pin B = 0.

#### *Example 4-10 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 ";
            intrinsic_rise : 1.3;
            intrinsic_fall : 1.5;
        }
        timing() {
            related_pin : "A";
            timing_sense : positive_unate;
            when : "!B";
            sdf_cond : " B == 1'B0 ";
            intrinsic_rise : 1.4;
            intrinsic_fall : 1.6;
        }
        timing() /* default timing arc */
    }
}
```

```

        related_pin : "A";
        timing_sense : non_unate;
        intrinsic_rise : 1.4;
        intrinsic_fall : 1.6;
    }
    timing() {
        related_pin : "B";
        timing_sense : negative_unate;
        when : "A";
        sdf_cond : "A == 1'B1 ";
        intrinsic_rise : 1.3;
        intrinsic_fall : 1.5;
    }
    timing() {
        related_pin : "B";
        timing_sense : positive_unate;
        when : "!A";
        sdf_cond : "A == 1'B0 ";
        intrinsic_rise : 1.4;
        intrinsic_fall : 1.6;
    }
    timing() /* default timing arc */
    {
        related_pin : "B";
        timing_sense : non_unate;
        intrinsic_rise : 1.4;
        intrinsic_fall : 1.6;
    }
}
}

```

## Setting Setup and Hold Constraints

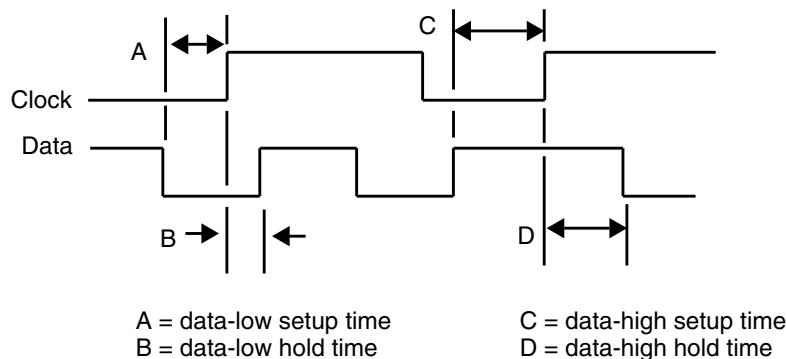
Signals arriving at an input pin have ramp times. Therefore, you must ensure that the data signal has stabilized before latching its value by defining setup and hold arcs as timing requirements. No path tracing takes place along these timing arcs; they are used solely for constraint verification.

- 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 may 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 may be latched.

By combining a setup time and a hold time, you can ensure the stability of data that is latched.

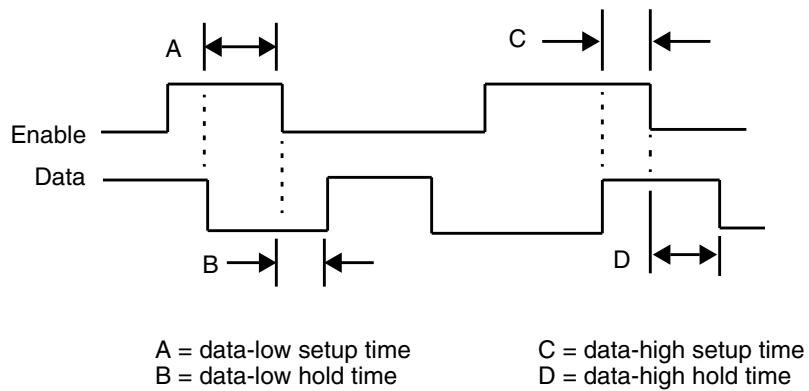
**Figure 4-11** shows setup and hold timing for a rising-edge-triggered flip-flop. The timing checks for flip-flops use the activating edge of the clock, which is the rising edge in this case.

*Figure 4-11 Setup and Hold Constraints for Rising-Edge-Triggered Flip-Flop*



**Figure 4-12** 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 4-12 Setup and Hold Constraints for High-Enable Latch*



---

## In the CMOS Generic Delay Model and Piecewise Linear Delay Model

Use setup and hold constraints only between data pins and clock pins.

### Setup Constraints

The values you can assign to a `timing_type` attribute to define timing arcs used for setup checks on clocked elements are

`setup_rising`

Designates the rising edge of the related pin for the setup check.

`setup_falling`

Designates the falling edge of the related pin for the setup check.

To define a setup constraint,

1. Assign one of the constraint values to the `timing_type` attribute.

2. Specify a related pin in the `timing` group.

The `related_pin` attribute in a timing arc with a `setup_rising` or `setup_falling` timing type identifies the pin used for the timing check. The rising or falling designation tells Design Compiler which edge of the related pin is active for the setup check.

### Example

This example describes the setup arc for the data pin on the rising-edge-triggered D flip-flop shown in [Figure 4-11](#). The `intrinsic_rise` value represents setup time C, and the `intrinsic_fall` value represents setup time A. The syntax shown assumes a generic or piecewise linear delay model.

```
timing() {
    timing_type : setup_rising ;
    intrinsic_rise : 1.5 ;
    intrinsic_fall : 1.5 ;
    related_pin : "Clock" ;
}
```

The following example describes the setup constraint for the data pin on the high-enable latch shown in [Figure 4-12](#). The `intrinsic_rise` value represents setup time C, and the `intrinsic_fall` value represents setup time A.

```
timing() {
    timing_type : setup_falling ;
    intrinsic_rise : 1.5 ;
    intrinsic_fall : 1.5 ;
    related_pin : "Enable" ;
}
```

Note:

The VHDL library generator accepts only one setup or hold value. For example, if you use both the `intrinsic_rise` and the `intrinsic_fall` attributes, the library generator uses the maximum of the two values.

## Hold Constraints

The values you can assign to the `timing_type` attribute to define timing constraints used for hold checks on clocked elements are

`hold_rising`

This value designates the rising edge of the related pin for the hold check.

`hold_falling`

This value designates the falling edge of the related pin for the hold check.

To define a hold constraint,

1. Assign one of the constraint values to the `timing_type` attribute.
2. Specify a related pin in the `timing` group.

The `related_pin` attribute in a timing arc with a `hold_rising` or `hold_falling` timing type identifies the pin used for the timing check. The rising or falling designation tells Design Compiler which edge of the related pin is active for the hold check.

## Example

This example shows the hold constraint for pin D on a rising-edge-triggered D flip-flop. The `intrinsic_rise` value represents hold time B in [Figure 4-11](#), and the `intrinsic_fall` value is hold time D.

```
timing() {
    timing_type : hold_rising;
    intrinsic_rise : 0.5 ;
    intrinsic_fall : 0.5 ;
    related_pin : "Clock" ;
}
```

The following example shows the hold constraint for the data pin on a high-enable latch. The `intrinsic_rise` value represents hold time B in [Figure 4-12](#), and the `intrinsic_fall` value represents hold time D.

```
timing() {
    timing_type : hold_falling ;
    intrinsic_rise : 1.5 ;
    intrinsic_fall : 1.5 ;
    related_pin : "Enable" ;
}
```

## Setup and Hold Timing Constraints

You can describe a complete setup and hold timing constraint by combining a setup constraint and a hold constraint. The following example shows setup and hold timing constraints on pin K in a JK flip-flop.

```
pin(K) {
    direction : input ;
    capacitance : 1.3 ;
    timing() {
        timing_type : setup_rising ;
        intrinsic_rise : 1.5 ;
        intrinsic_fall : 1.5 ;
```

```

        related_pin : "CP" ;
    }
    timing() {
        timing_type : hold_rising ;
        intrinsic_rise : 0.0 ;
        intrinsic_fall : 0.0 ;
        related_pin : "CP" ;
    }
}

```

---

## In the CMOS Nonlinear Delay Model

The CMOS nonlinear timing model can support timing constraints that are 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 constraint tables replace the `intrinsic_rise` and `intrinsic_fall` attributes used in the other delay models. The format of the lookup table template and the format of the lookup table are the same as described previously in “[Defining the CMOS Nonlinear Delay Model Template](#)” on page 4-23 and “[Creating Lookup Tables](#)” on page 4-28.

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 4-11](#) shows how to use tables to specify setup constraints for a flip-flop.

### *Example 4-11 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");
                }
            }
        }
    }
}
```

```
    }
}
```

---

## In the Scalable Polynomial Delay Model

Example 4-12 shows how to specify constraint in a scalable polynomial delay model.

### *Example 4-12 CMOS Scalable Polynomial Delay Model Using Constraint*

```
library(vendor_b) {
    /* Use polynomial delay model */
    delay_model : polynomial;
    /* Define template */
    poly_template ( constraint_template_poly ) {
        variables (constrained_pin_transition, related_pin_transition);
        variable_1_range (0.01, 3.00);
        variable_2_range (0.01, 3.00);
    }
    .....
    cell(dff) {
        pin(D) {
            direction : input ;
            timing() {
                related_pin : "CP" ;
                timing_type : setup_rising ;
                rise_constraint ( constraint_template_poly ) {
                    orders("1, 1");
                    /*  $(a_0 + a_1x_1)(b_0 + b_1x_2) = A_{00} + A_{10}x_1 + A_{01}x_2 + A_{11}x_1x_2$  */
                    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 ( constraint_template_poly ) {
                    orders("1, 2");
                    /*  $(a_0 + a_1x_1)(b_0 + b_1x_2 + b_2x_2^2) =$  */
                    /*  $A_{00} + A_{10}x_1 + A_{01}x_2 + A_{11}x_1x_2 + A_{02}x_2^2 + A_{12}x_1x_2^2$  */
                    coefs ("0.2732 +0.0668 +0.0216 -0.0002 -0.0003 +0.0000 " );
                    variable_1_range (0.01, 3.00);
                    variable_2_range (0.01, 3.00);
                }
            }
        }
    .....
}
```

```
.....  
}
```

---

## 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 *Library Compiler Reference Manual: Technology and Symbol Libraries*.

---

## 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;
        intrinsic_rise : 1.5;
        intrinsic_fall : 1.5;
```

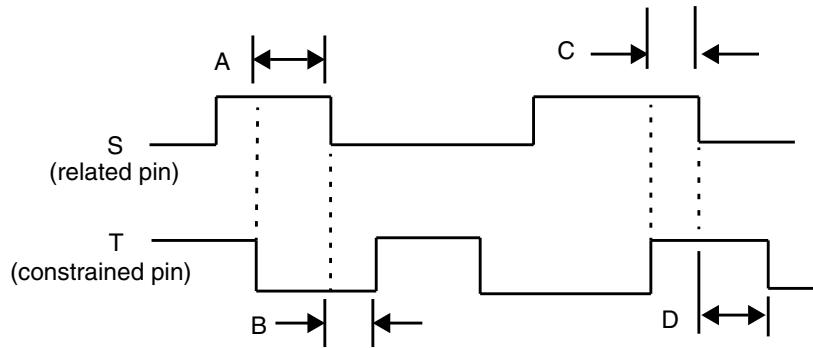
```

        related_pin : "S";
    }
}

```

**Figure 4-13** 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 4-13 Nonsequential Setup and Hold Constraints*



A = intrinsic\_fall in setup time      C = intrinsic\_rise in setup time  
 B = intrinsic\_rise in hold time      D = intrinsic\_fall in hold time

---

## Setting Recovery and Removal Timing Constraints

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

---

### Recovery Constraints

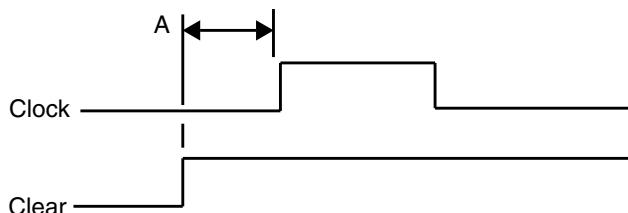
The recovery timing arc describes the minimum allowable time between the control pin transition to the inactive state and the active edge of the synchronous clock signal (time between the control signal going inactive and the clock edge that latches data in).

The asynchronous control signal must remain constant during this time, or else an incorrect value may appear at the outputs.

[Figure 4-14](#) shows the recovery timing arc for a rising-edge-triggered flip-flop with active-low clear.

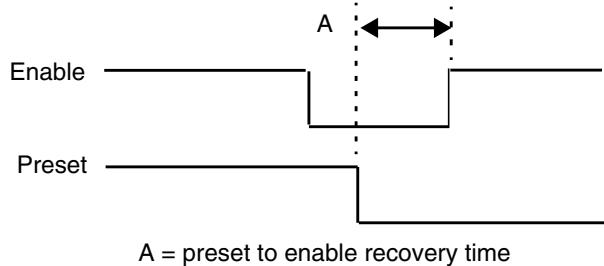
[Figure 4-15](#) shows the recovery timing arc for a low-enable latch with active-high preset.

*Figure 4-14 Recovery Timing Constraint for a Rising-Edge-Triggered Flip-Flop*



A = clear to clock recovery time

*Figure 4-15 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 `intrinsic_rise` statement.

For active-high control signals, define the recovery time with the `intrinsic_fall` statement.

## Example

This example shows a recovery timing arc for the active-low clear signal in a rising-edge-triggered flip-flop. The `intrinsic_rise` value represents clock recovery time A in [Figure 4-14](#).

```
pin (Clear) {  
    direction : input ;  
    capacitance : 1 ;  
    timing() {  
        related_pin : "Clock" ;  
        timing_type : recovery_rising;  
        intrinsic_rise : 1.0 ;  
    }  
}
```

The following example shows a recovery timing arc for the active-high preset signal in a low-enable latch. The `intrinsic_fall` value represents clock recovery time A in [Figure 4-15](#).

```
pin (Preset) {  
    direction : input ;  
    capacitance : 1 ;  
    timing() {  
        related_pin : "Enable" ;  
        timing_type : recovery_rising;  
        intrinsic_fall : 1.0 ;  
    }  
}
```

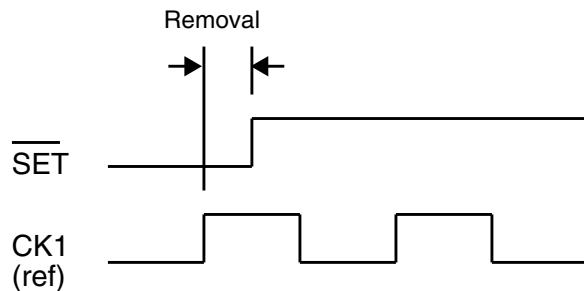
---

## 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 4-16](#)).

*Figure 4-16 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 `intrinsic_rise` attribute.

For active-high asynchronous control signals, define the removal time with the `intrinsic_fall` attribute.

## **Example**

```
pin ( SET ) {  
    ....  
    timing() {
```

```
    timing_type : removal_rising;
    related_pin : " CK1 ";
    intrinsic_rise : 1.0 ;
}
}
```

---

## Setting No-Change Timing Constraints

You can model no-change timing checks to use in static timing verification during synthesis.

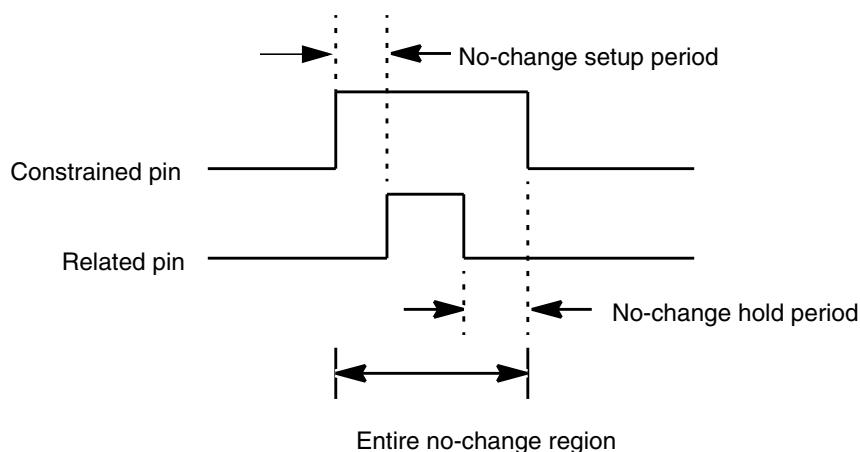
A no-change timing check checks a constrained signal against a level-sensitive related signal. The constrained signal must remain stable during an established setup period, for the width of the related pulse, and during an established hold period.

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 4-17](#) shows a no-change timing check between a constrained pin and its level-sensitive related pin.

*Figure 4-17 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`

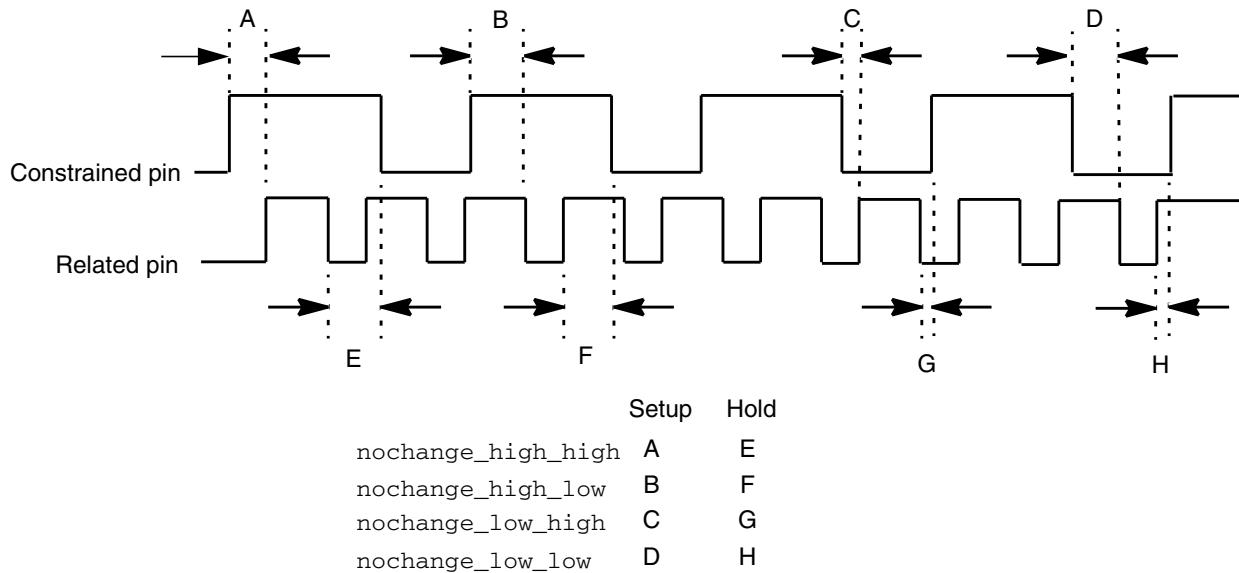
Specifies a negative pulse on the constrained pin and a positive pulse on the related pin.

`nochange_low_low`

Specifies a negative pulse on the constrained pin and a negative pulse on the related pin.

[Figure 4-18](#) shows the waveforms for these constraints.

Figure 4-18 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 [Table 4-6](#).

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 4-120 for more information.

*Table 4-6 Summary of No-Change Constraints and Delay Model Attributes*

No-change constraint	Setup attribute for generic delay and nonlinear delay models	Hold attribute for generic delay and nonlinear delay models
nochange_high_high	intrinsic_rise / rise_constraint	intrinsic_fall / fall_constraint
nochange_high_low	intrinsic_rise / rise_constraint	intrinsic_fall / fall_constraint
nochange_low_high	intrinsic_fall / fall_constraint	intrinsic_rise / rise_constraint
nochange_low_low	intrinsic_fall / fall_constraint	intrinsic_rise / rise_constraint

---

## In the CMOS Generic Delay Model

In the CMOS generic delay model, specify setup time with `intrinsic_rise` and hold time with `intrinsic_fall`.

This is the syntax for the no-change timing check in the CMOS generic delay model:

```
timing () {
    timing_type : nochange_high_high | nochange_high_low |
                  nochange_low_high | nochange_low_low;
    related_pin : related_pinname;
    intrinsic_rise : float; /* constrained signal rising */
    intrinsic_fall : float; /* constrained signal falling */
}
```

---

## In the CMOS Nonlinear Delay Model

In the CMOS nonlinear 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 nonlinear delay model:

```
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 using the CMOS nonlinear delay model:

```
library (the_lib) {
    delay_model : polynomial;
    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 (polynomial) { /* setup time */
```

```

        values (2.98);
    }
    fall_constraint (polynomial) { /* 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_nameid) {
        /* constrained signal rising */
        orders(integer, integer) ;
        coefs(float, ..., float) ;
        variable_n_range(float, float) ;
    }
    fall_constraint (poly_template_nameid) {
        /* 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);
                }
            }
        }
        ...
    }
    ...
}
```

---

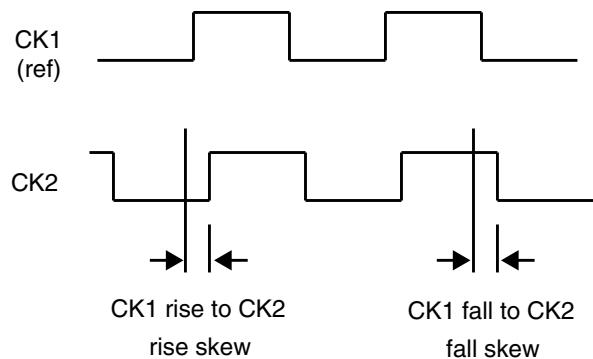
## Setting Skew Constraints

The skew constraint defines the maximum separation time allowed between two clock signals.

The VHDL library generator uses the skew constraint for simulation.

Figure 4-19 is a timing diagram showing skew constraint.

*Figure 4-19 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 `intrinsic_rise` value is the maximum skew time between the reference pin rising and the parent pin rising. The `intrinsic_fall` 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 `intrinsic_rise` value is the maximum skew time between the reference pin rising and the parent pin rising. The `intrinsic_fall` 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`:
  - `intrinsic_rise`
  - `intrinsic_fall`
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`
  - `intrinsic_rise`
  - `intrinsic_fall`

## Example

This example shows how to model constraint CK1 rise to CK2 rise skew:

```
pin (CK2) {  
    ....  
    timing() {
```

```
    timing_type : skew_rising;
    related_pin : " CK1 ";
    intrinsic_rise : 1.0 ;
}
}
```

---

## Setting Conditional Timing Constraints

A conditional timing constraint describes a check Design Compiler should perform 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. Use `when` to model timing check conditions for VITAL models.

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. Library Compiler applies `when` and `sdf_cond` attribute values to both start and end conditions. Library Compiler 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 4-88](#) 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, Library Compiler ignores the attribute.

## **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. [Example 4-13](#) shows how to specify a lookup table in a `min_pulse_width` group.

*Example 4-13 Lookup table for min\_pulse\_width group.*

```
library(ASIC) {
    delay_model : table_lookup;
    ...
    lu_table_template(pulse_width_template) {
        variable_1 : related_pin_transition;
        index_1 ("1.0, 2.0, 3.0");
    }
    cell(flop) {
        ...
        pin(CK) {
            direction : input;
            capacitance : 0.00707171;
            timing() {
                timing_type : "min_pulse_width";
                related_pin : "CK";
                rise_constraint("pulse_width_template") {
                    index_1(" 0.000000, 1.000000, 2.000000");
                    values(" 6.000000, 6.250000, 7.250000" );
                }
                fall_constraint("pulse_width_template") {
                    index_1(" 0.000000, 1.000000, 2.000000");
                    values(" 6.000000, 6.250000, 7.250000" );
                }
            }
            timing() {
                timing_type : "minimum_period";
                related_pin : "CK";
                rise_constraint("pulse_width_template") {
                    index_1(" 0.000000, 1.000000, 2.000000");
                    values(" 6.000000, 6.250000, 7.250000" );
                }
            }
        }
    }
}
```

```

        fall_constraint("pulse_width_template") {
            index_1(" 0.000000, 1.000000, 2.000000");
            values(" 6.000000, 6.250000, 7.250000 " );
        }
    }
    ...
} /* end cell */
} /* end library */

```

## **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, Library Compiler ignores the `min_period` attribute.

## **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.

---

## **Using Conditional Attributes With No-Change Constraints**

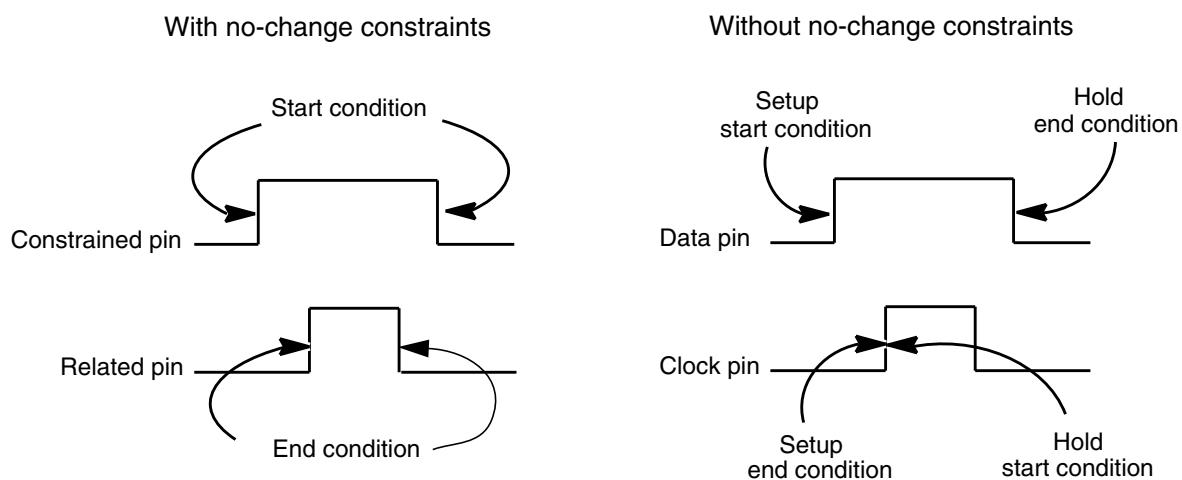
As shown in [Table 4-7](#), conditional timing check attributes have different interpretations when you use them with no-change timing constraints. See “[Setting No-Change Timing Constraints](#)” on [page 4-111](#) for a description of no-change timing constraint values.

*Table 4-7 Conditional Timing Attributes With No-Change Constraints*

Conditional attributes	With no-change constraints
<code>when</code> <code>sdf_cond</code>	Defines both the constrained and related signal-enabling conditions
<code>when_start</code> <code>sdf_cond_start</code>	Defines the constrained signal-enabling condition
<code>when_end</code> <code>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 4-20](#) shows the different interpretation of setup and hold conditions when used with a no-change timing constraint.

**Figure 4-20 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, using Design Compiler, and then using the `write_timing` command to write out an SDF file. See the Synopsys man pages for more information on the `write_timing` command.

An SDF file generated by Synopsys tools may contain conditional path delays. You create these path delays by putting the expression string, which you define in the `conditional_port_expr`, 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. Library Compiler ignores all white spaces during the string matching.

A common procedure is to use Library Compiler 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.

---

## Timing Arc Restrictions

The following section describes timing arc limitations.

---

### Impossible Transitions

The information in this section applies to the table lookup and all other delay models 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 will cause 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).

Library Compiler detects such invalid timing arcs if they are specified. For the impossible 0-to-1 and 1-to-0 transitions, Library Compiler deletes the timing arc. For the impossible 0-to-Z, Z-to-0, 1-to-Z, and Z-to-1 transitions, Library Compiler puts either the predefined `non_rising` or `non_falling` attribute on the timing arc.

---

## Examples of Libraries Using Delay Models

This section contains examples of libraries using the following CMOS delay models: generic delay, piecewise linear delay, nonlinear delay, and scalable polynomial delay.

---

### CMOS Generic Delay Model

In the CMOS D flip-flop description in [Example 4-14](#), pin D contains setup and hold constraints; pins Q and QN contain preset, clear, and edge-sensitive arcs.

#### *Example 4-14 D Flip-Flop Description (CMOS Generic Delay Model)*

```
library (example){  
    date : "January 14, 2002";  
    revision : 2000.01;  
    delay_model : generic_cmos;  
    technology (cmos);  
    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;  
                intrinsic_rise : 0.12;  
                intrinsic_fall : 0.12 ;  
            }  
            timing (){  
                related_pin : " CLK" ;  
            }  
        }  
    }  
}
```

```

        timing_type : setup_rising ;
        intrinsic_rise : 2.77 ;
        intrinsic_fall : 2.77 ;
    }
}
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 ;
        intrinsic_rise : 0.65 ;
        rise_resistance : 0.047 ;
        slope_rise : 1.0 ;
    }
    timing (){
        related_pin : "CLR" ;
        timing_type : clear ;
        timing_sense : positive_unate ;
        intrinsic_fall : 1.45 ;
        fall_resistance : 0.066 ;
        slope_fall : 0.8 ;
    }
    timing (){
        related_pin : "CLK" ;
        timing_type : rising_edge;
        intrinsic_rise : 1.40 ;
        intrinsic_fall : 1.91 ;
        rise_resistance : 0.071 ;
        fall_resistance : 0.041 ;
    }
}

```

```

pin ( QN ){
    direction : output ;
    function : "IQN" ;
    timing (){
        related_pin : "PRE" ;
        timing_type : clear ;
        timing_sense : positive_unate ;
        intrinsic_fall : 1.87 ;
        fall_resistance : 0.053 ;
        slope_fall : 1.2 ;
    }
    timing (){
        related_pin : "CLR" ;
        timing_type : preset ;
        timing_sense : negative_unate ;
        intrinsic_rise : 0.68 ;
        rise_resistance : 0.054 ;
        slope_rise : 1.0 ;
    }
    timing (){
        related_pin : "CLK" ;
        timing_type : rising_edge;
        intrinsic_rise : 2.37 ;
        intrinsic_fall : 2.51 ;
        rise_resistance : 0.036 ;
        fall_resistance : 0.041 ;
    }
}
}

```

# CMOS Piecewise Linear Delay Model

In the CMOS piecewise linear D flip-flop description in [Example 4-15](#), pin D contains setup and hold constraints; pins Q and QN contain preset, clear, and edge-sensitive arcs.

### *Example 4-15 D Flip-Flop Description (CMOS Piecewise Linear Delay Model)*

```
library (example){  
date : "January 14, 2002";
```

```

revision : 2000.01;
delay_model : piecewise_cmos;
technology (cmos);
piece_define("0,15,40");
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;
            intrinsic_rise : 0.12;
            intrinsic_fall : 0.12 ;
        }
        timing (){
            related_pin : "CLK" ;
            timing_type : setup_rising ;
            intrinsic_rise : 2.77 ;
            intrinsic_fall : 2.77 ;
        }
    }
    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 : positive_unate ;
    intrinsic_rise : 0.65 ;
    rise_delay_intercept (0,0.054); /* piece 0 */
    rise_delay_intercept (1,0.0); /* piece 1 */
    rise_delay_intercept (2,-0.062); /* piece 2 */
    rise_pin_resistance (0,0.25); /* piece 0 */
    rise_pin_resistance(1,0.50); /* piece 1 */
    rise_pin_resistance (2,1.00); /* piece 2 */
}
timing (){
    related_pin : "CLR" ;
    timing_type : clear ;
    timing_sense : negative_unate ;
    intrinsic_fall : 1.45 ;
    fall_delay_intercept (0,1.0); /* piece 0 */
    fall_delay_intercept (1,0.0); /* piece 1 */
    fall_delay_intercept (2,-1.0); /* piece 2 */
    fall_pin_resistance (0,0.25); /* piece 0 */
    fall_pin_resistance (1,0.50); /* piece 1 */
    fall_pin_resistance (2,1.00); /* piece 2 */
}
timing (){
    related_pin : "CLK" ;
    timing_type : rising_edge;
    intrinsic_rise : 1.40 ;
    intrinsic_fall : 1.91 ;
    rise_pin_resistance (0,0.25); /* piece 0 */
    rise_pin_resistance (1,0.50); /* piece 1 */
    rise_pin_resistance (2,1.00); /* piece 2 */
    fall_pin_resistance (0,0.15); /* piece 0 */
    fall_pin_resistance (1,0.40); /* piece 1 */
    fall_pin_resistance (2,0.90); /* piece 2 */
}
pin ( QN ){
    direction : output ;
    function : "IQN" ;
    timing (){
        related_pin : "PRE" ;
        timing_type : clear ;
        timing_sense : negative_unate ;
        intrinsic_fall : 1.87 ;
    }
}

```

```

    fall_delay_intercept (0,1.0); /* piece 0 */
    fall_delay_intercept (1,0.0); /* piece 1 */
    fall_delay_intercept (2,-1.0); /* piece 2 */
    fall_pin_resistance (0,0.25); /* piece 0 */
    fall_pin_resistance (1,0.50); /* piece 1 */
    fall_pin_resistance (2,1.00); /* piece 2 */
}
timing (){
    related_pin : "CLR" ;
    timing_type : preset ;
    timing_sense : positive_unate ;
    intrinsic_rise : 0.68 ;
    rise_delay_intercept (0,0.054); /* piece 0 */
    rise_delay_intercept (1,0.0); /* piece 1 */
    rise_delay_intercept (2,-0.062); /* piece 2 */
    rise_pin_resistance (0,0.25); /* piece 0 */
    rise_pin_resistance(1,0.50); /* piece 1 */
    rise_pin_resistance (2,1.00); /* piece 2 */
}
timing (){
    related_pin : "CLK" ;
    timing_type : rising_edge;
    intrinsic_rise : 2.37 ;
    intrinsic_fall : 2.51 ;
    rise_pin_resistance (0,0.25); /* piece 0 */
    rise_pin_resistance (1,0.50); /* piece 1 */
    rise_pin_resistance (2,1.00); /* piece 2 */
    fall_pin_resistance (0,0.15); /* piece 0 */
    fall_pin_resistance (1,0.40); /* piece 1 */
    fall_pin_resistance (2,0.90); /* piece 2 */
}
}
}

```

---

## CMOS Nonlinear Delay Model

In the nonlinear library description in [Example 4-16](#), pin D contains setup and hold constraints; pins Q and QN contain preset, clear, and edge-sensitive arcs.

### *Example 4-16 D Flip-Flop Description (CMOS Nonlinear Delay Model)*

```
library (NLDM) {
    date : "January 14, 2002";
    revision : 2000.01;
    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 4-17](#), pin D contains setup and hold constraints; pins Q and QN contain preset, clear, and edge-sensitive arcs.

### *Example 4-17 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 */

```

---

## Clock Insertion Delay Example

```
library( vendor_a ) {
    /* 1. Use delay polynomial to mix both lookup table and polynomials*/
    delay_model :polynomial;
    /* 2. 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");
    }
    /* 3. Define library-level poly_template with only one variable*/
    poly_template(poly_template) {
        variables(input_net_transition);
        variable_1_range (0.0, 2.0);
    }
    /* 4. 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");
                }
                rise_transition(poly_template) {
                    orders("2");
                    coefs("0.1, 0.2, 0.3");
                }
                fall_transition(poly_template) {
                    orders("2");
                    coefs("0.4, 0.5, 0.6");
                }
            }
            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");
                }
            }
        }
    }
}
```

```

        rise_transition(poly_template) {
            orders("2");
            coefs("0.2, 0.3, 0.4");
        }
        fall_transition(poly_template) {
            orders("2");
            coefs("0.5, 0.6, 0.7");
        }
    }
}
...
}

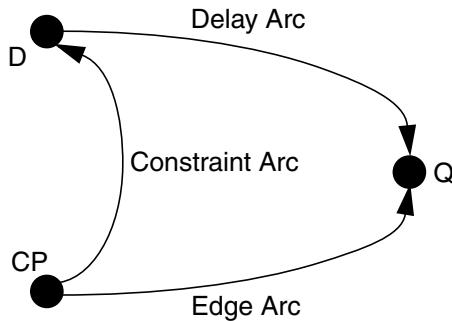
```

## 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 4-21](#) shows a transparent latch timing model.

*Figure 4-21 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 : value_Boolean ;
        }
    }
}

```

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 “qtm.”

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 Library Data

Use the timing data checker to screen for errors in your library timing data. To invoke the screener:

```
lc_shell> enable_timing_noise_signoff = 1
lc_shell> read_lib <library_name> -signoff_screening
```

Cell delay and output transition time are represented in library files as lookup tables. It is assumed that cell delay is based on input transition time and output load. The vendor runs various SPICE simulations at different slew rates and different output capacitance values to accumulate delay and transition time values. These values are then placed in four lookup tables that are indexed by input transition time and output load. The four tables for each timing arc are rise cell delay, fall cell delay, rise output transition, and fall output transition. The values for cell delay and transition time are then looked up using the input transition time and output capacitance numbers.

The screener warns you of the following issues, all of which can potentially cause timing inaccuracies:

- Checks non-monotonous delay and skew arcs in the library

When capacitance increases, the cell delay should increase. If the cell delay in the library does not increase with increasing capacitance, this indicates that the library was not characterized

properly, and timing calculation suffers as a result. The library screener therefore warns you when your library delay arcs are not increasing with increasing capacitance.

- Screens incorrect design rules in library (potential table extrapolations)

Only a limited number of points can be simulated in SPICE for each cell. When an input transition time and output load index, a cell delay, or output transition is between points, interpolation is performed. In general, there is a slight error between the delay and slew number derived by interpolation and the actual value, but this error is +- 1 percent. When a value is indexed that falls outside of the table's range, extrapolation is performed to determine the delay and slew. In general, the error that results is much larger than interpolation, and there can be a large discrepancy between calculated delay and slew and the actual values. The screener verifies that the default design rules, maximum transition, and min/max capacitance are set correctly so the delay calculator does not use cell values outside of the characterization range.

- Checks table size recommendation

Provide at least a 7 by 7 look-up table for delay and slew. In general, the more points in the table, the more accurate the timing calculation will be.

- Checks whether table values have enough significant digits

The slew/delay values in the table must have at least 4 significant digits or 0.1 picoseconds (ps) if the library time is in ns and a 0.18-micron technology library. This guideline becomes more stringent for smaller processes.

- Checks whether the starting index point in tables is 0 or very close to 0

In regions where the input transition time is small and the output load is small, delay and slew tend to increase nonlinearly with increasing input transition and output load. To avoid errors at these nonlinear regions, specify a starting index point of 0 or very close to 0.

---

## 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 in order to describe a collection of waveforms under 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). Note that 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.

Note that 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 and rise- 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 .db file.

## **Screening Rules**

Library Compiler checks the `normalized_driver_waveform` table data and issues a warning message if the following conditions are not met:

- If multiple driver waveform tables are defined, each `driver_waveform_name` value should be different. Otherwise, only the last name is retained and a warning message is issued. Other tables with the same driver waveform name are ignored and are not stored in the .db file.
- Normalized voltage values in `index_2` should be monotonically increasing in the range of [0, 1]. If any value goes beyond this range, an error message is issued. If there is not enough start and end points in the waveform (at least one point between 0 ~ 0.05 and 0.95 ~ 1.0), a warning message is issued.
- The time values for a specific input slew should be monotonically increasing and not overlapping.

---

## **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, Library Compiler 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 4-155.

### **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 4-156.

---

## Screening Rules

Library Compiler checks the driver waveform data integrity for the `normalized_driver_waveform` table and driver waveform attributes specified at the cell level and issues a warning message if the following conditions occur:

- There are not enough start and end points in the waveform (at least one point between 0 ~ 0.05 and 0.95 ~ 1.0).

- More than one common driver waveform table (a table without a `driver_waveform_name`) is defined in the library. In this case, only the last table is retained. The other tables are ignored and not stored in the .db file.
- More than one `normalized_driver_waveform` table with the same name is defined. In this case, only the last table is retained. The other tables are ignored and not stored in the .db file.
- If multiple `normalized_driver_waveform` templates are specified without the `driver_waveform_name` attribute to identify the `normalized_driver_waveform` template.

Library Compiler checks the driver waveform data integrity for the `normalized_driver_waveform` table and driver waveform attributes specified at the cell level and issues an error message if the following conditions occur:

- The `normalized_driver_waveform` table uses a nonexistent `lu_table_template` or uses the wrong `lu_table_template` where the `variable_1` and `variable_2` values are not equal to the `input_net_transition` and `normalized_voltage`.
- The `index_2` values (the normalized voltage) are not in monotonically increasing order, in the range of [0, 1].
- Any `index_2` value goes beyond the range of [0, 1].
- For a specific input slew, the time values are not in monotonically increasing order, or there is overlapping between the time values.
- The `driver_waveform_name` referred by the `driver_waveform`, `driver_waveform_rise`, and `driver_waveform_fall` attributes is not predefined.

---

## 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). In order 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 5, “Composite Current Source Modeling.”](#)

The following sections describe the pin sensitization condition information details used to characterize the timing data. The syntax pre-declares 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 will be 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_1out) {
    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_1out;
    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_1out sensitization
                           group. Wave form interpretation of the 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_1out 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, Library Compiler 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 in order 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).

---

## 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*/

```

## Library Screener

Library Compiler issues an error when the following situations occur in the sensitization group:

- No `pin_names` attribute is defined.
- The `pin_names` attribute is not declared first in the group.
- There is a duplicate pin name in the `pin_names` attribute.
- No `vector` attribute is defined.
- The `vector_id` value in the `vector` attribute is less than zero.

- The `vector_id` value in the `vector` attribute is duplicated in the sensitization group.
- The number of elements in `vector_string` is different from the number of arguments in the `pin_names` attribute.

Library Compiler issues an error when the following situations occur in a cell group specified with sensitization information:

- The number of pins in the `pin_name_map` attribute is different from the number of arguments in the sensitization master's pin names.
- The pin name in the `pin_name_map` attribute is an undefined pin name in the current cell.
- There is a duplicate pin name in the `pin_name_map` attribute.
- If only the `sensitization_master` attribute is defined, but no `pin_name_map` attribute is defined, and the pin name in the `pin_names` attribute is undefined in the current cell.

Library Compiler issues an error when the following situations occur when reading the timing group in a cell with sensitization information:

- There are `wave_rise` and `wave_fall` attributes found in the timing group but no `sensitization_master` attribute is defined either in the timing arc or in the cell.
- The pin name for `wave_rise` and `wave_fall` is undefined in the current cell. The priority for getting pin names for `wave_rise` and `wave_fall` in the timing arc is determined by the following order:
  1. The `pin_name_map` attribute in the current timing arc.
  2. The pin names of the `sensitization_master` attribute in the current timing arc.

3. The `pin_name_map` definition in the current cell.
  4. The pin names of the `sensitization_master` attribute in the current cell.
- The `pin_name_map` attribute is defined in the timing group and the number of pins in the attribute is different from that in the sensitization master. When you select a sensitization master group reference, higher priority is given to the `sensitization_master` group defined under the current timing group rather than the `sensitization_master` group specified at the cell level.
  - Any vector ID in `wave_rise` and `wave_fall` is not defined in the master sensitization group.
  - The vector ID number in `wave_rise` and `wave_fall` is less than 2.
  - The `wave_rise_sampling_index` and `wave_fall_sampling_index` value is less than zero or greater than or equal to the `wave_rise` and `wave_fall` size.
  - The number of arguments in `wave_rise_timing_interval` and `wave_fall_timing_interval` is less than 1 or greater than or equal to the `wave_rise` and `wave_fall` size.

## 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 */

```

## 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 */

```

## Using the LibChecker Utility

LibChecker is a free-standing utility that checks a technology library in Liberty (.lib) format for the completeness, consistency, and accuracy of its cell models, especially composite current source

(CCS) models. The LibChecker utility is typically used to verify the integrity of new or changed library cell models that have been created or modified by a characterization tool.

For more information about the LibChecker utility, see the *LibChecker Utility User Guide*.



# 5

## Composite Current Source Modeling

---

This chapter provides an overview of composite current source modeling (CCS). It covers the new 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
- Screening Checks
- Example of Composite Current Source Driver and Receiver Model

---

## **Modeling Cells With Composite Current Source Information**

Existing driver models can deliver acceptable accuracy when output waveforms are mostly linear and interconnect resistance is low.

However, as integrated circuit technology advances to nanometer geometries, waveforms can become highly nonlinear and interconnect delay can become a concern. At the same time, faster circuit speeds require more accurate delay calculation.

Composite current source modeling supports additional driver model complexity by using a time- and voltage- dependent current source with essentially an infinite drive resistance. The new driver model achieves high accuracy by not modeling the transistor behavior. Instead, it maps the arbitrary transistor behavior for lumped loads to that for an arbitrary detailed parasitic network.

The composite current source model improves the receiver model accuracy because the input capacitance of a receiver is dynamically adjusted during the transition by using two capacitance values. The driver model can be used with or without the receiver model.

---

## **Representing Composite Current Source Driver Information**

In the Liberty syntax, using the composite current source model, you can represent nonlinear delay information at the pin level by specifying a current lookup table at the timing group level that is dependent upon input slew and output load.

---

## Composite Current Source Lookup Table Model

You can represent composite current source driver models in your libraries by using lookup tables. To define your lookup tables, use the following groups and attributes:

- `output_current_template` group in the `library` group level
- `output_current_rise` and `output_current_fall` groups in the `timing` group level

---

### 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.

#### 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

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.

## 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.

## 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**

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.

### **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");
            ...
        }
    }
}

```

The following checks for the composite current source driver model are made:

- Library Compiler ensures that the variable lists of the `output_current_template` referred to in the `vector` group include only `input_net_transition`, `total_output_net_capacitance`, and `time`.
- Because the `index_1`, `index_2`, and `index_3` variables are required in the lookup tables that reference `output_current_template`, they are optional in `output_current_template`.

- For half-unate sequential timing arcs, if `timing_type` is `rising_edge` or `falling_edge` and `timing_sense` is `positive_unate`, only `output_current_rise` is required.

If `timing_type` is `rising_edge` or `falling_edge` and `timing_sense` is `negative_unate`, only `output_current_fall` is required.

For all other timing arcs, `output_current_rise` and `output_current_fall` have to be defined in a pair. Otherwise Library Compiler quits with an error message.

- The `output_current_rise` and `output_current_fall` groups can coexist with other delay tables in the timing group. It is legal to define only `output_current_rise` or `output_current_fall` inside the timing group for half-unate arcs.
- If more than one `output_current_rise` or `output_current_fall` group is defined in a timing group, only the last group defined is used.
- Inside each `output_current_rise` or `output_current_fall` group, `reference_time` must be the same for each load of an `input_net_transition` value.
- In the `vector` group, the indexes for `input_net_transition` and `total_output_net_capacitance` can define only one value each. Library Compiler quits with an error if `output_current_rise` and `output_current_fall` vectors do not fully fill the entire `input_net_transition` and `total_output_net_capacitance` two-dimensional table.

For every `input_net_transition` and `total_output_net_capacitance` pair, there must be a current time vector. The same value of load points should be specified for each slew. Otherwise, Library Compiler issues an error.

---

## Representing Composite Current Source Receiver Information

Composite current source receiver modeling must be used in conjunction with composite current source driver modeling. This model improves the receiver model accuracy.

With source driver modeling, the capacitance is adjusted at the delay threshold. The capacitances used to model the receiver are dependent on input slew and output load.

---

## Composite Current Source Lookup Table Model

Library information for composite current source receiver modeling can be defined as follows:

- At the pin level inside the `receiver_capacitance` group
- At the timing level by using the following groups:  
`receiver_capacitance1_rise`  
`receiver_capacitance1_fall`  
`receiver_capacitance2_rise`  
`receiver_capacitance2_fall`

Values for rise and fall can be defined at the pin or timing level. The pin-level definition does not depend on output capacitance and is useful when there are no forward timing arcs.

---

## Defining the Receiver Capacitance Group at the Pin Level

You can define a `receiver_capacitance` group at the pin level.

Use the `receiver_capacitance1_rise`,  
`receiver_capacitance1_fall`,  
`receiver_capacitance2_rise`, and  
`receiver_capacitance2_fall` groups.

## 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.

### Syntax

```
library(name_id) {  
    ...  
    lu_table_template(template_name_id) {  
        variable_1: input_net_transition;  
        index_1 ("float, ..., float");  
        ...  
    }  
    ...  
}
```

### Template Variables

In the `pin` group, the table template specifying composite current source receiver models can have one variable: `variable_1`. The only valid value is `input_net_transition`.

The index values in the `index_1` attribute are a list of ascending floating-point numbers.

## Example

```
...
    lu_table_template(LTT1) {
        variable_1: input_net_transition;
        index_1("0.1, 0.2, 0.3, 0.4");
    }
```

## Defining the Lookup Table `receiver_capacitance` Group

To specify the receiver capacitance for the nonlinear table model, use the `receiver_capacitance` group within the `pin` group.

### Syntax for Pin Level

```
pin(name_id) {
    direction: input; /* or "inout" */
    receiver_capacitance() {
        receiver_capacitance1_rise(template_name_id) {
            index_1("float,..., float"); /* optional */
            values("float,..., float");
        }
        receiver_capacitance1_fall(template_name_id) {
            index_1("float,..., float"); /* optional */
            values("float,..., float");
        }
        receiver_capacitance2_rise(template_name_id)
            index_1("float,..., float"); /* optional */
            values("float,..., float");
        }
        receiver_capacitance2_fall(template_name_id)
            index_1("float,..., float"); /* optional */
            values("float,..., float");
    }
}
```

```
    }
}
```

## Template Variables

In the `pin` group, the table template specifying receiver characteristics can have one variable: `variable_1`. The only valid value is `input_net_transition`.

The `index` values in the `index_1` attribute are a list of ascending floating-point numbers.

The `values` attribute defines a list of floating-point numbers that represent the capacitance of the receiver model.

## Example for Pin Level

```
pin(A) { /* pin-based receiver model */

    direction : input;
    receiver_capacitance() {
        receiver_capacitance1_rise(LTT1) {
            values("1.0, 4.1, 2.1, 3.0");
        }
        receiver_capacitance1_fall(LTT1) {
            values("1.0, 3.2, 2.1, 4.0");
        }
        receiver_capacitance2_rise(LTT1) {
            values("1.0, 4.1, 2.1, 3.0");
        }
        receiver_capacitance2_fall(LTT1) {
            values("1.0, 3.2, 2.1, 4.0");
        }
    }
}
```

Library Compiler makes the following checks:

- In the receiver\_capacitance group, the lu\_table\_template referred to by receiver\_capacitance1\_rise, receiver\_capacitance1\_fall, receiver\_capacitance2\_rise, and receiver\_capacitance2\_fall contains only input\_net\_transition as its parameter.
- If any of receiver\_capacitance1\_rise, receiver\_capacitance1\_fall, receiver\_capacitance2\_rise, or receiver\_capacitance2\_fall are defined in the receiver\_capacitance group, then all of them have to be defined.
- If more than one receiver\_capacitance1\_rise, receiver\_capacitance1\_fall, receiver\_capacitance2\_rise, or receiver\_capacitance2\_fall are defined in the receiver\_capacitance group, only the last one counts.
- The receiver\_capacitance group applies only to input or inout pins.
- Receiver model exclusive check: If input pin A defines the receiver\_capacitance group, then no timing arc A->Y (attached under pin Y) can define receiver\_capacitance1\_rise, receiver\_capacitance1\_fall, receiver\_capacitance2\_rise, or receiver\_capacitance2\_fall.

That is, either a pin-based or arc-based receiver\_capacitance group can be specified, but not both.

---

## **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 only the receiver\_capacitance1\_rise, receiver\_capacitance1\_fall, receiver\_capacitance2\_rise, and receiver\_capacitance2\_fall groups.

## **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.

### **Syntax**

```
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

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.

## 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");
    }
```

## Defining the Lookup Table receiver\_capacitance Groups

To specify the receiver capacitance for the nonlinear table model, use the receiver\_capacitance1\_rise, receiver\_capacitance1\_fall, receiver\_capacitance2\_rise receiver\_capacitance2\_fall groups within the timing group.

## Syntax for Timing Level

```
direction: output; /* or "inout" */
timing () {
    ...
    receiver_capacitance1_rise(template_nameid) {
        index_1("float,..., float"); /* optional */
        index_2("float,..., float"); /* optional */
        values("float,..., float");
    }
}
```

```

receiver_capacitance1_fall(template_name_id) {
    index_1("float,..., float"); /* optional */
    index_2("float,..., float"); /* optional */
    values("float,..., float");
}

receiver_capacitance2_rise(template_name_id) {

    index_1("float,..., float"); /* optional */
    index_2("float,..., float"); /* optional */
    values("float,..., float");
}

receiver_capacitance2_fall(template_name_id) {
    index_1("float,..., float"); /* optional */
    index_2("float,..., float"); /* optional */
    values("float,..., float");
}

...
}

```

## Template Variables

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.

## Example at the timing level

```

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");
}
...
}

```

Library Compiler makes the following checks:

- In the timing group, the lu\_table\_template referred to by receiver\_capacitance1\_rise, receiver\_capacitance1\_fall, receiver\_capacitance2\_rise, and receiver\_capacitance2\_fall must contain input\_net\_transition and total\_output\_net\_capacitance as its parameters.
- If any of receiver\_capacitance1\_rise, receiver\_capacitance1\_fall, receiver\_capacitance2\_rise, or receiver\_capacitance2\_fall are defined in the timing group, then all of them have to be defined.
- If more than one receiver\_capacitance1\_rise, receiver\_capacitance1\_fall, receiver\_capacitance2\_rise, or receiver\_capacitance2\_fall are defined in the timing group, only the last one counts.

---

## Screening Checks

Library Compiler checks the following composite current source and issues a warning or error message when it encounters potential problems:

- Current Polarity

The polarity of the current must be positive for rising outputs and negative for falling outputs.

The library syntax is such that the floating-point numbers associated with the `values` attribute in each current vector should all be positive if the vector is in the `output_current_rise` group or negative if the vector is in the `output_current_fall` group.

- Full Voltage Swing

Library Compiler issues a warning if the final voltage is not close to the final rail voltage. Calculate this margin as a percentage of the voltage swing. For example,

$$VmMargin = 0.05 * (VDD \mid VSS)$$

For the `output_current_rise` vector,

```
if ( fabs(Vfinal-VDD) > Vmargin ) { <LC issues warning> }
```

Similarly, for the `output_current_fall` vector,

```
if ( fabs(Vfinal-VSS) > Vmargin ) { <LC issues warning> }
```

Library Compiler issues an error message if the final voltage for a current vector does not reach beyond the second slew threshold; that is, the maximum of the slew thresholds for the `output_current_rise` vector, and the minimum of the slew thresholds for the `output_current_fall` vector. For example,

For `output_current_rise`,

```
max = MAX(slew_lower_threshold_pct_rise,
           slew_upper_threshold_pct_rise,
           output_threshold_pct_rise)
Verr = VSS + max * (VDD-VSS) * 0.01
if ( Vfinal < Verr ) {
    <LC issues error> }
```

Similarly for `output_current_fall`

```
min = MIN(slew_lower_threshold_pct_fall,
           slew_upper_threshold_pct_fall,
           output_threshold_pct_fall)
Verr = VSS + min * (VDD-VSS) * 0.01 if ( Vfinal > Verr )
{ <LC issues
error> }
```

Note that in the current format, VSS is always 0, and VDD is determined as follows for the output pin containing the `output_current_rise` and `output_current_fall` vectors:

- If the `output_signal_level` attribute exists for the pin, the `power_rail` value referenced by the `output_signal_level` attribute is applied. Otherwise, the value of the `voltage` attribute as defined in the default operating condition of the library is applied.
- For a vector of `output_current_rise` with time or current values ( $T_1, I_1, \dots, T_n, I_n$ ):

$$V_{final} = VSS + (0.5/C_{out}) * (I_2+I_1) * (T_2-T_1) + \dots + (0.5/C_{out}) * (I_n+I_{n-1}) * (T_n-T_{n-1})$$

- For a vector of `output_current_fall` with time or current values ( $T_1, I_1, \dots, T_n, I_n$ ):

```
Vfinal = VDD + (0.5/Cout) * (I2+I1) * (T2-T1) + ... + (0.5/Cout) * (In+In-1) * (Tn-Tn-1)
```

- Reference times  
If the reference times are zero or negative, Library Compiler issues an error message.
- Zero current values  
If the current vectors contain 0 values, Library Compiler issues an error message.
- Peak current  
If either the first or last current value in a current vector is the peak of the current waveform, Library Compiler issues an error message.
- Half-unate receiver modeling  
If the timing arc does not have the full receiver modeling information, Library Compiler issues a warning.
- Non-increasing current vector values  
Library Compiler checks if the time index values or the currents in the values attribute of each current vector are non-increasing.  
Library Compiler issues an error message for non-increasing timing indexes. If the adjacent currents are identical, Library Compiler issues a warning.

Note:

For scientific notation, Library Compiler accepts only the IEEE single-precision float value from the .lib files. The value can be represented by s#.#####E##, where s is +/- and # is an integer between 0 and 9.

- Significant digits without scientific notation

Library Compiler checks whether the current vector and receiver capacitance tables have at least four significant digits. If they do not, Library Compiler issues an informational message. For example, “Accuracy might be affected by not using enough significant digits.”

---

## Example of Composite Current Source Driver and Receiver Model

[Example 5-1](#) is an example of composite current source driver and receiver model syntax.

### *Example 5-1 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(DFF) {
    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 (DFF)*/
    . . .
    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 */

```



# 6

## 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 Support](#)
- [Variation-Aware Timing Modeling Support](#)

---

## **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. The new driver model achieves high accuracy by not modeling the transistor behavior. Instead, it maps the arbitrary transistor behavior for lumped loads to that for an arbitrary detailed parasitic network.

The composite current source model improves the receiver model accuracy because the input capacitance of a receiver is dynamically adjusted during the transition by using two capacitance values. The driver model can be used with or without the receiver model.

---

## **Compact CCS Timing Model Support**

Existing CCS timing driver modeling syntax requires that you describe each CCS driver switching current waveform by adaptively sampling data points. Often, a large amount of data is required to represent the library to model these switching curves. As the number of timing arcs in a standard cell library grows, the CCS timing library size can become very large.

This section describes the syntax of a compact modeling format 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 CCS timing library is efficiently compressed.

The topics in the following sections include:

- Using the `ccs_cet` utility to convert conventional CCS modeling information into compact CCS data format.
  - 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 Library Compiler.
- 

## Using the `ccs_cet` Utility

The `ccs_cet` utility allows you to convert conventional CCS modeling information into compact CCS data format. The compact CCS format offers the same high accuracy as conventional CCS data but uses much less space in library data files.

### Syntax

The compact CCS utility command syntax is as follows:

```
ccs_cet [-expand] -outputlib lib_name input_lib_name
```

You must specify the name of the output library produced by the command and the name of the input library containing the original (not compacted) CCS modeling information. Both files are in Liberty (.lib) format.

The `-expand` option performs the opposite function; it expands a library containing compact CCS modeling information into a library containing the original (not compacted) CCS modeling information.

---

## 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 6-1](#) 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 6-1 Inverter Cell Rise Transition With Existing CCS Format

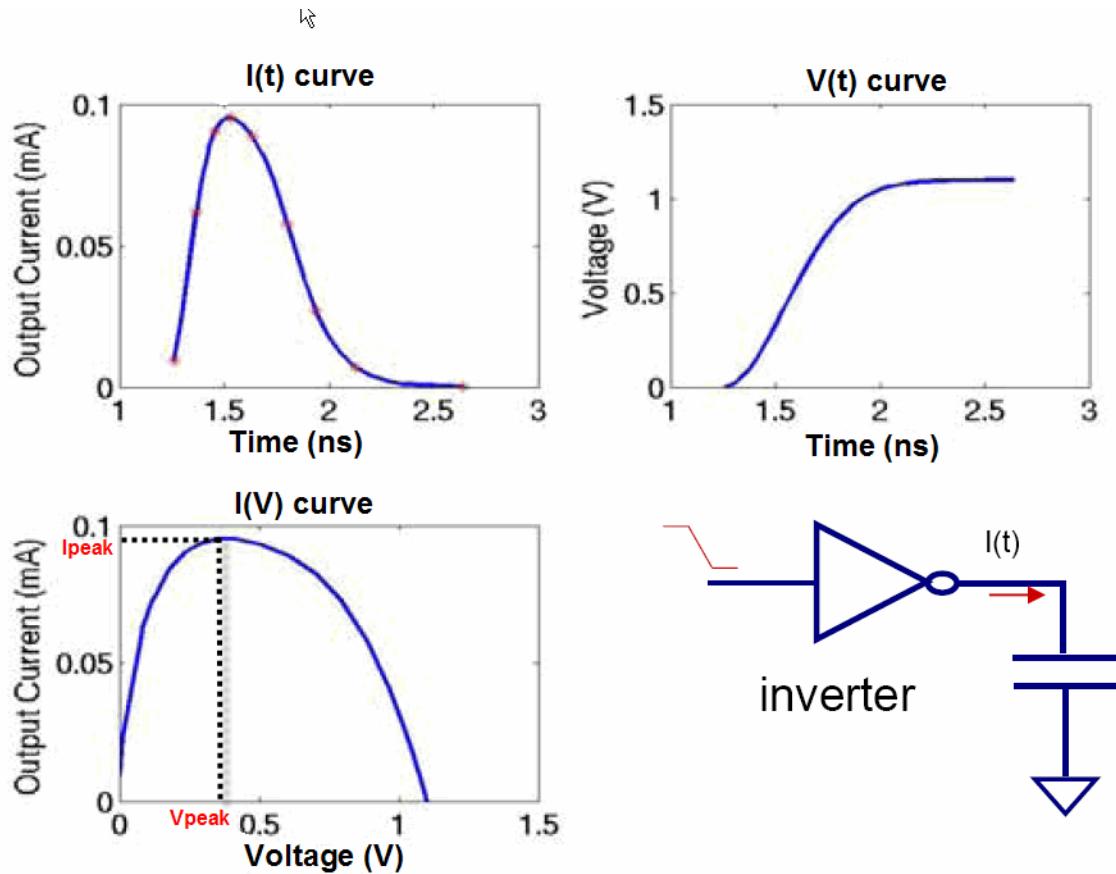


Figure 6-2 shows the mapping of  $I(V)$  curves to base curves. The  $I(V)$  curve is split into two halves at the peak, and an eleven point normalized base curve in the base curve database is selected to exactly match each half curve.

Only six parameters are required to model this inverter switching curve, reducing the storage cost by three times. The six parameters are as follows:

$I_{init}$

Switching current value at the starting point.

I<sub>peak</sub>

Peak switching current value.

V<sub>peak</sub>

Voltage value when current reaches peak value.

T<sub>peak</sub>

Time when current reaches peak value.

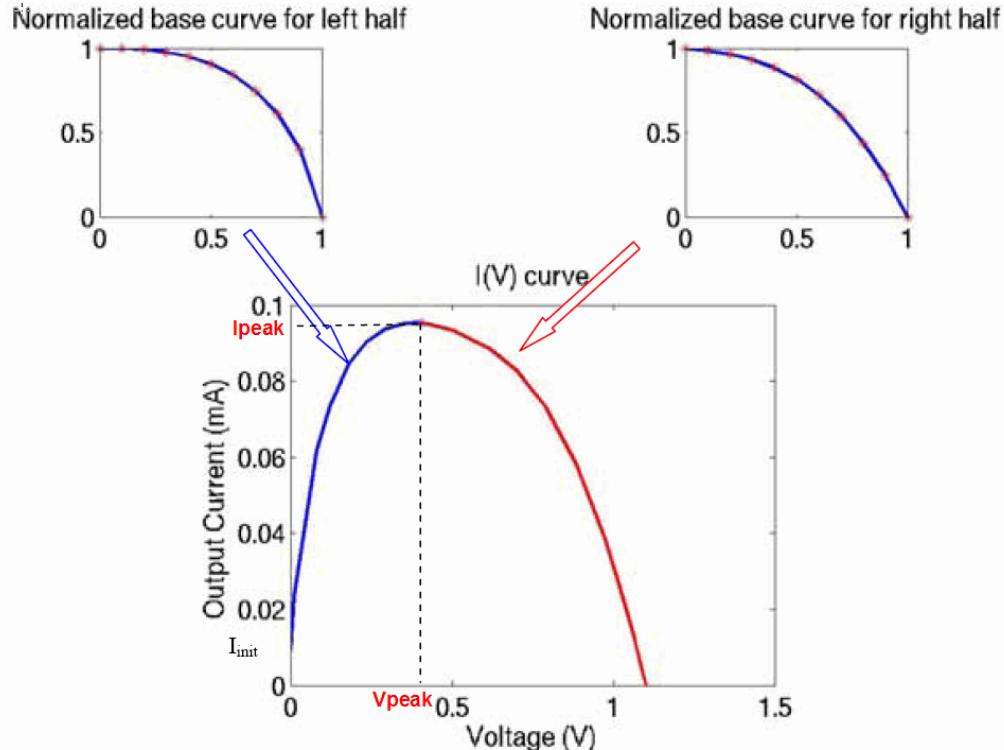
Left id

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 6-2 Using Base Curves to Simplify I(V) Curve Modeling



---

## Compact CCS Timing Model Syntax

In [Figure 6-2](#), 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 ("..."..., "..."\n
                    "...", "..."\n
                    "...", "..."\n
                    "...", "...")
        }/*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` is allowed for each `base_curves` group. See [Figure 6-2](#) 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 6-2](#)) is composed of one `curve_x` and one `curve_y`. You should define the `curve_x` base curve before `curve_y` for better readability and easier implementation.

There are two data sections in `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 valid string value for this variable.

### **index\_1 and index\_2 Attributes**

These are required attributes. They define sample `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_curve` 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, Library Compiler issues a compilation error.

Library Compiler 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\_{risefall} 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. Library Compiler 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`, Library Compiler does not check for other data in the `values` group. These six parameters are stored in the database as is.

---

## **Library Compiler Screener**

For `compact_lut_template`, Library Compiler issues a compilation error message when the following conditions are not satisfied:

- The `index_3` value is composed of string values for compact CCS timing data from the `base_curve_type` attribute in `base_curves_group`. The `index_3` value must contain at least these six strings: `init_current`, `peak_current`, `peak_voltage`, `peak_time`, `left_id`, `right_id`. After these six parameters are specified, you are allowed to specify more parameters.
- The `base_curves_group` is required in the template and its value should be a name of predefined `base_curve` group.

- All three variables and their indexes should be defined with right values and `index_1` and `index_2` values are non-negative float numbers.

For `base_curves`, Library Compiler issues a compilation error when the following conditions are not satisfied:

- `base_curve_type` must be specified, with value of `ccs_timing_half_curve`.
- Only one `base_curve` group with `ccs_timing_half_curve` is currently allowed in one library.
- Must have only one `curve_x` defined in the group. `curve_x` must be defined before `curve_y`. Values in the array should be between 0 and 1 and increase monotonically.
- The number of data values in `curve_x` and `curve_y` must be greater than or equal to three, when you specify `base_curve_type : ccs_timing_half_curve`.
- The number of data values in `curve_x` should match that in `curve_y` without taking `curve_id` into account.
- Each `curve_id` in `curve_y` should be greater than 0 and unique in the `base_curves` group.

When reading CCS timing data, Library Compiler issues a compilation error if the following situations are not satisfied:

- Only one representation of a CCS timing driver model can exist in the library, either regular CCS timing model `output_current_{rise|fall}` or compact CCS timing model `compact_ccs_{rise|fall}`.

- In the `compact_ccs_{rise|fall}` group, at the library level, the template name should be a predefined `compact_lut_template`.
- The `init_current` value should be greater than or equal to zero in `compact_ccs_rise` group and less than or equal to zero in `compact_ccs_fall` group.
- The `peak_current` value should be greater than zero in the `compact_ccs_rise` group and less than zero in the `compact_ccs_fall` group
- The `peak_voltage` should be between VSS and VDD
- `left_id` and `right_id` should be integers, and they should be predefined in the `base_cuves` group

Library Compiler issues a warning when the following is not satisfied:

- The `max_capacitance` design rule constraint of the output pin should be less than or equal to the value represented in the `compact_ccs_{rise|fall}` group.

## 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 (ctbct1){
    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: "ctbct1";
}
...

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*/

```

---

## Variation-Aware Timing Modeling Support

As process technologies scale to nanometer geometries, it is crucial to build variation-based cell models to solve uncertainties attributed to the variability in the device and interconnect. The CCS timing approach addresses the effects of nanometer processes by enabling advanced driver and receiver modeling.

This modeling capability supports variation parameters and is an extension of compact CCS timing driver modeling. You can even apply variation parameter models to CCS timing models. For more information about compact CCS timing driver modeling, see [“Compact CCS Timing Model Support” on page 6-2](#).

These timing models employ a single current-based behavior that enables the concurrent analysis and optimization of timing issues. The result is a complete open-source current based modeling solution that reduces design margins and speeds design closure.

Process variation is modeled in static timing analysis tools to improve parametric yield and to control the design for corner-based analysis. This section describes the extension for variation-aware timing modeling using the existing CCS syntax.

The following features are covered in detail in the following sections:

- Variation-aware modeling for compact CCS timing driver
- Variation-aware modeling for CCS timing receiver
- Variation-aware modeling for regular or interdependent timing constraint

The amount of data can be reduced by using the compact CCS timing syntax. Without compacting, variation parameter models require more library data storage. You are expected to be familiar with the compact CCS syntax before reading this section.

---

## Variation-Aware Compact CCS Timing Driver Model

This format supports variation parameters. It is an extension of a compact CCS timing driver modeling. The timing\_based\_variation groups specified in the timing group can represent variation-aware CCS driver information in a compact format. The syntax is as follows:

```
library (<library_name>) {  
    ...  
    base_curves (<base_curves_name>) {  
        base_curve_type: <enum (ccs_timing_half_curve)>;  
        curve_x ("<float>, ...");  
        curve_y (<integer>, "<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; ...}  
            va_parameters(<string> , ... );  
        ...  
        timing() {  
            compact_ccs_rise(<template_name>) { ... }  
            compact_ccs_fall(<template_name>) { ... }  
            timing_based_variation() {  
                va_parameters(<string> , ... );  
                nominal_va_values(<float>, ...);  
                va_compact_ccs_rise(<template_name>) {  
                    va_values(<float>, ...);  
                    values("...,<float>, ...,<integer>, ...", ...);  
                ... }  
            ...  
            va_compact_ccs_fall(<template_name>) { ... }  
        }  
}
```

```
    ... } /* end of timing_based_variation */
    ... } /* end of timing group */
...} /* end of library group */
```

## The timing\_based\_variation Group

This group specifies rising and falling output transitions for variation parameters. The rising and falling output transitions are specified in `va_compact_ccs_rise` and `va_compact_ccs_fall` respectively.

- The `va_compact_ccs_rise` group is required only if a `compact_ccs_rise` group exists within a timing group.
- The `va_compact_ccs_fall` group is required only if a `compact_ccs_fall` group exists within a timing group.

### **va\_parameters Attribute**

This attribute specifies a list of variation parameters.

- One or more variation parameters are allowed.
- Variation parameters are represented by a string.
- Values in `va_parameters` must be unique.
- The `va_parameters` must be defined before being referenced by `nominal_va_values` and `va_values`.

The `va_parameters` attribute can be specified within a variation group or within a library level. `timing_based_variation` can be specified within a timing group only, and `pin_based_variation` can be specified within a pin group only. None of these can be specified within a library group.

- If `va_parameters` is specified at the library level, all cells under the library default to the same variation parameters.
- If `va_parameters` is defined in the `variation group`, all `va_values` and `nominal_va_values` under the same variation group shall refer to this `va_parameters`.

The attribute values can be user-defined or predefined parameters. For more information, see [Example 6-1 on page 6-32](#).

The parameters defined in `default_operating_conditions` are process, temperature, and voltage. The voltage names are defined by using the `voltage_map` complex attribute. For more information see “[voltage\\_map Attribute](#)” on page 2-5.

You can use the following predefined parameters:

- For the parameters defined in `operating_conditions`, if `voltage_map` is defined, and you specify these attributes as values of `va_parameters`, Library Compiler takes these attributes as user-defined variables.
- For the parameters defined in `operating_conditions`, if there is no `voltage_map` attribute at library level, and you specify these attributes as values of `va_parameters`, Library Compiler takes these attributes as predefined parameters.

### **`nominal_va_values` Attribute**

This attribute characterizes nominal values of all variation parameters.

- It is required for every `timing_based_variation group`.
- The value of this attribute has a 1-to-1 mapping to the corresponding `va_parameters`.

- If a nominal compact CCS driver model group and a variation-aware compact CCS driver model group are defined under the same timing group, the nominal values are applied to the nominal compact CCS driver model group.

## The `va_compact_ccs_rise` and `va_compact_ccs_fall` Groups

The `va_compact_ccs_rise` and `va_compact_ccs_fall` groups specify characterization corners with variation value parameters. All screener checks applicable to the compact CCS format apply to attributes under this group.

- These groups can be specified under different `timing_based_variation` groups if they cannot share the same `va_parameters`.
- You should characterize two corners at each side of the nominal value of all variation parameters as specified in `va_parameters`. When corners are characterized for one of the parameters, all other variations are assumed to be nominal values. Therefore, a `timing_based_variation` group with  $N$  variation parameters requires exactly  $2N$  characterization corners. For an example, see [Example 6-2](#).
- All variation-aware compact CCS driver model groups inside the `timing_based_variation` share the same `va_parameters` attribute.

### `va_values` Attribute

The `va_values` attribute specifies values of each variation parameter for all corners characterized in the variation-aware compact CCS driver model groups.

- Required for the variation-aware compact CCS driver model groups.
- The value of this attribute has a 1-to-1 mapping to the corresponding `va_parameters`.

For an example that shows how to specify `va_values` with three variation parameters, see [Example 6-3 on page 6-34](#). In the example, the first variation parameter has a nominal value of 0.50, the second parameter has a nominal value of 1.0, and the third parameter has a nominal value of 2.0. All parameters have a variation range of -10% to +10%.

### **values Attribute**

The `values` attribute follows the same rules as the nominal compact CCS driver model groups with the following exceptions:

- The `left_id` and `right_id` are optional.
- The `left_id` and `right_id` values must be used together. They can either be omitted or defined together in the `compact_lut_template`.
- If `left_id` and `right_id` are not defined in the variation-aware compact CCS driver model group, they default to the values defined in the nominal compact CCS driver model group.

### **Library Compiler Screener**

Library Compiler checks the variation-aware syntax using the compact CCS timing syntax for driver modeling and issues an error message if the following conditions are not satisfied:

- The `va_values` attribute is required in the variation-aware compact CCS driver model groups.

- The `va_compact_ccs_rise` attribute can be specified in a `timing_based_variation` group only if `compact_ccs_rise` is defined under the same timing group as where the `timing_based_variation` group is defined.

This same rule is applied to `va_compact_ccs_fall` and `compact_ccs_fall` groups.

- The `peak_voltage` value cannot be less than the smallest ground voltage of a cell or greater than the largest power voltage specified in a cell. If `va_values` contains predefined power or ground values, the following additional checking is needed:

The `peak_voltage` value cannot be less than the smallest ground values or greater than the largest power values defined in the `va_values` attribute. For an example, see [Example 6-4](#).

- The `left_id` and `right_id` can either be omitted or specified together under the `values` attribute. An error is issued if only one of them is defined in a `values` attribute.
- The variation-aware compact CCS driver model groups must refer to a valid `compact_lut_template`.
- The size of tables in the variation-aware compact CCS driver model group must be at least either 2x2x4 if `left_id` and `right_id` are not presented.

## **timing\_based\_variation and pin\_based\_variation Groups**

These groups represent variation-aware receiver capacitance information under the timing and pin groups.

- If receiver\_capacitance group exists in pin group, variation-aware CCS receiver model groups are required in pin\_based\_variation.
- If nominal CCS receiver model groups exist in the timing group, variation-aware CCS receiver model groups are required in timing\_based\_variation.

### **va\_parameters Attribute**

The va\_parameters attribute specifies a list of variation parameters within a timing\_based\_variation or pin\_based\_variation group. See “[va\\_parameters Attribute](#)” on page 6-17 for details.

### **nominal\_va\_values Attribute**

The nominal\_va\_values attribute characterizes nominal values for all variation parameters. The following list describes the nominal\_va\_values attribute.

- The attribute is required in the timing\_based\_variation and pin\_based\_variation groups.
- The value of this attribute has one-to-one mapping to the corresponding va\_parameters attribute.
- In pin-based models, if nominal CCS receiver model and variation-aware CCS receiver model groups are defined under the same pin, the nominal values are applied to the nominal CCS receiver model groups. For an example, see [Example 6-5](#).
- In timing-based models, if the nominal compact CCS driver model group and variation-aware CCS receiver model groups are defined under the same timing group, the nominal values are applied to the nominal compact CCS driver model groups.

## **va\_receiver\_capacitance1\_rise, va\_receiver\_capacitance1\_fall, va\_receiver\_capacitance2\_rise, va\_receiver\_capacitance2\_fall Groups**

These groups specify characterization corners with variation values in `timing_based_variation` and `pin_based_variation` groups. All attributes under these groups are checked with the same library screener as nominal CCS receiver models.

- You should characterize two corners at each side of the nominal value of all variation parameters specified in `va_parameters`.  
  
When corners are characterized for one of parameters, all other variations are assumed to be nominal value. Therefore, for a `timing_based_variation` group with  $N$  variation parameters, exactly  $2N$  characterization corner groups are required. This same rule applies to `pin_based_variation`.
- All variation-aware CCS receiver model groups in `timing_based_variation` or `pin_based_variation` group share the same `va_parameters` attribute.

## **va\_values Attribute**

Specifies values of each variation parameter for all corners characterized in variation-aware CCS receiver model groups.

- The attribute is required for variation-aware CCS receiver model groups.
- The value of this attribute has 1-to-1 mapping to the corresponding `va_parameters` attribute.

## **Library Compiler Screener**

This Library Compiler checks the following variation aware syntax of the CCS timing in receiver modelling, and issues an error if any of the following are not satisfied:

- The `va_values` is required in variation-aware CCS receiver model groups.
- The variation-aware CCS receiver model groups can be specified in a `timing_based_variation` group only if the nominal CCS receiver model groups are defined within the same timing group where the `timing_based_variation` group is defined.
- The variation-aware CCS receiver model groups can be specified in a `pin_based_variation` group only if `receiver_capacitance` group is defined within the same pin group as where the `pin_based_variation` group is defined.
- The variation-aware CCS receiver model group must refer to a valid `lu_table_template`.

---

## **Variation-Aware CCS Timing Receiver Model**

The variation-aware CCS receiver model is expected to be used together with variation-aware compact CCS driver model. The `timing_based_variation` and `pin_based_variation` groups specify timing and pin groups respectively. In both groups, the variation-aware CCS receiver model groups are used to represent variation-aware CCS receiver information as defined below.

### **Syntax**

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

```

lu_table_template(<timing_based_template_name>) {
    variable_1 : input_net_transition;
    variable_2 : total_output_net_capacitance; ...}
lu_table_template(<pin_based_template_name>) {
    variable_1 : input_net_transition; ...}
va_parameters(<string> , ...);

...
pin(<pin_name>) {
    receiver_capacitance() {
        receiver_capacitance1_rise(<template_name>) { ...}
        receiver_capacitance2_rise(<template_name>) { ...}
        receiver_capacitance1_fall(<template_name>) { ...}
        receiver_capacitance2_fall(<template_name>) { ... }
    }
    pin_based_variation() {
        va_parameters(<string> , ... );
        nominal_va_values(<float>, ...);
        va_receiver_capacitance1_rise(<pin_based_template_name>) {
            va_values(<float>, ...);
            values("<float>, ...", ...);
        ...
        va_receiver_capacitance2_rise(<pin_based_template_name>) {...}
        va_receiver_capacitance1_fall(<pin_based_template_name>) {...}
        va_receiver_capacitance2_fall(<pin_based_template_name>) { ... }
        ...} /* end of pin_based_variation */
    ...
    } /* end of pin */
}

...
pin(<pin_name>) {
...
    timing() {
        receiver_capacitance1_rise(<template_name>) {...}
        receiver_capacitance2_rise(<template_name>) {...}
        receiver_capacitance1_fall(<template_name>) {...}
        receiver_capacitance2_fall(<template_name>) {...}
        timing_based_variation() {
            va_parameters(<string> , ... );
            nominal_va_values(<float>, ...);
            va_receiver_capacitance1_rise(<timing_based_template_name>) {
                va_values(<float>, ...);
                values("<float>, ...", ...);
            ...
            va_receiver_capacitance2_rise(<timing_based_template_name>) {...}
            va_receiver_capacitance1_fall(<timing_based_template_name>) {...}
            va_receiver_capacitance2_fall(<timing_based_template_name>) { ... }
            ...} /* end of timing_based_variation */
        ...
    } /*end of timing */
    } /* end of pin */
}

```

## **timing\_based\_variation and pin\_based\_variation Groups**

These groups represent variation-aware receiver capacitance information under the pin or timing group level.

- If the receiver\_capacitance group exists in the pin group, variation-aware CCS receiver model groups are required in pin\_based\_variation.
- If nominal CCS receiver model groups exist in the timing group, variation-aware CCS receiver model groups are required in timing\_based\_variation.

## **va\_parameters Complex Attribute**

This attribute specifies a list of variation parameters within timing\_based\_variation or pin\_based\_variation. See [va\\_parameters Attribute](#) for more detail.

## **nominal\_va\_values Complex Attribute**

This complex attribute specifies the nominal values of all variation parameters.

- The attribute is required for timing\_based\_variation and pin\_based\_variation groups.
- The value of this attribute has one-to-one mapping to the corresponding va\_parameters.
- In a pin-based model, if nominal CCS receiver model and variation-aware CCS receiver model groups are defined under the same pin, the nominal values are applied to nominal CCS receiver model groups. For an example, see [Example 6-5](#).

- In a timing-based model, if nominal CCS receiver model and variation-aware CCS receiver model groups are defined under the same timing group, the nominal values are applied to nominal CCS receiver model groups.

**va\_receiver\_capacitance1\_rise, va\_receiver\_capacitance1\_fall,  
va\_receiver\_capacitance2\_rise, and  
va\_receiver\_capacitance2\_fall Groups**

These groups specify characterization corners with variation values in the `timing_based_variation` and `pin_based_variation` groups. The same library screening checks apply to all attributes under these groups as with the nominal CCS receiver model groups.

- You should characterize two corners at each side of the nominal value of all variation parameters specified in `va_parameters`.

When corners are characterized for one of the parameters, all other variations are assuming to be nominal value. Therefore, for a `timing_based_variation` group with  $N$  variation parameters, exactly  $2N$  characterization corner groups are required. This rule also applies to `pin_based_variation`.

- All variation-aware CCS receiver model groups in `timing_based_variation` or `pin_based_variation` group share the same `va_parameters`.

**va\_values Attribute**

Specifies values of each variation parameter for all corners characterized in variation-aware CCS receiver model groups.

- Required for variation-aware CCS receiver model groups.
- The value of this attribute has 1-to-1 mapping to the corresponding `va_parameters`.

## **Library Compiler Screener**

Library Compiler checks the following variation-aware syntax of the receiver model, and issues an error if any of the following are not satisfied:

- The `va_values` is required in variation-aware CCS receiver model groups.
- The variation-aware CCS receiver model groups are specified in a `timing_based_variation` group only if nominal CCS receiver model groups are defined within the same timing group.
- The variation-aware CCS receiver model groups are specified in a `pin_based_variation` group only if `receiver_capacitance` group is defined within the same pin group.
- The variation-aware CCS receiver model must refer to a valid `lu_table_template`.

---

## **Variation-Aware Timing Constraint Model**

This syntax supports variation parameters. It is an extension of the timing constraint modeling. It also applies to interdependent setup and hold. The `timing_based_variation` groups specified in the timing group represent variation-aware timing constraint sensitive information, as defined in the syntax below.

### **Syntax**

```
library() {  
    ...  
    lu_table_template(<template_name>) {  
        variable_1 : <variables>;  
        variable_2 : <variables>;  
        variable_3 : <variables>; ... }  
}
```

```

va_parameters(<string> , ...);

...
timing () {
...
interdependence_id :<integer>;
rise_constraint(<template_name>) { ... }
fall_constraint(<template_name>) { ... }
timing_based_variation() {
    va_parameters(<string> , ... );
    nominal_va_values(<float>, ...);
    va_rise_constraint(<template_name>) {
        va_values(<float>, ...);
        values("<float>, ...");
    ...
    va_fall_constraint(<template_name>) { ... }
    ... } /* end of timing_based_variation */
... } /* end of timing */
... } /* end of pin */
... } /* end of library */

```

## **timing\_based\_variation Group**

The `timing_based_variation` group specifies the rise and fall timing constraints for variation parameters within a timing group. The rise and fall timing constraints are specified in the `va_rise_constraint` and `va_fall_constraint` groups respectively.

- The `va_rise_constraint` group is required only if `rise_constraint` group exists within a timing group.
- The `va_fall_constraint` group is required only if `fall_constraint` group exists within a timing group.

## **va\_parameters Complex Attribute**

This complex attribute specifies a list of variation parameters within `timing_based_variation` or `pin_based_variation`. See “[“va\\_parameters Attribute” on page 6-17](#) for details.

## **nominal\_va\_values Complex Attribute**

This complex attribute is used to specify nominal values of all variation parameters. See “[nominal\\_va\\_values Attribute](#)” on [page 6-18](#) for more information.

## **va\_rise\_constraint and va\_fall\_constraint Groups**

The `va_rise_constraint` and `va_fall_constraint` groups specify characterization corners with variation values in `timing_based_variation`. The screener checks all attributes under this group to normal timing constraint models.

- The template name refers to the `lu_table_template` group.
- Both groups can be specified under different `timing_based_variation` groups if they cannot share the same `va_parameters`.
- You are expected to characterize two corners at each side of the nominal value of all variation parameters as specified in `va_parameters`.

When corners are characterized for one parameter, all other variations are assumed to be of nominal value. Therefore, for a `timing_based_variation` group with  $N$  variation parameters, exactly  $2N$  characterization corners are required.

- All `va_rise_constraint` and `va_fall_constraint` groups in the `timing_based_variation` group share the same `va_parameters`.

## **va\_values Attribute**

Specifies values of each variation parameter for all corners characterized in `va_rise_constraint` and `va_fall_constraint` groups.

- Required for `va_rise_constraint` and `va_fall_constraint` groups.
- The value of this attribute has a one-to-one mapping to the corresponding `va_parameters`.

## **Library Compiler Screener**

Library Compiler checks the following syntax of variation-aware CCS timing constraint modeling, and issues an error if the following are not satisfied:

- The `va_values` is required attribute in `va_rise_constraint` and `va_fall_constraint` groups.
- The `va_rise_constraint` and `va_fall_constraint` groups can be specified in a `timing_based_variation` group only if nominal timing constraint model groups are defined within the same timing group where the `timing_based_variation` group is defined.
- The `va_rise_constraint` and `va_fall_constraint` must refer to a valid `lu_table_template`.

## **Library Compiler Screener for Timing Models**

Library Compiler checks the compact CCS timing driver, CCS timing receiver, and timing constraint models and issues an error if the following conditions are not satisfied:

- One or more variables are expected within each `va_parameters` attribute.
- The values in `va_parameters` cannot be duplicates.

- `nominal_va_values` is required in `timing_based_variation` and `pin_based_variation` groups.
- The `va_parameters` attribute is referenced by the `nominal_va_values` and `va_values` attributes. The `va_parameters` attribute must be defined before being referenced. See [Example 6-6 on page 6-36](#).
- If predefined variation parameters are used in `va_parameters`, the mapping values in `nominal_va_values` should be the same as its defined values. See [Example 6-7 on page 6-37](#).
- The number of values in `va_values` must be identical to that of the corresponding `va_parameters`. This same rule is applied to `nominal_va_values`.
- The number of `va_compact_ccs_rise` must be exactly twice as the number of values in the corresponding `va_parameters`.  
This same rule is applied to `va_compact_ccs_fall`, variation-aware CCS receiver model, `va_rise_constraint`, and `va_fall_constraint` groups.
- Only two variational values are allowed for each parameter. One must be greater than the normal value. The other must be less than the normal value. See [Example 6-8 on page 6-37](#).
- On all but one parameter, `va_values` and `nominal_va_values` must have identical set of values.

## Variation-Aware Syntax Examples

### *Example 6-1 va\_parameters in Advanced CCS Modeling Usage*

```
library (sample) {
  ...
}
```

```

operating_conditions (typical) {
    process : 1.5 ;
    temperature : 70 ;
    voltage : 2.75 ;
    ...
}
default_operating_conditions: typical;
...
/* "temperature", "voltage" and "process" are predefined parameters, and
   "Vthr" is an user-defined parameter. */

    va_parameters(temperature, voltage, process, Vthr, ...);
...
}

library (sample) {
...
operating_conditions (typical) {
    process : 1.5 ;
    temperature : 70 ;
    voltage : 2.75 ;
    ...
}
voltage_map(VDD1, 2.75);
voltage_map(GND2, 0.2);
default_operating_conditions: typical;
...
/* "VDD1" and "GND2" are predefined parameters, and LC takes "voltage" as
   an user-defined parameter. */

    va_parameters(VDD1, GND2, voltage,...);
}

```

For information about `va_parameters`, see “[va\\_parameters Attribute](#)” on page 6-17.

### *Example 6-2 va\_compact\_ccs\_rise and va\_compact\_ccs\_fall Groups*

```

...
timing() {
    ...
compact_ccs_rise(temp) { /* nominal I-V waveform */ ...}
timing_based_variation() {
    va_parameters(<string>, ...); /* N variation parameters */
    ...
    va_compact_ccs_rise(temp) { /* 1st corner */ ...}
    ...
    va_compact_ccs_rise(temp) { /* last corner : total (2 * N) corners */ ...}
} /* end of timing_based_variation */ ...} /* end of timing */

```

...

For information about `va_compact_ccs_rise` and `va_compact_ccs_fall`, see “[The `va\_compact\_ccs\_rise` and `va\_compact\_ccs\_fall` Groups](#)” on page 6-19.

### *Example 6-3 va\_values With Three Variation Parameters*

```
...
timing_based_variation ( ) {
    va_parameters(var1, var2, var3); /* assumed that three variation parameters
                                     are var1, var2 and var3 */
    nominal_va_values(0.5, 1.0, 2.0);
    va_compact_ccs_rise ( ) {
        va_values(0.50, 1.0, 1.8); ...
        va_compact_ccs_rise ( ) {
            va_values(0.50, 1.0, 2.2); ...
            va_compact_ccs_rise ( ) {
                va_values(0.50, 0.9, 2.0); ...
                va_compact_ccs_rise ( ) {
                    va_values(0.50, 1.1, 2.0); ...
                    va_compact_ccs_rise ( ) {
                        va_values(0.45, 1.0, 2.0); ...
                        va_compact_ccs_rise ( ) {
                            va_values(0.55, 1.0, 2.0); ...
                }
            }
        }
    }
}
...

```

For information about using `va_values` with the `va_compact_ccs_rise` and `va_compact_ccs_fall` groups, see “[The `va\_compact\_ccs\_rise` and `va\_compact\_ccs\_fall` Groups](#)” on page 6-19.

### *Example 6-4 peak\_voltage in Values Attribute*

```
...
library(va_ccs) {
    compact_lut_template(clt) {
        index_3 ("init_current, peak_current, peak_voltage, peak_time, left_id,
right_id"); ...
        voltage_map(VDD1, 3.0);
        voltage_map(VDD2, 3.5);
        voltage_map(GND1, 0.5);
        voltage_map(GND2, 0.2);
    }
}
...

```

```

cell(test) {
    pg_pin(v1) {
        voltage_name : VDD1; ...
    }
    pg_pin(v2) {
        voltage_name : VDD2; ...
    }
    pg_pin(g1) {
        voltage_name : GND1; ...
    }
    pg_pin(g2) {
        voltage_name : GND2; ...
    }
    ...
    timing() {
        timing_based_variation () {
            va_parameters(Vthr);
            nominal_va_values(0.23);

            va_compact_ccs_rise (clt) {
                /* error : There are two power pins (v1 and v2)
                   and two ground pins (g1 and g2) in the cell "test".
                   The peak_voltage cannot be greater than the largest power
                   voltage, which is 3.5, and less than the smallest ground voltage,
                   which is 0.2. The value 4.0 is greater than 3.5 and 0.1
                   is less than 0.2. Both of them are wrong. */

                va_values(0.25);
                values("0.21, 0.54, 4.0, 0.36, 1, 2", \
                       "0.15, 0.55, 0.1, 0.85, 2, 4", ...); ...
            }

            ...
            timing_based_variation () {
                va_parameters(Vthr, VDD2, GND2);
                nominal_va_values(0.23, 3.5, 0.2);
                va_compact_ccs_rise (clt) {
                    /* In this group, the variation value of VDD2 is 4.1,
                       and GND2 is 0.0, so the largest power voltage is 4.1 and
                       the smallest ground voltage is 0.0. The peak_voltage 4.0 is
                       less than 4.1 and 0.1 is greater than 0.0, which is okay. */
                    va_values(0.18, 4.1, 0.0);
                    values("0.21, 0.54, 4.0, 0.36, 1, 2", \
                           "0.15, 0.55, 0.1, 0.85, 2, 4", ...); ...
                }

                ...
            }
        }
    }
}

```

For information about `peak_voltage` in `values` attribute see “[The `va\_compact\_ccs\_rise` and `va\_compact\_ccs\_fall` Groups](#) section.

### *Example 6-5 pin\_based\_variation Group*

```

...
pin(pin_name) {
    receiver_capacitance() {

```

```

    receiver_capacitance1_rise(template_name) {
        /* nominal input capacitance table */ ...
    ...
} pin_based_variation() {
    nominal_va_values(2.0, 4.54, 0.23);
    /* These nominal values apply to nominal input capacitance tables. */
    va_receiver_capacitance1_rise(template_name) {
        /* variational input capacitance table */
    ...
} ...
} /* end of pin */
...

```

For information on `peak_voltage` in `values` attribute see  
[“timing\\_based\\_variation and pin\\_based\\_variation Groups.”](#)

### *Example 6-6 pin-based Model With nominal CCS receiver model and Variation-aware CCS Receiver Model Groups*

```

/* Assume that there is no va_parameters defined */
library(lib_name) {
    ...
    timing() {
        timing_based_variation() {
            va_compact_ccs_rise(cltdf) {
                base_curves_group : base_name;
                va_values(2.4)
                /* error : can't find a corresponding va_parameters */ ...
            ...
        } /* end of timing_based_variation */
    ...
} /* end of library */

/* Assume that va_parameters is defined at the end of timing_based_variation
group and no default va_parameters is defined at library level */
...
    timing_based_variation() {
        nominal_va_values(2.0);
        /* error : can't find a corresponding va_parameters */
        ...
        va_parameters(Vthr);
        /* within a timing_based_variation, this shall be defined before
        all nominal_va_values and va_values attributes */
    } /* end of timing_based_variation */
    ...

/* Assume that va_parameters is defined only at library level */
...
library(lib_name) {
    ...
    timing_based_variation() {
        nominal_va_values(2.0);

```

```

    /* error : can't find a corresponding va_parameters */ ...
...
    va_parameters(Vthr);
    /* within a library, this shall be defined before all
       cell groups (or all nominal_va_values and va_values attributes) */
} /* end of library */

```

For information on `peak_voltage` in `values` attribute see “[The timing\\_based\\_variation Group](#).”

### *Example 6-7 nominal\_va\_values in Advanced CCS Modeling Usage*

```

/* ASSUME that there is no voltage_map defined in library */

library (sample) {
    operating_conditions (typical) {
        process : 1.5 ;
        temperature : 70 ;
        voltage : 2.75 ; ...}
    default_operating_conditions: typical;
...
    timing_based_variation() {
        va_parameters(voltage, temperature, process);
        nominal_va_values(2.00, 70, 1.5);
        /* error : The nominal voltage defined in
           default_operating_conditions is 2.75. values 2.00 is wrong. */

/* There is voltage_map defined at library level. */
library (sample) {
    operating_conditions (typical) {
        process : 1.5 ;
        temperature : 70 ;
        voltage : 2.75 ; ...}
    voltage_map(VDD1, 2.75);
    default_operating_conditions: typical;
...
    timing_based_variation() {
        va_parameters(voltage);
        nominal_va_values(2.00);
        /* This is okay because "voltage" is an user-defined parameter. */
...

```

### *Example 6-8 Variational Values in Advanced CCS Modeling Usage*

```

/* When var2 is in nominal value (1.0), var1 has three variational values (0.45,

```

```

0.55 and 0.50), which is wrong because only two are allowed. When var1 is in
nominal value (0.50), var2 has two variational values (0.8 and 1.0). 0.8 is
less than nominal value (1.0), which is okay, however, 1.0 is not greater than
nominal value, which is wrong. */

...
timing_based_variation ( ) {
    va_parameters(var1, var2);
    nominal_va_values(0.50, 1.0);
    va_receiver_capacitance2_rise (temp_1)  {
        va_values(0.45, 1.0);
        ...
    }
    va_receiver_capacitance2_rise (temp_1)  {
        va_values(0.55, 1.0);
        ...
    }
    va_receiver_capacitance2_rise (temp_1)  {
        va_values(0.50, 0.8);
        ...
    }
    va_receiver_capacitance2_rise (temp_1)  {
        va_values(0.50, 1.0);
        ...
    }
}
...

```

### *Example 6-9 Variation-Aware CCS Driver or Receiver with Timing Constraints*

```

library(my_lib) {
    ...
    base_curves (ctbct1){
        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(3, "0.7, 0.5, 0.45");
        .....
        curve_y(37, "0.23, 1.4, 6.23");
    }
    compact_lut_template(LUT4x4) {
        variable_1 : input_net_transition;
        variable_2 : total_output_net_capacitance;
        variable_3 : curve_parameters;
        index_1("0.1, 0.2, 0.3, 0.4");
        index_2("1.0, 2.0, 3.0, 4.0");
    }
}

```

```

        index_3 ("init_current, peak_current, peak_voltage, peak_time, left_id,
right_id");
        base_curves_group: "ctbct1";
    }
    lu_table_template(LUT3) {
        variable_1: input_net_transition;
        index_1("0.1, 0.3, 0.5");
    }
    lu_table_template(LUT3x3) {
        variable_1: input_net_transition;
        variable_2: total_output_net_capacitance;
        index_1("0.1, 0.3, 0.5");
        index_2("1.0, 3.0, 5.0");
    }
    lu_table_template(LUT5x5) {
        variable_1: constrained_pin_transition;
        variable_2: related_pin_transition;
        index_1("0.01, 0.05, 0.1, 0.5, 1");
        index_2("0.01, 0.05, 0.1, 0.5, 1");
    }
}
...
cell(INV1) {
...
pin (A) {
    direction: input;
    capacitance: 0.3;
    receiver_capacitance () {
        ...
    }
    pin_based_variation () {
        va_parameters(channel_length, threshold_voltage);
        nominal_va_values(0.5, 0.5) ;
        va_receiver_capacitance1_rise (LUT3) {
            va_values(0.50, 0.45);
            values("0.29, 0.30, 0.31");
        }
        va_receiver_capacitance1_rise (LUT3) {
            va_values(0.50, 0.55);
            ...
        }
        va_receiver_capacitance1_rise (LUT3) {
            va_values(0.45, 0.50);
            ...
        }
        va_receiver_capacitance1_rise (LUT3) {
            va_values(0.55, 0.50);
            ...
        }
    }
}

```

```

va_receiver_capacitance2_rise (LUT3) {
    va_values(0.50, 0.45);
    values("0.19, 8.60, 5.41");
}
va_receiver_capacitance2_rise (LUT3) {
    va_values(0.50, 0.55);
    ...
}
va_receiver_capacitance2_rise (LUT3) {
    va_values(0.45, 0.50);
    ...
}
va_receiver_capacitance2_rise (LUT3) {
    va_values(0.55, 0.50);
    ...
}
va_receiver_capacitance1_fall (LUT3) {
    va_values(0.50, 0.45);
    values("0.53, 2.16, 9.18");
}
va_receiver_capacitance1_fall (LUT3) {
    va_values(0.50, 0.55);
    ...
}
va_receiver_capacitance1_fall (LUT3) {
    va_values(0.45, 0.50);
    ...
}
va_receiver_capacitance1_fall (LUT3) {
    va_values(0.55, 0.50);
    ...
}
va_receiver_capacitance2_fall (LUT3) {
    va_values(0.50, 0.45);
    values("0.39, 0.98, 5.15");
}
va_receiver_capacitance2_fall (LUT3) {
    va_values(0.50, 0.55);
    ...
}
va_receiver_capacitance2_fall (LUT3) {
    va_values(0.45, 0.50);
    ...
}
va_receiver_capacitance2_fall (LUT3) {
    va_values(0.55, 0.50);
    ...
}

```

```

        }
    } /* end of pin */
pin (Y) {
    direction: output;
    timing ( ) {
        related_pin: "A";
        compact_ccs_rise(LUT4x4) {
            ...
        }
        compact_ccs_fall(LUT4x4) {
            ...
        }
        timing_based_variation() {
            va_parameters(channel_length, threshold_voltage);
            nominal_va_values(0.50, 0.50);
            va_compact_ccs_rise (LUT4x4 ) { /* without optional fields */
                va_values(0.50, 0.45);
                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", \
                    "0.1, 0.2, 0.3, 0.4, 1, 3", \
                    "0.2, 0.3, 0.4, 0.5, 4, 5", \
                    "0.3, 0.4, 0.5, 0.6, 2, 4", \
                    "0.4, 0.5, 0.6, 0.7, 7, 8", \
                    "0.5, 0.6, 0.7, 0.8, 10, 4", \
                    "0.5, 0.6, 0.8, 0.9, 11, 2", \
                    "0.25, 0.55, 1.65, 1.85, 3, 4", \
                    "1.2, 1.6, 1.7, 1.9, 5, 2", \
                    "1.1, 2.2, 2.3, 0.4, 1, 30", \
                    "1.2, 2.3, 1.4, 0.5, 17, 5", \
                    "1.3, 2.4, 1.5, 0.6, 22, 24", \
                    "1.4, 2.5, 1.6, 1.7, 17, 18", \
                    "1.5, 2.6, 0.7, 0.8, 10, 33");
            }
            va_compact_ccs_rise (LUT4x4 ) {
                va_values(0.50, 0.55);
                ...
            }
            va_compact_ccs_rise (LUT4x4 ) {
                va_values(0.45, 0.50);
                ...
            }
            va_compact_ccs_rise (LUT4x4 ) {
                va_values(0.55, 0.50);
                ...
            }
            va_compact_ccs_fall (LUT4x4 ) { /* without optional fields */
                ...
            }
        }
    }
}

```

```

        }
        ...
    } /* end of timing_based_variation */
    ...
} /* end of timing */
...
} /* end of pin */
...
} /* end of cell */
...

cell(INV4) {
    ...
    pin (Y) {
        direction: output;
        timing ( ) {
            related_pin: "A";
            receiver_capacitance1_rise (LUT3x3) {
                ...
            }
            receiver_capacitance2_rise (LUT3x3) {
                ...
            }
            receiver_capacitance1_fall (LUT3x3) {
                ...
            }
            receiver_capacitance2_fall (LUT3x3) {
                ...
            }
            rise_constraint(LUT5x5) {
                ...
            }
            fall_constraint(LUT5x5) {
                ...
            }
            timing_based_variation ( ) {
                va_parameters(channel_length,threshold_voltage);
                nominal_va_values(0.50, 0.50) ;
                va_receiver_capacitance1_rise (LUT3x3) {
                    va_values(0.50, 0.45);
                    values( "1.10, 1.20, 1.30", \
                            "1.11, 1.21, 1.31", \
                            "1.12, 1.22, 1.32");
                }
                va_receiver_capacitance2_rise (LUT3x3) {
                    va_values(0.50, 0.45);
                    values("1.20, 1.30, 1.40", \
                           "1.21, 1.31, 1.41", \

```

```

        "1.22, 1.32, 1.42");
    }
    va_receiver_capacitance1_fall (LUT3x3) {
        va_values(0.50, 0.45);
        values("1.10, 1.20, 1.30", \
               "1.11, 1.21, 1.31", \
               "1.12, 1.22, 1.32");
    }
    va_receiver_capacitance2_fall (LUT3x3) {
        va_values(0.50, 0.45);
        values( 1.20, 1.30, 1.40", \
                "1.21, 1.31, 1.41", \
                "1.22, 1.32, 1.42");
    }
    va_receiver_capacitance1_rise (LUT3x3) {
        va_values(0.50, 0.55);
        ...
    }
    ...
    va_receiver_capacitance1_rise (LUT3x3) {
        va_values(0.45, 0.50);
        ...
    }
    ...
    va_receiver_capacitance1_rise (LUT3x3) {
        va_values(0.55, 0.50);
        ...
    }
    ...
    va_rise_constraint(LUT5x5) {
        va_values(0.50, 0.45);
        values( "-0.1452, -0.1452, -0.1452, -0.1452, 0.3329", \
                "-0.1452, -0.1452, -0.1452, -0.1452, 0.3952", \
                "-0.1245, -0.1452, -0.1452, -0.1358, 0.5142", \
                " 0.05829,  0.0216,  0.01068,  0.06927,  0.723", \
                " 1.263, 1.227, 1.223, 1.283, 1.963");
    }
    va_rise_constraint(LUT5x5) {
        va_values(0.50, 0.55);
        ...
    }
    va_rise_constraint(LUT5x5) {
        va_values(0.55, 0.50);
        ...
    }
    va_rise_constraint(LUT5x5) {
        va_values(0.45, 0.50);
        ...
    }

```

```

}
va_fall_constraint(LUT5x5) {
    va_values(0.50, 0.55);
    ...
}
va_fall_constraint(LUT5x5) {
    va_values(0.55, 0.50);
    ...
}
va_fall_constraint(LUT5x5) {
    va_values(0.45, 0.50);
    ...
}
va_fall_constraint(LUT5x5) {
    va_values(0.50, 0.45);
    ...
}
} /* end of timing_based_variation */
...
} /* end of timing */
...
} /* end of pin */
...
} /* end of cell */
...
} /* end of library */

```

# 7

## Composite Current Source Signal Integrity Modeling

---

This chapter provides an overview of composite current source modeling (CCS) to support noise (signal integrity) modeling for advanced technologies. The following information is covered:

- Composite Current Source Signal Integrity Noise Model

---

## Composite Current Source Signal Integrity Noise Model

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 Noise Information

When you compile a library containing CCS noise information with the `read_lib` command in Library Compiler (after screening data without finding any errors), Library Compiler 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, `lc_disable_ccs_noise_extraction` is set to 0, which means that Library Compiler screens and extracts CCS noise information in the libraries with the `read_lib` command.

If `lc_disable_ccs_noise_extraction` 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.

---

## Syntax

The CCS noise syntax is as follows:

```
library (name) {
...
  cell (name) {
    pin (name) {
      ...
        ccsn_first_stage () {
          is_needed : <boolean>;
          is_inverting : <boolean>;
          stage_type : <stage_type_value>;
          miller_cap_rise : <float>;
          miller_cap_fall : <float>;
          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>;
}
}
}
}

```

---

## Screening Rules

This section lists error and warnings messages that Library Compiler issues.

## Screening Errors

Library Compiler issues an error message if one of the following conditions is not satisfied:

- For a library with CCS noise data, each timing group can only contain the `ccsn_last_stage` group when the `ccsn_first_stage` group is present. Otherwise, Library Compiler reports an error.
- The pin direction must be consistent with the `ccsn_first_stage` or `ccsn_last_stage` group in the pin group. This involves an input pin with a `ccsn_first_stage` group, an output pin with a `ccsn_last_stage` group, and an inout pin with both a `ccsn_first_stage` group and `ccsn_last_stage` group. Otherwise, Library Compiler reports an error.
- For pin-based CCS noise models, the pin function and `driver_type` must be consistent with the subgroups of the `ccsn_first_stage` or `ccsn_last_stage` group.

For example, the `ccsn_first_stage` or `ccsn_last_stage` group on a driver pin with `driver_type` of `pull_up` must have `stage_type` with a value of `PULL_UP`. Otherwise, Library Compiler reports an error.

- The timing arc type and sense must be consistent with the sense of the `ccsn_first_stage` or `ccsn_last_stage` group on the arc. Only combinational arcs and sequential arcs including `rising_edge`, `falling_edge`, `clear`, and `preset` can have `ccsn_first_stage` and `ccsn_last_stage` groups.

The arc sense must match the combination (chaining) of the senses of the contained `ccsn_first_stage` and `ccsn_last_stage` groups. Specifically, an inverting arc must

contain only a single inverting `ccsn_first_stage` group or an inverting `ccsn_first_stage` group and a noninverting `ccsn_last_stage` group, or a noninverting `ccsn_first_stage` group and an inverting `ccsn_last_stage` group. Otherwise, Library Compiler reports an error.

- If the `is_needed` attribute is set to `FALSE` in the `ccsn_first_stage` or `ccsn_last_stage` group, Library Compiler verifies that there is no other CCS noise data within the same `ccsn_first_stage/ccsn_last_stage` group.
- If the `is_needed` attribute is set to `TRUE` in the `ccsn_first_stage` or `ccsn_last_stage` group, Library Compiler verifies the following:
  - The `stage_type` and `is_inverting` attributes are both defined.
  - If the `stage_type` attribute is set to `PULL_DOWN`, the `miller_cap_fall` attribute, `output_voltage_fall` group, and `propagated_noise_high` group are required. Library Compiler does not issue warnings if the `miller_cap_rise` attribute, `output_voltage_rise` group, and `propagated_noise_low` group exist.
  - If the `stage_type` attribute is set to `PULL_UP`, then the `miller_cap_rise` attribute, `output_voltage_rise` group, and `propagated_noise_low` group are required. Library Compiler does not issue warnings if the `miller_cap_fall` attribute, `output_voltage_fall` group, and `propagated_noise_high` group exist.
  - If the `stage_type` attribute is set to `BOTH`, then the following attribute or groups are required:

- `miller_cap_fall` attribute
- `miller_cap_rise` attribute
- `output_voltage_rise` group
- `output_voltage_fall` group
- `propagated_noise_rise` group
- `propagated_noise_high` group
- If the `is_needed` attribute is set to TRUE in the `ccsn_first_stage` or `ccsn_last_stage` group, Library Compiler verifies the following:
  - That the `dc_current` group exists.
  - That the values of both `index_1` and `index_2` in the `dc_current` group need to be in increasing order.
  - For the values of both `index_1` and `index_2` in the `dc_current` group, that the first entry must be less than or equal to VSS, and the last entry must be greater than or equal to VDD. Otherwise, Library Compiler issues a warning and not an error message.
- When either the `miller_cap_rise` or `miller_cap_fall` attribute exist, Library Compiler verifies the following:
  - That the value of the attribute is greater than or equal to 0.
  - That the value of the attribute is not greater than the library maximum capacitance. Otherwise, Library Compiler issues a warning and not an error.
- For the `output_voltage_rise` or `output_voltage_fall` group, Library Compiler verifies the following:

- It has at least one vector group.
- The template to which the vector group contains only the following three variables as `index_1`, `index_2`, and `index_3`, respectively: `input_net_transition`, `total_output_net_capacitance`, and `time`.
- The values of both `index_1` and `index_2` of the vector group must be positive or 0.
- The values of `index_3` of the vector group must be positive or 0 and they must be in strict increasing order.
- The voltage values (values attribute) of the vector group must be between VSS (inclusive) and VDD (inclusive).
- For the `propagated_noise_low` or `propagated_noise_high` group, Library Compiler verifies the following:
  - It has at least one vector group.
  - The template that the vector group refers to contains only the following four variables as `index_1`, `index_2`, `index_3` and `index_4` respectively: `input_noise_height`, `input_noise_width`, `total_output_net_capacitance`, and `time`.
  - The value of `index_1` (`input_noise_height`) of the vector group must be between VSS (not inclusive) and VDD (inclusive).
  - The values of `index_2` (`input_noise_width`) of the vector group must be greater than 0 (not inclusive).
  - The values of `index_3` (`total_output_net_capacitance`) of the vector group must be greater than or equal to 0.

- The values of `index_4` of the vector group must be positive or 0 and they must be in strict increasing order.
- The voltage values (values attribute) of the vector group must be between VSS (inclusive) and VDD (inclusive).
- The size of the `dc_current` table in the `ccsn_first_stage`/`ccsn_last_stage` group must be at least 6x6.

## **Library Compiler Screener Warnings**

Library Compiler issues a warning if either of the following conditions is not satisfied:

- Each input pin or bus must have at least one `ccsn_first_stage` group defined at the pin or bus or `ccsn_first_stage` defined at one timing group inside the pin or bus.
- Each output pin or bus must have at least one `ccsn_last_stage` group defined at the pin or bus or all of its timing groups must specify the `ccsn_first_stage` group. This applies to load, dummy, and via cells.

## **Sample CCS Noise Library**

The following is a sample CCS noise library.

### *Example 7-1 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,
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 */
```



# 8

## 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](#)
- [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 8-7. The leakage current syntax is as follows:

### *Example 8-1 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 will infer 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 8-2 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 */  
        ...  
    }  
}
```

## **Leakage Current Syntax Checking**

Library Compiler checks the following conditions in the leakage current syntax and reports error and warning messages as applicable:

- The when attribute must be a valid Boolean expression with the cell's structural pins. Otherwise, Library Compiler issues an error.
- Each when state must be mutually exclusive in a cell, which means that only one condition defined in the when state can be met at any given time. Otherwise, Library Compiler issues an error.
- For a cell with a single power and ground pin, either simplified format or regular format is allowed. Otherwise, Library Compiler issues an error.

The simplified format is allowed for a cell with a single power and ground pin. For the regular format, the `pg_current` group is required within a `leakage_current` group. Simplified and regular syntax formats cannot be applied to the same cell.

- Only one default state is allowed within a cell. Otherwise, a warning message is issued, and Library Compiler stores only the last default state and discards the others.
- Regular format must be chosen for a cell with multiple power or ground pins. Otherwise, Library Compiler issues an error.

The following rules apply only for the regular format:

- A power or ground pin is required, and only one specified power and ground pin is allowed for each `pg_current` group. You must provide a valid single power or ground pin of the cell. Otherwise, Library Compiler issues an error.
- Only one power or ground pin can be omitted within a `pg_current` group under a `leakage_current` group. Otherwise, Library Compiler issues an error.
- If a power or ground pin is a power pin or internal ground pin, the value of the leakage current must be 0 or a positive floating-point number. Otherwise, Library Compiler issues an error.
- If a power or ground pin is a ground pin or internal power pin, the value of the leakage current must be 0 or a negative floating-point number. Otherwise, Library Compiler issues an error.
- If all power and ground pins in a `leakage_current` group are specified, then values must sum up to 0.0 with a relative error tolerance of 1.0e-6 (relative to sum of the absolute values), see equation below:

$$0 \leq \frac{|\sum x|}{\sum |x|} \leq 1.0e - 6$$

where x is value of leakage\_current. Otherwise, Library Compiler issues an error.

- If there is more than one pg\_current group under a leakage\_current group, the pg\_current group must be assigned to different power or ground pins. Otherwise a warning message is issued to alert you that there are duplicated pg\_current groups. In this case, only the last group is recognized by Library Compiler. Otherwise, Library Compiler issues a warning.

The following rule is applied only for the simplified format:

- The value of leakage current must be 0 or a positive floating-point number. Otherwise, Library Compiler issues an error.

## 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(<a pg pin name>) {
            value : <float>;
        }
        ...
        gate_leakage(<an 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 8-3.

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` 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` is specified in `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.

## **Library Compiler Screener**

Library Compiler checks the following items under a leakage\_current group, and reports error or warning messages if applicable.

- If there is more than one gate\_leakage group under a leakage\_current group, the gate\_leakage groups must be assigned to different input or inout pins. Otherwise, a warning message is issued, and only the last group is stored by Library Compiler.
- An input or inout pin is required. Only one specified pin is allowed for each gate\_leakage group, and it must be a valid single pin of the cell. Otherwise, Library Compiler issues an error.
- `input_high_value` must be positive or zero if it is specified. Otherwise, Library Compiler issues an error.
- `input_low_value` must be negative or zero if it is specified. Otherwise, Library Compiler issues an error.
- The `input_high_value` must be 0.0 or omitted if the state condition implies that an input pin can not be logic one. Otherwise, Library Compiler issues an error.
- The `input_low_value` must be 0.0 or omitted if the state condition implies that an input pin can not be logic zero. Otherwise, Library Compiler issues an error.
- There is no check for current conservation if `gate_leakage` is present. Library Compiler ignores and does not issue an error or warning.

---

## Intrinsic Parasitic

You can use the syntax in [Example 8-3](#) for intrinsic parasitic models. The syntax consists of two parts: one is intrinsic resistance and the other is intrinsic capacitance.

### *Example 8-3 Intrinsic Parasitic Model*

```
cell (<cell_name>) {
    ...
    intrinsic_parasitic() {
        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 */
    }
}
```

## Intrinsic Parasitic Model Syntax Checking

Library Compiler checks the following conditions in the intrinsic model syntax and reports error or warning messages if applicable:

- The `when` attribute must be a valid Boolean expression in Library Compiler. Otherwise, Library Compiler issues an error.
- Each `when` state must be mutually exclusive in a cell, which means that only one condition defined in the `when` state can be met at any given time. Otherwise, Library Compiler issues an error.

- A power or ground pin is required and only one specified power or ground pin is allowed for each `intrinsic_capacitance` group. You must provide a valid single power or ground pin of a cell. Otherwise, Library Compiler issues an error.
- A `related_output` is required and only one specified output pin is allowed for each `intrinsic_resistance` group. You must provide a valid single logic output or inout pin. Otherwise, Library Compiler issues an error.
- Intrinsic resistance and capacitance values must be positive floating-point numbers. Otherwise, Library Compiler issues an error.
- One default state is allowed within a cell. Otherwise a warning message is issued. Library Compiler keeps only the last default state and discards the others.
- If there is more than one `intrinsic_capacitance` group under an `intrinsic_parasitic` group, they must contain different power or ground pins. Otherwise a warning message is issued to alert that there are duplicated `intrinsic_capacitance` groups, and only the last group is recognized by Library Compiler.
- If there is more than one `intrinsic_resistance` group under an `intrinsic_parasitic` group, they must contain a different output pin. Otherwise a warning message is issued to alert that there are duplicated `intrinsic_resistance` groups, and only the last group is recognized by Library Compiler.

---

## 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 value is provided by the tool.
- The parasitics modeling of total capacitance in macros cells is not state dependent. This means that there is no state condition specified in `intrinsic_parasitic`.

### Syntax

```
cell (<cell_name> ) {  
    power_cell_type : <enum(stdcell, macro)>;  
    ...  
    intrinsic_parasitic() {  
        total_capacitance(<a pg pin name>) {  
            value : <float>;  
        }  
        ...  
    }  
    ...  
}
```

## **Library Compiler Screener**

Library Compiler checks the following items in a parasitics model.

- If there is more than one `total_capacitance` group under an `intrinsic_parasitic` group, it must be assigned to different power and ground pins. Otherwise, a warning message is issued to alert that a duplicated `total_capacitance` group exists. In this case, Library Compiler only stores the last group.
- A power or ground pin is required and only one specified pin is allowed for each `total_capacitance` group, and it must be a valid single power or ground pin of the cell. Otherwise, Library Compiler issues an error.
- If `power_cell_type` is a macro cell and there is more than one `intrinsic_parasitic` group under a cell group, a warning message is issued and only the last group counts by Library Compiler.
- No state condition is allowed in `intrinsic_parasitic` group if `power_cell_type` is a macro cell. Otherwise, Library Compiler issues an error.
- No `intrinsic_capacitance` and `intrinsic_resistance` groups are allowed in `intrinsic_parasitic` group if `power_cell_type` is a macro cell. Otherwise, Library Compiler issues an error.
- No `total_capacitance` group is allowed in the `intrinsic_parasitic` group if `power_cell_type` is a standard cell. Otherwise, Library Compiler issues an error.

---

## 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.

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.

### Syntax

The following syntax for dynamic power format provides more accurate models for macro cells:

...

```

cell(<cell_name>) {
    ...
    power_cell_type : <enum(stdcell, macro)>
dynamic_current() {
    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`.

### **Library Compiler Screener**

Library Compiler checks the following items in a dynamic power model.

- The `power_cell_type` attribute must be a macro cell if the `min_input_switching_count` and `max_input_switching_count` attributes are specified. Otherwise, Library Compiler issues an error.
- The `min_input_switching_count` and `max_input_switching_count` attributes must always be specified in pair within a `switching_group`. Otherwise, Library Compiler issues an error.
- The `min_input_switching_count`, `max_input_switching_count`, and the `input_switching_condition` attributes should not be specified in the same `switching_group` group. Otherwise, Library Compiler issues an error.
- If the `min_input_switching_count` and `max_input_switching_count` attributes are defined in one `switching_group` group, they must be defined for the rest of the `switching_group` groups under the same `dynamic_current` group. Otherwise, Library Compiler issues an error.
- The counts within a `dynamic_current` should cover all number bits of input specified in `related_inputs`. Otherwise, Library Compiler issues an error.
- The counts within a `dynamic_current` cannot overlap. Otherwise, Library Compiler issues an error.
- `min_input_switching_count` must be greater than 0. Otherwise, Library Compiler issues an error.
- The `max_input_switching_count` attribute must be greater than or equal to `min_input_switching_count` and less than or equal to the total number of bits in the input pins specified in `related_inputs`. Otherwise, Library Compiler issues an error.

## 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 */
...

```

## Power and Ground Current Syntax Checking

Library Compiler checks the following conditions in power or ground current template syntax and issues an error message if any of the conditions are not satisfied:

- The last variable must be `time`. The `time` variable is required.
- `input_net_transition` and `total_output_net_capacitance` are two available values for all variables except for the last variable. They can be placed in any order.

- There can be zero or one `input_net_transition` variable.
  - There can be zero, one, or two `total_output_net_capacitance` variables.
- 

## Dynamic Current Syntax

The following syntax is used for instantaneous power data, which is captured at the cell level.

### *Example 8-4 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 */
```

## **Dynamic Current Syntax Checking**

Library Compiler checks the following conditions in the dynamic current syntax and reports error or warning messages as applicable:

- The `when` attribute must be a valid Boolean expression in Library Compiler. Otherwise, Library Compiler issues an error.
- The values specified in the `when`, `related_inputs`, and `related_outputs` attributes must be mutually exclusive. Otherwise, Library Compiler issues an error.
- The number of entries in the `typical_capacitances` and `related_outputs` attributes must be identical. Otherwise, Library Compiler issues an error.
- If `related_outputs` is specified, the list in the attribute must not be empty, and the entries in the list must be either output or inout pins without repetition. Otherwise, Library Compiler issues an error.
- At most, one `related_outputs` attribute can be specified for each `dynamic_current` group. Otherwise, a warning message is issued and only the last `related_outputs` attribute is recognized by Library Compiler. All the duplicated attributes are discarded.
- Bus and bundle can be specified in a `related_outputs` attribute only in the bit level. Otherwise, Library Compiler issues an error.
- Bus and bundle can be supported in the `related_inputs` attribute. Otherwise, Library Compiler issues an error.

- Only one `related_inputs` attribute is required for each `dynamic_current` group. Otherwise, a warning message is issued and only the last `related_inputs` attribute is recognized by Library Compiler. All the duplicated attributes are discarded.
- The list in a `related_inputs` attribute must not be empty, and the entries in the list must be either input or inout pins without repetition. Otherwise, Library Compiler issues an error.

Two `dynamic_current` groups are considered to be overlapping if both of the following conditions are true:

- Two `dynamic_current` groups have at least one identical input pin in the list of their `related_inputs` attributes, and both groups have the same `related_outputs` list or both have no `related_outputs` defined.
- The `when` attributes specified within two `dynamic_current` groups are not mutually exclusive, or there is no `when` attribute specified within both groups.
- No two `dynamic_current` groups in the same cell can be in an overlapping condition as defined above. Otherwise, Library Compiler issues an error.
- At least one `switching_group` must be defined for each `dynamic_current` group. Otherwise, Library Compiler issues an error.
- At least one `pg_current` must be defined for each `switching_group` group. Otherwise, Library Compiler issues an error.
- If cell type is macro, then `related_outputs` and `output_switching_condition` attributes cannot be applied. Otherwise, Library Compiler issues an error.

- The values in `input_switching_condition` can be either `rise` or `fall`. Otherwise, Library Compiler issues an error.
- The `input_switching_condition` attribute is optional if both conditions `rise` and `fall` of the input pin can be applied for the same output condition. If it is defined, only one value is allowed. Otherwise, Library Compiler issues an error.
- Only one `input_switching_condition` attribute is allowed within a `switching_group`. Otherwise, a warning message is issued, and Library Compiler takes only the last attribute and discards the others.
- Only one `output_switching_condition` attribute is allowed within a `switching_group`. Otherwise a warning message is issued, and Library Compiler takes only the last attribute and discards the others. If `rise` or `fall` can be applied, the `input_switching_condition` is optional.
- The number of entries in an `output_switching_condition` and `related_outputs` attribute must be identical. Otherwise, Library Compiler issues an error.
- The values in `output_switching_condition` can be either `rise` or `fall`. Otherwise, Library Compiler issues an error.

Two `switching_group` groups are considered to be an overlapping condition if both of the following bullets are true:

- `output_switching_condition` is undefined in both `switching_group` groups  
or  
`output_switching_condition` is defined and the values of it are identical in two `switching_group` groups

- `input_switching_condition` is undefined on both `switching_group` groups  
or  
the values are different but one is undefined and the other has a value `rise` or `fall`
- No two `switching_group` groups under the same `dynamic_current` group can be in an overlapping condition. Otherwise, Library Compiler issues an error.
- Only one name can be labeled in `pg_current` and it must be a valid power or ground pin. Otherwise, Library Compiler issues an error.
- No two `pg_current` groups under the same `switching_group` can be labeled with the same name. Otherwise a warning message is issued, and Library Compiler takes only the last `pg_current` group and discards the other duplicated groups.
- The name specified in the index related to the output pin variable attribute must match one of the values in `related_outputs`, and it must be a valid output pin. Otherwise, Library Compiler issues an error.
- The only allowed values in `power_cell_type` are `stdcell` and `macro`. Otherwise, Library Compiler issues an error.
- All vectors must be specified with the same template under the same `pg_current` group. Otherwise, Library Compiler issues an error.
- If the `index_output` attribute is defined in one vector, it must be defined for the rest of the vectors under the same `pg_current` group. Otherwise, Library Compiler issues an error.

- If the `index_output` attribute is specified in one vector, then `typical_capacitances` must be defined under the same `dynamic_current` where vector data are defined. Otherwise, Library Compiler issues an error. Otherwise, Library Compiler issues an error.
- If `index_output` attribute is specified in one of vectors within a `pg_current` group, the `related_outputs`, defined under the same `dynamic_current` where the vector is defined, must contain more than one pin. Otherwise, Library Compiler issues an error.
- If no `index_output` attribute is defined in all vectors under a `pg_current` group and the template that those vectors refer to contains only one `total_output_net_capacitance`, `related_outputs`, defined under the same `dynamic_current` as where vectors are defined, must contain at least one pin. Otherwise, Library Compiler issues an error.
- If the `index_output` attribute is defined in one vector, the template that applied for that vector must contain exactly one `total_output_net_capacitance` variable. Otherwise, Library Compiler issues an error. Otherwise, Library Compiler issues an error.
- A cross point is defined as follows: if an `index_output` is defined in a vector, and the value of `total_output_net_capacitance` `index` is the same as one of the values specified in the `typical_capacitances` attribute, that value maps to one of the pins in the `related_outputs` attribute, which is the same pin as specified by the `index_output` attribute. Otherwise, Library Compiler issues an error.

- If `pg_current` is represented as a sparse cross table, there must be at least one cross point in one of the vectors. Cross point is defined in the previous example. Otherwise, Library Compiler issues an error.
- If two cross point vector groups have the same index value of `input_net_transition`, the index values of time, values, and `reference_time` must be the same for both vectors. Cross point is defined above. Otherwise, Library Compiler issues an error.
- Under the same `pg_current` group, no two vectors can have the same `index_output` and the values of all index attributes, except for the last index (`time`).
- Only one value is allowed for all index attributes, except for the last one. Otherwise, Library Compiler issues an error.
- The number of entries must be the same between time, index and values attributes. Otherwise, Library Compiler issues an error.
- If a template with two `total_output_net_capacitance` variables (or dense table) is applied to one of the vectors under a `dynamic_current` group, the exact two output pins must be specified in a `related_output` attribute. Otherwise, Library Compiler issues an error.
- If a template with two `total_output_net_capacitance` variables is applied to all vectors under a `pg_current` group, all possible combination of capacitances between two output pins must be specified. In short, the table must be dense. Otherwise, Library Compiler issues an error.

The following two rules are applied for all vectors with a template containing an `input_net_transition` variable under a `pg_current` group. The rule pertaining to cross points is assumed to be applied before the first of the next two rules can be considered. Otherwise, Library Compiler issues an error.

If two cross points vector groups have the same index value of `input_net_transition`, `index` values of time, values, and `reference_time` must be the same for both vectors. Otherwise, Library Compiler issues an error.

- Number of entries of `total_output_net_capacitance` values (or size of `total_output_net_capacitance`) for the same output pin or `index_output` (cross type) must be identical for all vectors that have different `input_net_transition` `index` values. Otherwise, Library Compiler issues an error.
- The values of `total_output_net_capacitance` for the same output pin or `index_output` (cross type) must be identical for all vectors that have different `input_net_transition` `index` values. Otherwise, Library Compiler issues an error.

---

## 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

For more information about the syntax in the following examples, see the *Library Compiler Reference Manual: Technology and Symbol Libraries*.

## 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() {
        when: D;
        related_inputs : A B;
        related_outputs : Q QN;
        switching_group ( ) {
            input_switching_condition(rise);
            output_switching_condition(rise,
                fall);
        pg_current (VSS ) {
            vector ( CCS_power_1 ) {
                reference_time : 0.01;
                index_1 ( 0.01 )
                index_2 ( 1.0 )
                index_3 ( 1.0 )
                index_4 ( 0.000, 0.0873, 0.135, 0.764)
                values ( 0.002, 0.009, 0.134, 0.546)
            }
        }
    }
}
```

## 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_capacitance(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)
}
}
```

# 9

## Interface Timing of Complex Sequential Blocks

---

This chapter describes the procedure and methodology for specifying the static timing of complex sequential 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 interface 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

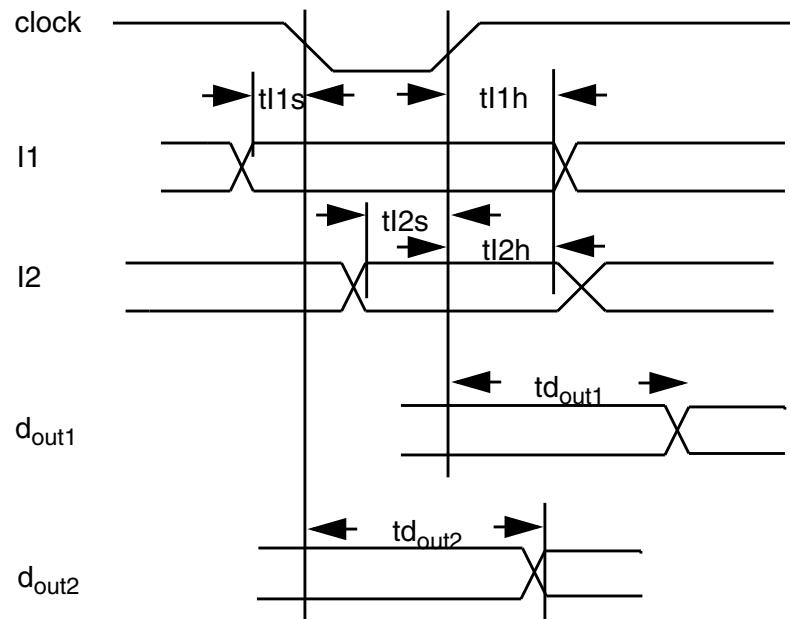
## Interface Timing Specification

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 9-1 is an example of the interface timing of a complex block.

*Figure 9-1 Interface Timing of a Complex Block*



In Figure 9-1,

- 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

When a cell is interpreted as above, some of the timing information specified in the library is ignored. For example, if a cell is interpreted as a rising-edge-triggered device, Library Compiler 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. Library Compiler 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 more effectively with PrimeTime 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 PrimeTime, see the PrimeTime suite of documents: *PrimeTime Modeling User Guide*, *PrimeTime Tutorial*, and *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, Library Compiler writes out a separate model design .db as an instantiation of the model cell plus the net connections for the shorted ports.

Library Compiler 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_id") ;
```

## **Example**

```
short (b, y) ;
```

## **model and short Examples**

[Example 9-1](#) shows `model` and `short` descriptions.

### *Example 9-1 model Group With short Attributes*

```
model(modelA) {
    area : 0.4;
    short(b, y);
    short(c, y);
}
cell_name : "cell";
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 (nameid) {
    ...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 9-2](#) shows a generated clock description.

### *Example 9-2 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.
  - Specify the `cell` group or the `model` group separately and use the `update_lib` command to associate it with a library that already exists in `.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.

---

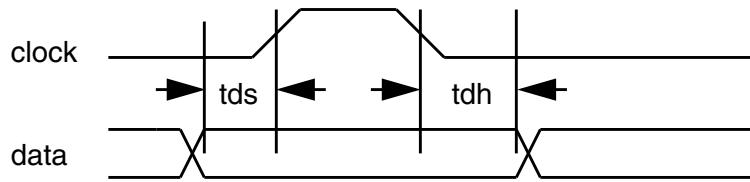
## Examples of Interface Timing Relationships

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 9-2](#) 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 9-2 Data Input With Specified Setup and Hold Constraints*



[Example 9-3](#) shows the Library Compiler description of these relationships.

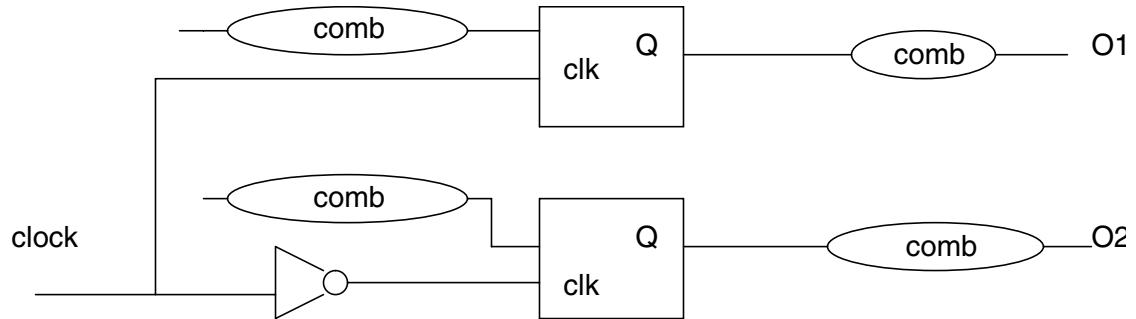
*Example 9-3 Description of Timing Relationships in Figure 9-2*

```
pin (I2) {
    timing () {
        timing_type : setup_rising;
        intrinsic_rise : 2.20; /* tds */
        intrinsic_fall : 2.20;
        related_pin : "clock";
    }
    timing () {
        timing_type : hold_falling;
        intrinsic_rise : 0.95; /* tdh */
        intrinsic_fall : 0.95;
        related_pin : "clock";
    }
}
```

## Output Propagation

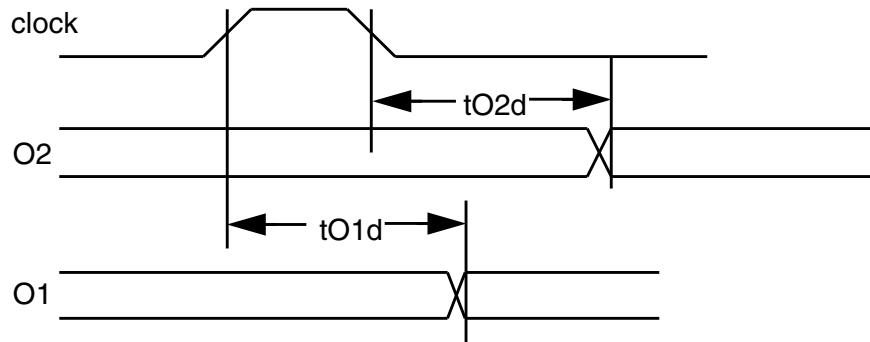
The output delay can be specified with respect to the rising or falling edge of the clock. [Figure 9-3](#) shows the output section of a block. Output O1 is propagated on the rising edge of the clock, and output O2 is propagated on the falling edge.

*Figure 9-3 Output Section of a Macro Block*



*Figure 9-4 shows the interface timing relationships for the two outputs.*

*Figure 9-4 Output Timing Relationships for the Block in Figure 9-3*



*Example 9-4 shows the Library Compiler description of the relationships in Figure 9-4.*

*Example 9-4 Description of Timing Relationships in Figure 9-4*

```

pin(O1) {
    direction : output;
    timing () {
        timing_type : rising_edge; /*sequential delay arc */
        intrinsic_rise : 2.25; /* t01d */
        rise_resistance : 0.020;
        intrinsic_fall : 2.57;
    }
}
  
```

```

        fall_resistance : 0.017;
        related_pin : "clock";
    }
}

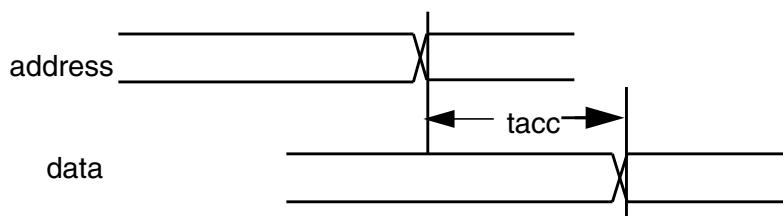
pin(02){
    direction : output;
    timing () {
        timing_type : falling_edge; /*sequential delay arc */
        intrinsic_rise : 3.15; /* t02d */
        rise_resistance : 0.020;
        intrinsic_fall : 3.00;
        fall_resistance : 0.017;
        related_pin : "clock";
    }
}

```

## 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 will always change (after some delay) when the input changes. [Figure 9-5](#) shows such a case.

*Figure 9-5 Timing Relationships for Through Combinational Paths*



[Example 9-5](#) shows the Library Compiler description of the relationships in [Figure 9-5](#).

*Example 9-5 Description of Timing Relationships in Figure 9-5*

```

bus(address) {
    direction : input;

```

```

        bus_type  : "bus10"
    }

bus(data){
    direction : output;
    bus_type  : "bus8"
    timing () {
        timing_sense : non_unate;
        intrinsic_rise : 8.15; /* tacc */
        rise_resistance : 0.020;
        intrinsic_fall : 7.00;
        fall_resistance : 0.017;
        related_pin : "address";
    }
}

```

## Interface Timing of a Complex Sequential Block

Figure 9-6 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 9-6 Complex Sequential Block Supported by the Interface Timing Specification*

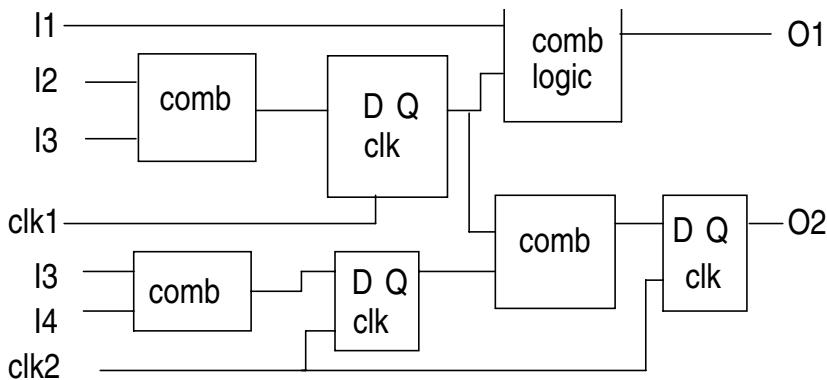
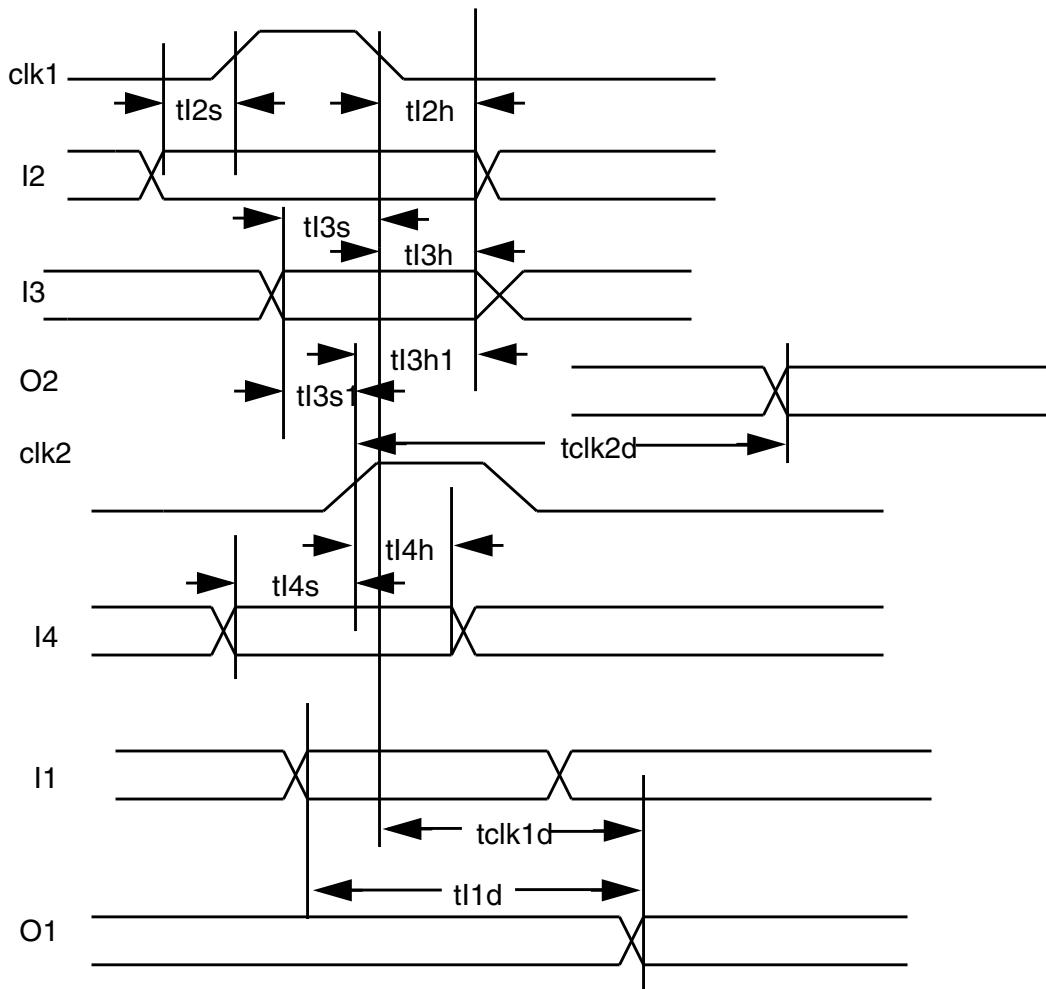


Figure 9-7 is the timing diagram illustrating the various interface relationships of this block.

- 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 9-7 Timing Relationships for the Circuit in Figure 9-6*



Example 9-6 shows the library description of the interface timing specification of the block shown in Figure 9-6. The CMOS delay model is used for this cell.

*Example 9-6 Description of the Timing Relationships in Figure 9-6*

```
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 () {
        timing_type : setup_rising;
        intrinsic_rise : 2.20; /* tI2s */
        intrinsic_fall : 2.20;
        related_pin : "clk1";
    }
    timing () {
        timing_type : hold_falling;
        intrinsic_rise : 0.95; /* tI2h */
        intrinsic_fall : 0.95;
        related_pin : "clk1";
    }
}
pin (I3) {
    direction : input;
    capacitance : 1.46;
    fanout_load : 1.46;
    timing () {
        timing_type : setup_falling;
        intrinsic_rise : 1.20; /* tI3s */
        intrinsic_fall : 1.20;
        related_pin : "clk1";
    }
    timing () {
        timing_type : hold_falling;
        intrinsic_rise : 1.95; /* tI3h */
        intrinsic_fall : 1.95;
        related_pin : "clk1";
    }
    timing () {
        timing_type : setup_rising;
        intrinsic_rise : 0.70; /* tI3s1 */
        intrinsic_fall : 0.70;
        related_pin : "clk2";
    }
    timing () {
        timing_type : hold_rising;
        intrinsic_rise : 2.95; /* tI3h1 */
        intrinsic_fall : 2.95;
        related_pin : "clk2";
    }
}

```

```

}

pin (I4) {
    direction : input;
    capacitance : 1.46;
    fanout_load : 1.46;
    timing () {
        timing_type : setup_rising;
        intrinsic_rise : 1.70; /* tI2s */
        intrinsic_fall : 1.70;
        related_pin : "clk2";
    }
    timing () {
        timing_type : hold_rising;
        intrinsic_rise : 0.80; /* tI2h */
        intrinsic_fall : 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 () {
        timing_sense : non_unate; /* combinational delay */
        intrinsic_rise : 3.25; /* tI1d */
        rise_resistance : 0.020;
        intrinsic_fall : 3.50;
        fall_resistance : 0.017;
        related_pin : "I1";
    }
    timing () {
        timing_type : falling_edge;

```

```

        /* sequential delay arc */
        intrinsic_rise : 1.25; /* tclk1d */
        rise_resistance : 0.020;
        intrinsic_fall : 1.50;
        fall_resistance : 0.017;
        related_pin : "clk1";
    }
}
pin (O2) {
    direction : output;
    timing () {
        timing_type : rising_edge;
        /*sequential delay arc */
        intrinsic_rise : 2.25; /* tclk2d */
        rise_resistance : 0.020;
        intrinsic_fall : 2.57;
        fall_resistance : 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. Refer to 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

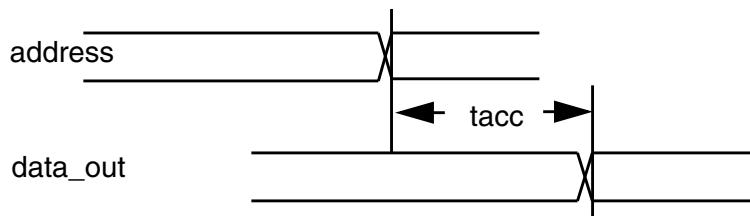
Synthesis tools do not observe other types of sequential arcs (such as clear, preset, recovery, and removal).

---

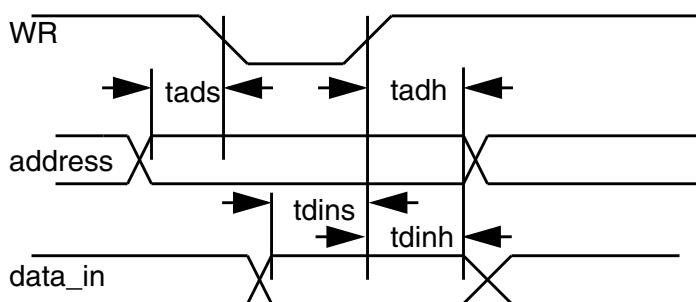
## Examples Using Interface Timing Specification

The following example shows how to specify the interface timing of a typical RAM, using the interface timing specification. [Figure 9-8](#) and [Figure 9-9](#) show the timing relationships of the RAM read and write cycles.

*Figure 9-8 Read Cycle Timing of the RAM (WR = 1)*



*Figure 9-9 Write Cycle Timing of the RAM (WR = 0)*



[Example 9-7](#) 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 9-7 Library Compiler Model of RAM in Figure 9-8 and Figure 9-9*

```
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 () {
        timing_type : setup_falling;
        intrinsic_rise : 3.20; /* tads */
        intrinsic_fall : 3.20;
        related_pin : "WR";
    }
    timing () {
        timing_type : hold_rising;
        intrinsic_rise : 1.85; /* tadh */
        intrinsic_fall : 1.85;
        related_pin : "WR";
    }
}
bus (data_in) {
    bus_type : "bus8"
    direction : input;
    capacitance : 1.46;
    fanout_load : 1.46;
    timing () {
        timing_type : setup_rising;
        intrinsic_rise : 1.50; /* tdis */
        intrinsic_fall : 1.50;
        related_pin : "WR";
    }
    timing () {
        timing_type : hold_rising;
        intrinsic_rise : 0.65; /* tdih */
        intrinsic_fall : 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 () {
    timing_sense : non_unate; /* combinational delay */
    intrinsic_rise : 5.25;      /* tacc*/
    rise_resistance : 0.020;
    intrinsic_fall : 5.50;
    fall_resistance : 0.017;
    related_bus_pin : "address";
}
}
} /* end cell */

```

**Example 9-8** shows a memory model in the interface timing specification format that contains modes.

*Example 9-8 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("D01") {
    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() {
            .....
        }
    }
}
```



# 10

## Modeling Noise

---

This chapter provides an overview of modeling noise to support gate-level static noise analysis. It covers various topics on modeling noise for calculation, detection, and propagation, in the following sections:

- Modeling Noise Terminology
- Modeling Cells for Noise
- Representing Noise Calculation Information
- Representing Noise Immunity Information
- Representing Propagated Noise Information
- Examples of Modeling Noise
- Checking Noise Library Data

---

## Modeling Noise Terminology

A net can be either an aggressor or a victim:

- An aggressor net is a net that injects noise onto a victim net.
- A victim net is a net onto which noise is injected by one or more neighboring nets through the cross-coupling capacitors between the nets.

Noise effect can be categorized in two ways:

- Delay noise
- Functional noise

Delay noise occurs when victim and aggressor nets switch simultaneously. This activity alters the delay and slew of the victim net.

Functional noise occurs when a victim net is intended to be at a stable value and the noise injected onto this net causes it to glitch. The glitch might propagate to a state element, such as a latch, altering the circuit state and causing a functional failure.

For a gate-level static analysis tool to compute and detect any delay or functional noise failure, the following are calculated:

- Noise calculation
- Noise immunity
- Noise propagation

---

## **Noise Calculation**

Coupled noise is the noise voltage induced at the output of nonswitching gates when coupled adjacent drivers to the output (aggressor drivers) are switching.

---

## **Noise Immunity**

The main concept of noise immunity is that for most cells, a glitch on the input pin has to be greater than a certain fixed voltage to cause a failure. However, a glitch with a tall height might still not cause any failure if the glitch width is very small. This is mainly because noise failure is related to input noise glitch energy and this energy is proportional to the area under the glitch waveform.

For example, if a large voltage glitch in terms of height and width occurs on the clock pin of a flip-flop, the glitch can cause a change in the data and therefore the flip-flop output might change.

---

## **Noise Propagation**

Propagated noise is the noise waveform created at the output of nonswitching gates due to the propagation of noise from the inputs of the same gate.

---

## Modeling Cells for Noise

Library information for noise can be characterized in the following ways:

- [I-V Characteristics and Drive Resistance](#)
  - [Noise Immunity](#)
  - [Noise Propagation](#)
- 

### I-V Characteristics and Drive Resistance

To calculate the coupled noise glitch on a victim net, you need to know the effective steady-state drive resistance of the net.

[Figure 10-1 on page 10-5](#) shows the four different types of noise glitch:

- V<sub>h+</sub>: Input is high, and the noise is over the high voltage rail.
- V<sub>h-</sub>: Input is high, and the noise is below the high voltage rail.
- V<sub>l+</sub>: Input is low, and the noise is over the low voltage rail.
- V<sub>l-</sub>: Input is low, and the noise is below the low voltage rail.

Because the current is a nonlinear function of the voltage, you need to characterize the steady-state I-V characteristics curve, which provides a more accurate view of the behavior of a cell in its steady state. This information is specified for every timing arc of the cell that can propagate a transition. If an I-V curve cannot be obtained for a specific arc, the steady-state drive resistance single value can be used, but it is less accurate than the I-V curve.

Figure 10-1 Noise Glitch and Steady-State Drive Resistance

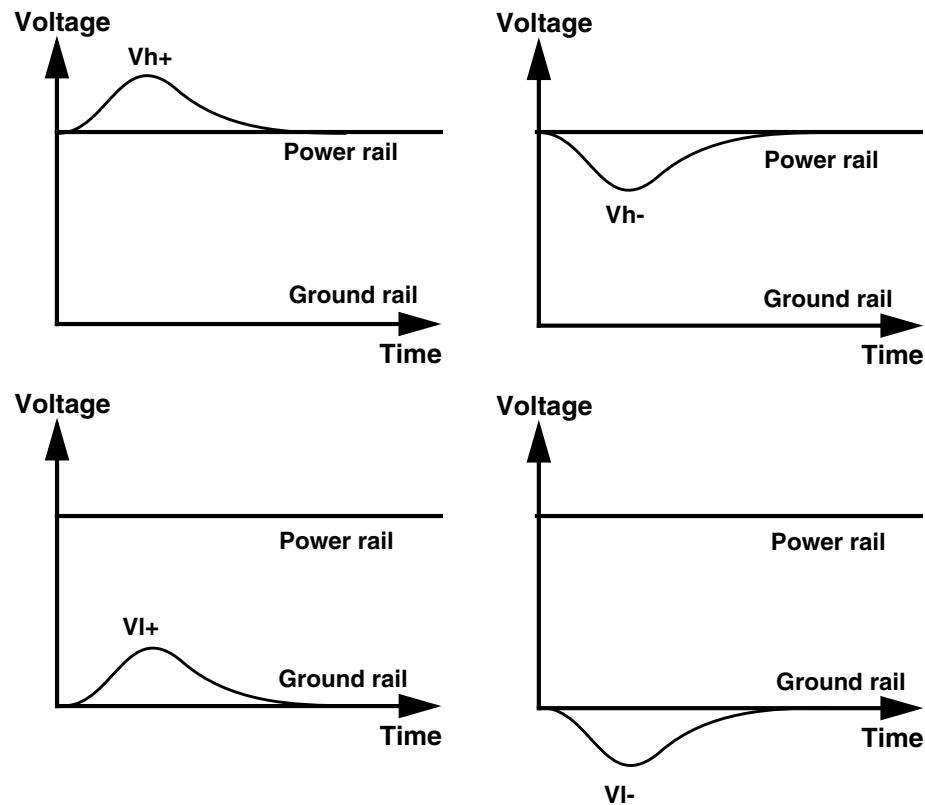
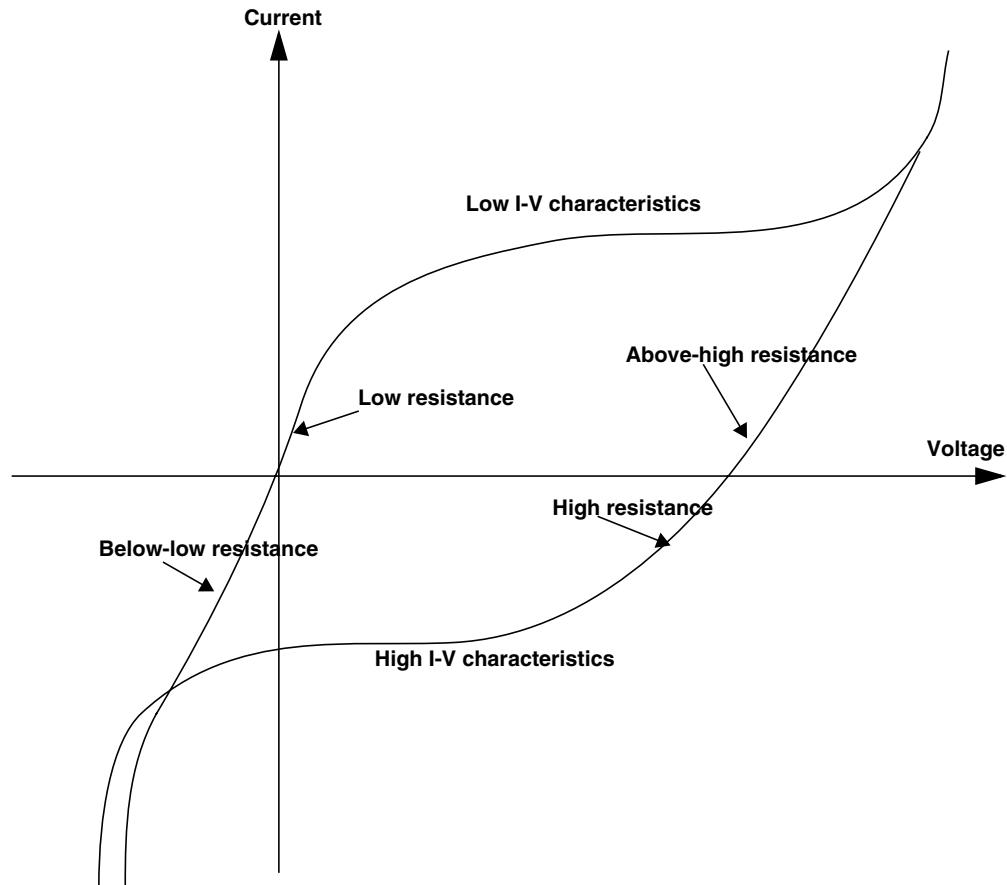


Figure 10-2 on page 10-6 is an example of two I-V curves and the steady-state resistance value.

*Figure 10-2 I-V Characteristics and Steady-State Drive Resistance*

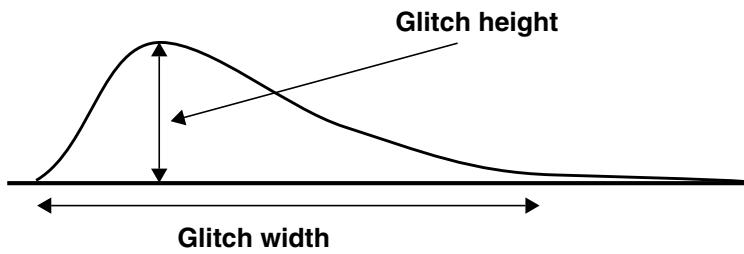


---

## Noise Immunity

Circuits can tolerate large glitches at their inputs and still work correctly if the glitches deliver only a small energy. Given this concept, each cell input can be characterized by application of a wide range of coupling voltage waveform stimuli on it. [Figure 10-3](#) shows a glitch noise model.

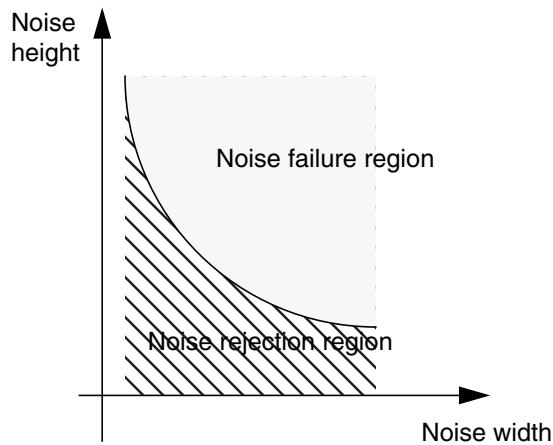
*Figure 10-3 Glitch Noise Modeling*



One method of modeling the noise immunity curve involves applying coupling voltage waveform stimuli with various heights (in library voltage units) and widths (in library time units) to the cell input, and then observing the output voltage waveform. The exact set of input stimuli (in terms of height and width) that produces an output noise voltage height equal to a predefined voltage is on the noise immunity curve. This predefined voltage is known as the *cell failure voltage*. Any input stimulus that has a height and width above the noise immunity curve causes a noise voltage higher than the cell failure voltage at the output and produces a functional failure in the cell.

Figure 10-4 shows an example of a noise immunity curve.

*Figure 10-4 Noise Immunity Curve*

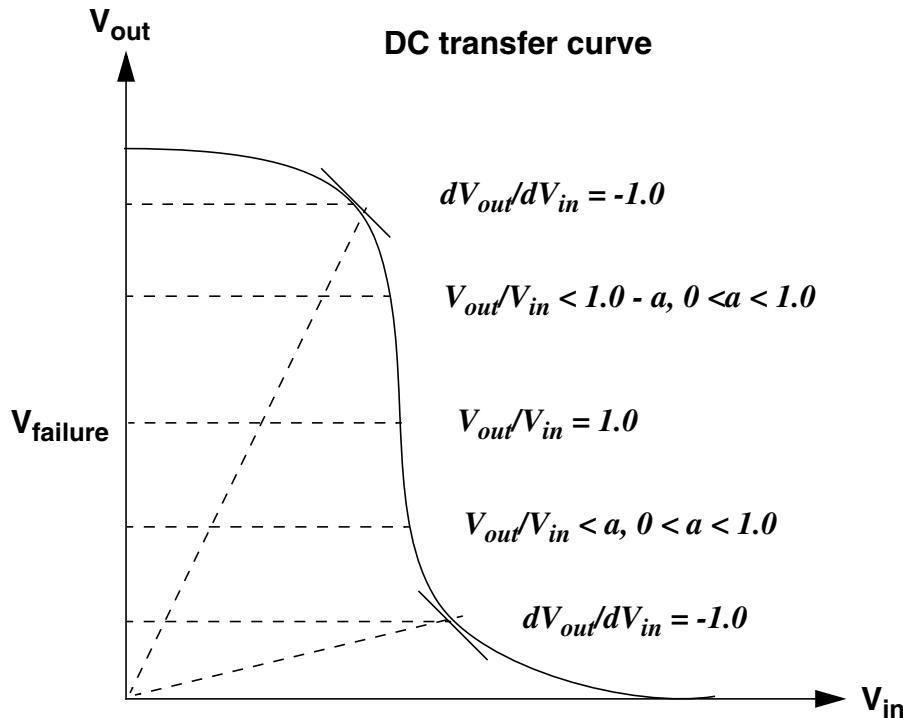


As shown in [Figure 10-4](#), any noise width and height combination that falls above the noise rejection curve causes functional failure.

The selection of cell failure voltage is important for noise immunity curve characterization. There are many ways to select a failure voltage for a cell that produces usable noise immunity curves, including the following:

- $V_{failure}$  equal to the output DC noise margin
- $V_{failure}$  equal to the next cell's  $V_{IL}$  or  $(V_{cc} - V_{IH})$
- $V_{failure}$  corresponding to the point on the DC transfer curve where  $dV_{out}/dV_{in}$  is 1.0 or -1.0
- $V_{failure}$  corresponding to the point on the DC transfer curve where  $V_{out}/V_{in}$  is less than 1.0 or -1.0
- $V_{failure}$  corresponding to the point on the DC transfer curve where  $V_{out}/V_{in}$  is 1.0 or -1.0

*Figure 10-5 Different Failure Voltage Criteria for Noise Immunity Curve*



The noise immunity curve can also be a function of output loads, where cells with larger output loads can tolerate greater input noise.

The noise immunity curve is also state-dependent. For example, the noise on the A-to-Z arc of an XOR gate when B = 0 might be different from the B-to-Z arc when B = 1, because the arcs might go through different sets of transistors.

---

## Using the Hyperbolic Model

The noise immunity curve resembles a hyperbola, because the area of different noise along the hyperbola is constant. Therefore noise immunity can be defined as a hyperbolic function with only three coefficients for every input on an I/O library pin. The formula for the height based on these three coefficients is as follows:

$$\text{height} = \text{height\_coefficient} + \text{area\_coefficient} / (\text{width} - \text{width\_coefficient});$$

Your tool (such as, PrimeTime SI) gets these coefficients from the library and applies the calculated height and width to determine whether the noise can cause functional failure. Any point above the hyperbolic curve signifies a functional failure.

---

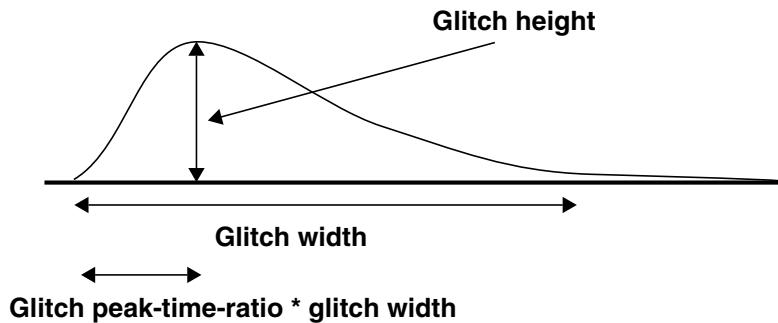
## Noise Propagation

Propagated noise from the input to the output of a cell is modeled by

- Output glitch height
- Output glitch width
- Output glitch peak time ratio
- Output load

[Figure 10-6](#) illustrates basic noise characteristics.

*Figure 10-6 Basic Noise Characteristics*



The output noise width, height, and peak-time ratio depend on the input noise width, height, and peak-time ratio as well as on the output load. However, in some cases, the dependency on peak-time ratio can be negligible; therefore, to reduce the amount of data, the lookup table does not have a peak-time-ratio dependency.

**Table 10-1** shows a summary of the syntax used to model cases when the cell is not switching.

*Table 10-1 Summary of Library Requirements for Noise Model*

Category		Model type	Description
Noise detection	Voltage ranges (DC noise margin)	Lookup table and polynomial	input_voltage/output_voltage defined for all library pins
	Hyperbolic noise immunity curves	Lookup table and polynomial	Four hyperbolic curves; each has three coefficients, defined for input or bidirectional library pins
	Noise immunity tables	Lookup table	Four tables indexed by noise width and output load defined for timing arcs
	Noise immunity polynomials	Polynomial	Four polynomials as a function of noise width and output load defined for timing arcs

*Table 10-1 Summary of Library Requirements for Noise Model (Continued)*

Category		Model type	Description
Noise calculation	Steady-state resistances	Lookup table and polynomial	Four floating-point values defined for timing arcs
	I-V characteristics tables	Lookup table	Two tables indexed by output steady-state voltage for non-three-state arcs and one table for three-state arcs
	I-V characteristics polynomials	Polynomial	Two polynomials as a function of output steady-state voltage for non-three-state arcs and one table for three-state arcs
Noise propagation	Noise propagation tables	Lookup table	Four pairs of noise width and height tables, each indexed by noise width, height, and load
	Noise propagation polynomials	Polynomial	Four sets of three polynomials (width, height, and peak-time ratio), each a function of width, height, peak-time ratio, and load

## Representing Noise Calculation Information

In the Library Compiler syntax, you can represent coupled noise information with an I-V characteristics lookup table model or polynomial model at the timing level or four simple attributes defined at the timing level:

- `steady_state_resistance_above_high`
- `steady_state_resistance_below_low`

- steady\_state\_resistance\_high
  - steady\_state\_resistance\_low
- 

## I-V Characteristics Lookup Table Model

You can describe I-V characteristics in your libraries by using lookup tables. To define your lookup tables, use the following groups and attributes:

- The iv\_lut\_template group in the library group
- The steady\_state\_current\_high, steady\_state\_current\_low, and steady\_state\_current\_tristate groups in the timing group

### **iv\_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 I-V output voltage and the breakpoints for the axis. Assign each template a name. Make the template name the group name of a steady\_state\_current\_low group, steady\_state\_current\_high group, or steady\_state\_current\_tristate group.

### **Syntax**

```
library(name_string) {
    ...
    iv_lut_template(template_name_string) {
        variable_1: iv_output_voltage ;
        index_1 ("float,..., float");
    }
}
```

```
    }  
    ...  
}
```

## Template Variables

To specify I-V characteristics, define the following variable and index:

`variable_1`

The only valid value is `iv_output_voltage`, which specifies the I-V voltage of the output pin specified in the `pin` group. The voltage is measured from the pin to the ground.

`index_1`

The index values are a list of floating-point numbers that can be negative or positive. The values in the list must be in increasing order. The number of floating-point numbers in the `index_1` variable determines the dimension.

## Example

```
iv_lut_template(my_current_low) {  
    variable_1: iv_output_voltage;  
    index_1 ("-1, -0.1, 0, 0.1 0.8, 1.6, 2");  
}  
iv_lut_template(my_current_high) {  
    variable_1 : iv_output_voltage;  
    index_1("-1, 0, 0.3, 0.5, 0.8, 1.5, 1.6, 1.7, 2");  
}
```

---

## Defining the Lookup Table Steady-State Current Groups

To specify the I-V characteristics curve for the nonlinear table model, use the `steady_state_current_high`, `steady_state_current_low`, or `steady_state_current_tristate` groups within the `timing` group.

### Syntax for Table Model

```
timing() { /* for non-three-state arcs */
    steady_state_current_high(template_name_string) {
        values("float,..., float");
    }
    steady_state_current_low(template_name_string) {
        values("float,..., float");
    }
    ...
}

timing() { /* for three-state arcs */
    steady_state_current_tristate(template_name_string) {
        values("float,..., float");
    }
}
...
```

*float*

The values are floating-point numbers indicating values for current.

The following rules apply to lookup table groups:

- Each table must have an associated name for the `iv_lut_template` it uses. The name of the template must be identical to the name defined in a library `iv_lut_template` group.

- You can overwrite `index_1` in a lookup table, but the overwrite must come before the definition of values.
- The current values of the table are stored in a `values` attribute. The values can be negative.

## Example

```
timing() {
    ...
    steady_state_current_low(my_current_low) {
        values("-0.1, -0.05, 0, 0.1, 0.25, 1, 1.8");
    }
    steady_state_current_high(my_current_high) {
        values("-2, -1.8, -1.7, -1.4, -1, -0.5, 0, 0.1, 0.8");
    }
}
```

## I-V Characteristics Curve Polynomial Model

As with the lookup table model, you can describe an I-V characteristics curve in your libraries by using the polynomial representation. To define your polynomial, use the following groups and attributes:

- The `poly_template` group in the `library` group
- The `steady_state_current_high`, `steady_state_current_low`, and `steady_state_current_tristate` groups within the `timing` group

## **poly\_template Group**

You can define a `poly_template` group at the library level to specify the equation variables, the variable ranges, the voltages mapping, and the piecewise data. The valid values for the variables are extended to include `iv_output_voltage`, `voltage`, `voltage $i$` , and `temperature`.

### **Syntax**

```
library(name_string) {
    ...
    poly_template(template_name_string) {
        variables(variable_1_enum, ..., variable_n_enum);
        variable_i_range: (float, float);
        ...
        variable_n_range: (float, float);
        mapping(voltage_enum, power_rail_id);
        domain(domain_name_string) {
            variable_i_range: (float, float);
            ...
            variable_n_range: (float, float);
        }
        ...
    }
    ...
}
```

The syntax of the `poly_template` group is the same as that used for the delay model, except that the variables used in the format are

- `iv_output_voltage` for the output voltage of the pin
- `voltage`, `voltage $i$` , `temperature`

The piecewise model through the `domain` group is also supported.

For the complete poly\_template group syntax description, see the *Library Compiler Reference Manual: Technology and Symbol Libraries*.

## Example

```
poly_template ( my_current_low ) {
    variables ( iv_output_voltage, voltage,
voltage1,temperature );
    mapping(voltage1, VDD2);
    variable_1_range (-1, 2);
    variable_2_range (1.4, 1.8);
    variable_3_range (1.1, 1.5);
    variable_4_range (-40, 125);
}
}
```

---

## Defining Polynomial Steady-State Current Groups

To specify the I-V characteristics curve to define the polynomial, use the steady\_state\_current\_high, steady\_state\_current\_low, and steady\_state\_current\_tristate groups within the timing group.

## Syntax for Polynomial Model

```
timing { /* for non-three-state arcs */
    steady_state_current_high(template_name_string) {
        orders("integer,..., integer");
        coefs("float,..., float");
        domain(domain_name_string) {
            orders("integer,..., integer");
            coefs("float,..., float");
        }
        ...
    }
    steady_state_current_low(template_name_string) {
        ...
    }
}
```

```

timing() /* for three-state arcs */
steady_state_current_tristate(template_name_string) {
    ...
}
...
}

```

The orders, coefs, and variable\_range attributes represent the polynomial for the current for high, low, and three-state. The syntax for orders and coefs is the same as that described in the *Library Compiler Reference Manual: Technology and Symbol Libraries*.

The output voltage, temperature, and any power rail of the cell are allowed as variables for steady\_state\_current groups.

## Example

```

timing() {
    steady_state_current_low(my_current_low) {
        orders ("3, 3, 0, 0");
        coefs ("8.4165, 0.3198, -0.0004, 0.0000, \
                1133.8274, 8.7287, -0.0054, 0.0000, \
                139.8645, -60.3898, 0.0589, -0.0000, \
                -167.4473, 95.7112, -0.1018, 0.0000");
    }
    steady_state_current_high(my_current_high) {
        orders ("3, 3, 0, 0");
        coefs ("10.9165, 0.2198, -0.0003, 0.0000, \
                1433.8274, 8.7287, -0.0054, 0.0000, \
                128.8645, -60.3898, 0.0589, -0.0000, \
                -167.4473, 95.7112, -0.1018, 0.0000");
    }
    ...
}

```

---

## Using Steady-State Resistance Simple Attributes

To represent steady-state drive resistance values, use the following attributes to define the four regions:

- `steady_state_resistance_above_high`
- `steady_state_resistance_below_low`
- `steady_state_resistance_high`
- `steady_state_resistance_low`

These attributes are defined within the `timing` group to represent the steady-state drive resistance. If one of these attributes is missing, Library Compiler will not flag it, but the model will be inaccurate.

### Syntax

```
pin(name) {  
    ...  
    timing() {  
        ...  
        steady_state_resistance_above_high : float;  
        steady_state_resistance_below_low : float;  
        steady_state_resistance_high : float;  
        steady_state_resistance_low : float;  
        ...  
    }  
}
```

*float*

The value of steady-state resistance for the four different noise regions in the I-V curve.

### Example

```
steady_state_resistance_above_high : 200.0;  
steady_state_resistance_below_low : 100.0;
```

```
steady_state_resistance_high : 100.0;  
steady_state_resistance_low : 1100.0
```

---

## Using I-V Curves and Steady-State Resistance for tied\_off Cells

In tied-off cells, the output pins are tied to either high or low and there is no need to define timing information for related pins. The tied-off cells have been enhanced to accept I-V curve and steady-state resistance in the `timing` group. PrimeTime SI uses these timing groups for noise data access, and no timing data is requested. However, to specify only the noise data (I-V curves and steady-state resistance) in the `timing` group, you must specify a new Boolean attribute, `tied_off`, and set it to true.

---

## Defining tied\_off Attribute Usage

You can specify the I-V characteristics and steady-state drive resistance values on tied-off cells by using the `tied_off` attribute in the `timing` group.

### Syntax

```
pin(name) {  
    ...  
    timing() {  
        ...  
        tied_off : boolean;  
        /* timing type is not defined */  
        /* steady-state resistance */
```

The following rules apply to `tied_off` cells:

- Steady-state resistance and I-V curves can coexist in the same timing arc of a `tied_off` output pin.

- If the output pin is tied to low (function : "0") and its timing arc specifies the steady\_state\_current\_high group, Library Compiler issues an LBDB-625 error message to alert you that the output pin "high" has a tied-off timing arc containing an invalid steady\_state\_current\_low group. Similarly if the output pin is tied to high (function : "1") and its timing arc specifies the steady\_state\_current\_low group, Library Compiler issues an LBDB-625 error message to alert you that the output pin "low" has a tied-off timing arc containing an invalid steady\_state\_current\_high group.
- If noise immunity and noise propagation are specified in the timing arcs of a tied\_off pin, Library Compiler issues an LBDB-626 error message to alert you that the tied-off timing arc contains an invalid noise\_immunity or noise\_propagation group.
- If the related\_pin attribute is specified on a tied\_off output pin, Library Compiler issues an LBDB-635 error message to alert you that the tied-off timing arc contains an illegal related\_pin attribute.

## Example

```

pin (high) {
    direction : output;
    capacitance : 0;
    function : "1";

    /* noise information */
    timing() {
        tied_off : true;
        steady_state_resistance_high : 1.22;
        steady_state_resistance_above_high : 1.00;
        steady_state_current_high(iv1x5){
            index_1("0.3,0.75,1.0,1.2,2");
            values("-513.2,-447.9,-359.3,-245.7,497.3");
        }
    }
}

```

---

## Representing Noise Immunity Information

In the Library Compiler syntax, you can represent noise immunity information with a

- Lookup table or a polynomial model at the timing level
- Input noise width range at the pin level
- Hyperbolic model at the pin level

---

### Noise Immunity Lookup Table Model

You can represent noise immunity in your libraries by using lookup tables. To define your lookup tables, use the following groups and attributes:

- `noise_lut_template` group in the `library` group
- `noise_immunity_above_high`,  
`noise_immunity_above_low`,  
`noise_immunity_below_low`, and `noise_immunity_high` groups in the `timing` group

### **`noise_lut_template` Group**

Use this library-level group to create templates of common information that multiple noise immunity lookup tables can use.

A table template specifies the input noise width, the output load, and their corresponding breakpoints for the axis. Assign each template a name, and make the name the group name of a noise immunity group.

## Syntax

```
library(name_string) {  
    ...  
    noise_lut_template(template_name_string) {  
        variable_1: value;  
        variable_2: value;  
  
        index_1 ("float, ..., float");  
        index_2 ("float, ..., float");  
    }  
    ...  
}
```

## Template Variables

The library-level table template specifying noise immunity can have two variables (`variable_1` and `variable_2`). The variables indicate the parameters used to index the lookup table along the first and second table axes. The parameters are `input_noise_width` and `total_output_net_capacitance`.

The index values in `index_1` and `index_2` are a list of positive floating-point numbers. The values in the list must be in increasing order.

The unit for the input noise width is the library time unit.

## Example

```
noise_lut_template(my_noise_reject) {  
    variable_1 : input_noise_width;  
    variable_2 : total_output_net_capacitance;  
    index_1("0, 0.1, 0.3, 1, 2");  
    index_2("1, 2, 3, 4, 5");  
}
```

---

## Defining the Noise Immunity Table Groups

To represent noise immunity, use the noise\_immunity\_above\_high, noise\_immunity\_below\_low, noise\_immunity\_high, and noise\_immunity\_low groups within the timing group.

### Syntax for Noise Immunity Table Model

```
timing() {
    noise_immunity_above_high(template_name_string) {
        index_1 ("float,..., float");
        index_2 ("float,..., float");
        values("float,...,float"..."float,...,float");
    }
    noise_immunity_below_low(template_name_string) {
        ...
    }
    noise_immunity_high(template_name_string) {
        ...
    }
    noise_immunity_low(template_name_string) {
        ...
    }
}
```

The following rules apply to the noise immunity groups:

- These tables are optional, and each of them can exist separately on the library timing arcs.
- Each noise immunity table has an associated name for the noise\_lut\_template it uses. The name of the table must be identical to the name defined in a library noise\_lut\_template group.

- Each table is two-dimensional. The indexes are `input_noise_width` and `total_output_net_capacitance`. The values in the table are the noise heights (that is, height as a function of width and output load).
- You can overwrite any or both indexes in a noise template. However, the overwrite must occur before the actual definition of the values.
- The height values of the table are stored in the `values` attribute. Each height value is the absolute difference of the noise bump height voltage and the related rail voltage and is, therefore, a positive number. Any point over this curve describes a height/width combination that causes functional failure.
- The unit for the height is the library voltage unit.
- For points outside table ranges, extrapolation is used.

## Example

```
pin ( Y ) {
    ...
    timing () {
        noise_immunity_below_low (my_noisel) {
            values ("1, 0.8, 0.5", \
                    "1, 0.8, 0.5", \
                    "1, 0.8, 0.5");
        }
        noise_immunity_above_high (my_noisel) {
            values ("1, 0.8, 0.5", \
                    "1, 0.8, 0.5", \
                    "1, 0.8, 0.5");
        }
    }
}
```

---

## Noise Immunity Polynomial Model

As with the lookup table model, you can represent noise immunity in your libraries by using the polynomial representation. To define your polynomial, use the following groups and attributes:

- The `poly_template` group in the library group
- The `noise_immunity_above_high`,  
`noise_immunity_below_low`, `noise_immunity_high`,  
and `noise_immunity_low` groups in the timing group

### **poly\_template Group**

You can define a `poly_template` group at the library level to specify the polynomial equation variables, the variable ranges, the voltage mapping, and the piecewise data. The valid values for the variables include `total_output_net_capacitance`, `input_noise_width`, `voltage`, `voltagei`, and `temperature`.

### **Syntax**

```
library(name_string) {  
    ...  
    poly_template(template_name_string) {  
        variables(variable_i enum, ..., variable_n enum);  
        variable_i_range: (float, float);  
        ...  
        variable_n_range: (float, float);  
        mapping(voltage enum, power_rail id);  
        domain(domain_name_string); {  
            variable_i_range: (float, float);  
            ...  
            variable_n_range: (float, float);  
        }  
    }  
    ...  
}
```

## Template Variables

The syntax of the `poly_template` group is the same as that used for the delay model, except that the variables used in the format are

- `input_noise_width`
- `total_output_net_capacitance`
- `voltage, voltage $i$ , temperature`

The piecewise model through the `domain` group is also supported.

For the complete `poly_template` group syntax description, see the *Library Compiler Reference Manual: Technology and Symbol Libraries*.

## Example

```
poly_template (my_noise_reject) { /* existing syntax */
    variables (input_noise_width,voltage,voltage1, \
               temperature,
               total_output_net_capacitance);
    mapping(voltage1, VDD2);
    variable_1_range (0, 2);
    variable_2_range (1.4, 1.8);
    variable_3_range (1.1, 1.5);
    variable_4_range (-40, 125);
    variable_5_range (0.01, 1.0);
    domain (typ) {
        variables (input_noise_width,voltage,voltage1, \
                   temperature,total_output_net_capacitance);
        variable_1_range (0, 2);
    }
}
```

## Defining the Noise Immunity Polynomial Groups

To represent noise immunity, use the `noise_immunity_above_high`, `noise_immunity_below_low`, `noise_immunity_high`, and `noise_immunity_low` groups within the `timing` group.

### Syntax

```
...
timing() {
    noise_immunity_above_high(template_name_string) {
        orders("integer,..., integer");
        coefs("float,..., float");
        ...
        domain(domain_name_string) {
            orders("integer,..., integer");
            coefs("float,..., float");
        }
        ...
    }
    noise_immunity_below_low(template_name_string) {
        ...
    }
    noise_immunity_high(template_name_string) {
        ...
    }
    noise_immunity_low(template_name_string) {
        ...
    }
    ...
}
```

The syntax for `poly_template`, `orders`, and `coefs` is the same as that described in the *Library Compiler User Guide: Modeling Timing and Power Technology Libraries*. Because the polynomial model is a superset of the lookup table model, all syntax supported in the lookup table is also supported in the polynomial model. For

example, you can have a polynomial `noise_immunity_high` and a table `noise_immunity_low` defined in the same group in a scalable polynomial delay model library.

## Example

```
noise_immunity_low (my_noise_reject) {
    domain (typ) {
        orders ("1, 1, 1, 1, 1")
        coefs ("1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, \
                1.0, 1.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, \
                1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0");
    domain (min) {
        orders("1 3 1 1");
        coefs("-0.01, 0.02, 1.41, -0.54, \
               1.85, 1.83, -5.58, -2.96, -0.0001, \
               0.0001, -0.002, -0.0019, 0.002, \
               0.0012, -0.010, -0.0061, 0.034, \
               0.015, 2.08, -0.22, 4.13, 2.44, \
               -14.02, -7.83, 7.09e-05, -1.98e-05, \
               -0.0019, 0.0009, 0.0065, -0.0004, \
               -0.027, -0.016");
    }
}
```

---

## Input Noise Width Ranges at the Pin Level

So that the PrimeTime S/I tool can identify whether a noise immunity or propagation table is referenced within the noise range indexes, the Library Compiler syntax allows you to specify the minimum and maximum values of the input noise width.

## Defining the `input_noise_width` Range Limits

You can specify two float attributes, `min_input_noise_width` and `max_input_noise_width`, at the pin level. These attributes are optional and specify the minimum and maximum values of the input noise width.

## Syntax

```
pin(name_string) {  
    ...  
    /* used for noise immunity or propagation */  
  
    min_input_noise_width : float;  
    max_input_noise_width: float;  
    ...  
}  
  
float
```

The values of `min_input_noise_width` and `max_input_noise_width` are the minimum and maximum input noise width, in library time units.

The following rules apply to `input_noise_width` range limits:

- The `min_input_noise_width` and `max_input_noise_width` attributes can be defined only on input or inout pins. Otherwise, Library Compiler issues an LBDB-619 error message to alert you that the `min_input_noise_width` and `max_input_noise_width` attributes cannot be specified on an output pin.
- The `min_input_noise_width` and `max_input_noise_width` attributes must both be defined. Otherwise, Library Compiler issues an LBDB-622 error message to alert you that the `max_input_noise_width` (or `min_input_noise_width`) attribute is missing for pin A.
- Library Compiler checks whether the `min_input_noise_width <= max_input_noise_width` constraints are met. If the constraints aren't met, Library Compiler issues an LBDB-623 error message to alert you that the `min_input_noise_width` attribute value is larger than the `max_input_noise_width` attribute value for pin A.

- Library Compiler does not check whether the specification of these noise range attributes is associated with noise groups.

## Example

```

pin ( O) {
    direction : output ; /* existing syntax */
    capacitance : 1 ; /* existing syntax */
    fanout_load : 1 ; /* existing syntax */

    /* Noise range */
    min_input_noise_width : 0.0;
    max_input_noise_width : 2.0;

    /* Timing group defines what is acceptable noise on input pins. */

    timing () {
        /* Noise immunity.
        * Defines maximum allowed noise height for given pulse width.
        * Pulse height is absolute value from the signal level.
        * Any of the following four tables are optional. */

        noise_immunity_low (my_noise_reject) {
            values ("1.5, 0.9, 0.8, 0.65, 0.6");
        }
        noise_immunity_high (my_noise_reject) {
            values ("1.3, 0.8, 0.7, 0.6, 0.55");
        }
        noise_immunity_below_low (my_noise_reject_outside_rail) {
            values ("1, 0.8, 0.5");
        }
        noise_immunity_above_high (my_noise_reject_outside_rail) {
            values ("1, 0.8, 0.5");
        }
    } /* end of timing group */
} /* end of pin group */

```

## Defining the Hyperbolic Noise Groups

To specify hyperbolic noise immunity information, use the `hyperbolic_noise_above_high`, `hyperbolic_noise_below_low`, `hyperbolic_noise_high`, and `hyperbolic_noise_low` groups within the pin group.

## Syntax

```
pin(name_string) {  
    ...  
    hyperbolic_noise_above_high() {  
        height_coefficient : float;  
        area_coefficient : float;  
        width_coefficient : float;  
    }  
    hyperbolic_noise_below_low() {  
        ...  
    }  
    hyperbolic_noise_high() {  
        ...  
    }  
    hyperbolic_noise_low() {  
        ...  
    }  
    ...  
}  
float
```

The coefficient values for height, width, and area must be 0 or a positive number.

The following rules apply to noise immunity groups:

- The hyperbolic noise groups are optional, and each can be defined separately from the other three.
- For the same region (above-high, below-low, high, or low), the hyperbolic noise groups can coexist with normal noise immunity tables.
- For different regions (above-high, below-low, high, or low), a combination of tables and hyperbolic functions is allowed. For example, you might have a hyperbolic function for below and above the rails and have tables for high and low tables on the same pin.

- When no table or hyperbolic function is defined for a given pin, the application checks other measures for noise immunity, such as DC noise margins.
- The unit for height and height\_coefficient is the library unit of voltage. The unit for width and width\_coefficient is the library unit of time. The unit for area\_coefficient is the library unit of voltage multiplied by the library unit of time.

### **Example**

```
hyperbolic_noise_low() {
    height_coefficient : 0.4;
    area_coefficient : 1.1;
    width_coefficient : 0.1;
}
hyperbolic_noise_high() {
    height_coefficient : 0.3;
    area_coefficient : 0.9;
    width_coefficient : 0.1;
}
```

## **Representing Propagated Noise Information**

In the Library Compiler syntax, you can represent propagated noise information at the timing level by using a

- Propagated Noise Lookup Table Model
- Propagated Noise Polynomial Model

---

## Propagated Noise Lookup Table Model

You can represent propagated noise in your libraries by using lookup tables. To define your lookup tables, use the following groups and attributes:

- propagation\_lut\_template group in the library group
- propagated\_noise\_height\_above\_high  
propagated\_noise\_height\_below\_low  
propagated\_noise\_height\_high  
propagated\_noise\_height\_low  
propagated\_noise\_width\_above\_high  
propagated\_noise\_width\_below\_low  
propagated\_noise\_width\_high  
propagated\_noise\_width\_low groups in the timing group

### propagation\_lut\_template Group

Use this library-level group to create templates of common information that multiple propagation lookup tables can use.

A table template specifies the propagated noise width, height, and output load and their corresponding breakpoints for the axis. Assign each template a name. Make the template name the group name of a propagated noise group.

### Syntax

```
library(name_string) {  
    ...  
    propagation_lut_template(template_name_string) {  
        variable_1: value;  
        variable_2: value;  
        variable_3: value;  
        index_1 ("float, ..., float");  
    }  
}
```

```

        index_2 ("float, ..., float");
        index_3 ("float, ..., float");
    }
    ...
}

```

## Template Variables

The table template specifying propagated noise can have three variables (variable\_1, variable\_2, and variable\_3). The variables indicate the parameters used to index the lookup table along the first, second, and third table axes. The parameters are input\_noise\_width, input\_noise\_height, and total\_output\_net\_capacitance.

The index values in the index\_1, index\_2, and index\_3 attributes are a list of positive floating-point numbers. The values in the list must be in increasing order.

The unit for input\_noise\_width and input\_noise\_height is the library time unit.

## Example

```

propagation_lut_template(my_propagated_noise) {
    variable_1 : input_noise_width;
    variable_2 : input_noise_height;
    variable_3 : total_output_net_capacitance;
    index_1("0.01, 0.2, 2");
    index_2("0.2, 0.8");
    index_3("0, 2");
}

```

## Defining the Propagated Noise Table Groups

To represent propagated noise, use the following groups within the timing group: propagated\_noise\_height\_above\_high, propagated\_noise\_height\_below\_low,

`propagated_noise_height_high,`  
`propagated_noise_height_low, and`  
`propagated_noise_width_above_high.`

## Syntax for Table Model

```
timing() {  
    ...  
    propagated_noise_height_above_high (temp_name_string) {  
        index_1 ("float,...., float");  
        index_2 ("float,...., float");  
        index_3 ("float,...., float");  
        values("float,...., float,"..."float,..., float");  
    }  
    propagated_noise_height_below_low (temp_name_string) {  
        ...  
    }  
    propagated_noise_width_above_high (temp_name_string) {  
        ...  
    }  
    propagated_noise_width_below_low (tempname_string) {  
        ...  
    }  
    propagated_noise_height_high(template_name_string) {  
        ...  
    }  
    propagated_noise_height_low(template_name_string) {  
        ...  
    }  
    propagated_noise_width_high(template_name_string) {  
        ...  
    }  
    propagated_noise_width_low(template_name_string) {  
        ...  
    }  
    ...  
}
```

The following rules apply to the propagation noise groups:

- Each of the three pairs of tables is optional; the assumption is that if one pair is missing, the corresponding region does not propagate any noise.
- If a pair of tables for a particular region (high, low, above-high, or below-low) is specified, both width and height must be specified.
- Each propagated noise table has an associated name for the propagation\_lut\_template it uses. The name of the table must be identical to the name defined in a library propagated\_noise\_template group.
- Each table can be two-dimensional or three-dimensional. The indexes are input\_noise\_width, input\_noise\_height, and total\_output\_net\_capacitance. The values are coefficients of height and width. The coefficient values for height and width must be 0 or a positive number.
- You can overwrite any or all indexes in a propagated noise template. However, the overwrite must occur before the actual definition of the values.
- The width and height values of the table are stored in the values attribute. Each height value is the absolute difference of the noise bump height voltage and the related rail voltage and is, therefore, a positive number. Any point over this curve describes a height/width combination that causes functional failure.
- The unit for all propagated height is the library voltage unit. The unit for all propagated width is the library unit of time.
- For points outside table ranges, extrapolation is used.

## Example

```
propagated_noise_width_high(my_propagated_noise) {  
    values ("0.01, 0.10, 0.15", "0.04, 0.14, 0.18", \  
            "0.05, 0.15, 0.24", "0.07, 0.17, 0.32");  
}  
propagated_noise_height_high(my_propagated_noise) {  
    values ("0.01, 0.20, 0.25", "0.04, 0.24, 0.28", \  
            "0.05, 0.25, 0.28", "0.07, 0.27, 0.35");  
}
```

---

## Propagated Noise Polynomial Model

As with the lookup table model, you can describe propagated noise in your libraries by using polynomial representation. To define your polynomial, use the following groups:

- The `poly_template` group in the `library` group
- The `propagated_noise_height_above_high`,  
`propagated_noise_height_below_low`,  
`propagated_noise_height_high`,  
`propagated_noise_height_low`,  
`propagated_noise_width_above_high`,  
`propagated_noise_width_below_low`,  
`propagated_noise_width_high`,  
`propagated_noise_width_low`,  
`propagated_noise_peak_time_ratio_above_high`,  
`propagated_noise_peak_time_ratio_below_low`,  
`propagated_noise_peak_time_ratio_high`, and  
`propagated_noise_peak_time_ratio_low` groups in the `timing` group

---

## **poly\_template Group**

You can define a `poly_template` group at the library level to specify the equation variables, the variable ranges, the voltage mapping, and the piecewise data. The valid values for the variables are extended to include `input_noise_width`, `input_noise_height`, `input_peak_time_ratio`, `total_output_net_capacitance`, `temperature`, and the related rail voltages.

### **Syntax**

```
library(name_string) {  
    ...  
    poly_template(template_name_string) {  
        variables(variable_i_enum, ..., variable_n_enum);  
        variable_i_range: (float, float);  
        ...  
        variable_n_range: (float, float);  
        mapping(voltage_enum, power_rail_id);  
        domain(domain_name_string) {  
            variable_i_range: (float, float);  
            ...  
            variable_n_range: (float, float);  
        }  
    }  
    ...  
}
```

### **Template Variables**

The syntax of the `poly_template` group is the same as that of the delay model, except that the variables used in the format are

- `input_noise_width`, `input_noise_height`, `input_peak_time_ratio`
- `total_output_net_capacitance`

- voltage, voltage*i*, temperature

The piecewise model through the domain group is also supported.

The syntax for poly\_template, orders, and coefs is the same as that described in *Library Compiler Reference Manual: Technology and Symbol Libraries*.

The input\_peak\_time\_ratio is always specified as a ratio of width, so it is a value between 0.0 and 1.0.

## Example

```
poly_template(my_propagated_noise) {
    variables (input_noise_width, input_noise_height, input_peak_time_ratio,
              total_output_net_capacitance);
    variable_1_range (0.01, 2);
    variable_2_range (0, 0.8);
    variable_3_range (0.0, 1.0);
    variable_4_range (0, 2);
} /* poly_template(propagated_noise) */
```

## Defining Propagated Noise Groups for Polynomial Representation

To specify polynomial representation, use the propagated\_noise\_height\_above\_high, propagated\_noise\_height\_below\_low, propagated\_noise\_height\_high, propagated\_noise\_height\_low, propagated\_noise\_width\_above\_high, propagated\_noise\_width\_below\_low, propagated\_noise\_width\_high, propagated\_noise\_width\_low, propagated\_noise\_peak\_time\_ratio\_above\_high, propagated\_noise\_peak\_time\_ratio\_below\_low,

`propagated_noise_peak_time_ratio_high`, and  
`propagated_noise_peak_time_ratio_low` groups within the  
timing group to define the polynomial.

The `peak_time_ratio` groups are supported only in the  
polynomial model.

## Syntax for Polynomial

```
timing() {
    ...
    propagated_noise_height_above_high (temp_name_string) {
        variable_i_range: (float, float);
        orders("integer,..., integer");
        coefs("float,..., float");
        domain(domain_name_string) {
            variable_i_range: (float, float);
            orders("integer,..., integer");
            coefs("float,..., float");
        }
        ...
    }
    propagated_noise_width_above_high (temp_name_string) {
        ...
    }
    propagated_noise_height_below_low (temp_name_string) {
        ...
    }
    propagated_noise_width_below_low (temp_name_string) {
        ...
    }
    propagated_noise_height_high (temp_name_string) {
        ...
    }
    propagated_noise_width_high (temp_name_string) {
        ...
    }
    propagated_noise_height_low (temp_name_string) {
        ...
    }
    propagated_noise_width_low (temp_name_string) {
```

```

    ...
}

propagated_noise_peak_time_ratio_above_high(
    temp_name_string) {

    ...
}

propagated_noise_peak_time_ratio_below_low(
    temp_name_string) {

    ...
}

propagated_noise_peak_time_ratio_high (temp_name_string) {

    ...
}

propagated_noise_peak_time_ratio_low (temp_name_string) {

    ...
}

...
}

```

The syntax for `poly_template`, `orders`, and `coefs` is the same as that described in the *Library Compiler User Guide: Modeling Timing and Power Technology Libraries*. Because the polynomial model is a superset of the lookup table model, all syntax supported in the lookup table is also supported in the polynomial model. For example, you can have a `propagated_noise_width_high` polynomial and a `propagated_noise_width_low` table defined in the same group in a scalable polynomial delay model library.

## Example

```

propagated_noise_width_high(my_propagated_noise) {
    orders("1, 1, 1, 1 ");
    coefs("1, 2, 3, 4 ,\
           1, 2, 3, 4 ,\
           1, 2, 3, 4 ,\
           1, 2, 3, 4 ");
}

propagated_noise_height_high(my_propagated_noise) {
    orders("1, 1, 1, 1 ");

```

```

        coefs("1, 2, 3, 4 ,\
               1, 2, 3, 4 ,\
               1, 2, 3, 4 ,\
               1, 2, 3, 4 ");
    }

```

## Examples of Modeling Noise

The examples in this section model libraries for noise extension for scalable polynomials and nonlinear lookup table model libraries.

### Scalable Polynomial Model Noise Sample

A scalable polynomial delay library allows you to describe how noise parameters vary with rail voltage and temperature.

```

library (my_noise_lib) {
    delay_model : "polynomial";
    time_unit : "1ns";
    voltage_unit : "1V";
    current_unit : "1mA";
    capacitive_load_unit (1,pf);
    pulling_resistance_unit : 1kohm;
    power_supply() {
        default_power_rail : VDD1;
        power_rail(VDD1, 1.6);
        power_rail(VDD2, 1.3);
    }
    nom_voltage : 1.0;
    nom_temperature : 40;
    nom_process : 1.0;
    /* Templates of DC noise margins and output levels */
    input_voltage(MY_CMOS_IN) {
        vil : 0.3;
        vih : 1.1;
        vimin : -0.3;
        vimax : VDD + 0.3;
    }
    output_voltage(MY_CMOS_OUT) {
        vol : 0.1;
        voh : 1.4;
    }
}

```

```

    vomin : -0.3;
    vomax : VDD + 0.3;
}
/* Template definitions for noise immunity. Variable:
 * input_noise_width */
poly_template ( my_noise_reject ) {
    temperature, total_output_net_capacitance);
variables ( input_noise_width, voltage, voltage1, \
            temperature, total_output_net_capacitance);
mapping(voltage, VDD1);
mapping(voltage1, VDD2);
variable_1_range (0, 2);
variable_2_range (1.4, 1.8);
variable_3_range (1.1, 1.5);
variable_4_range (-40, 125);
variable_5_range (0.0, 1.0);
domain (typ) {
    variables ( input_noise_width, voltage, voltage1,\ 
                temperature, total_output_net_capacitance);
    variable_1_range (0, 2);
    variable_2_range (1.5, 1.7);
    variable_3_range (1.2, 1.4);
    variable_4_range (25, 25);
    variable_5_range (0.0, 1.0);
    mapping(voltage, VDD1);
    mapping(voltage1, VDD2);
}
domain (min) {
    variables ( input_noise_width, voltage, voltage1, \
                temperature );
    variable_1_range (0, 2);
    variable_2_range (1.7, 1.8);
    variable_3_range (1.4, 1.5);
    variable_4_range (-40, -40);
    mapping(voltage, VDD1);
    mapping(voltage1, VDD2);
}
domain (max) {
    variables ( input_noise_width, voltage, voltage1, \
                temperature );
    variable_1_range (0, 2);
    variable_2_range (1.6, 1.7);
    variable_3_range (1.1, 1.2);
    variable_4_range (125, 125);
    mapping(voltage, VDD1);
    mapping(voltage1, VDD2);
}
} /* end poly_template (my_noise_reject) */

```

```

poly_template ( my_noise_reject_outside_rail ) {
    variables ( input_noise_width, voltage, voltage1, \
                temperature );
    mapping(voltage, VDD1);
    mapping(voltage1, VDD2);
    variable_1_range (0, 2);
    variable_2_range (1.4, 1.8);
    variable_3_range (1.1, 1.5);
    variable_4_range (-40, 125);
} /* end poly_template ( my_noise_reject_outside_rail ) */
/* Template definitions for I-V characteristics. Variable:
* iv_output_voltage */
poly_template ( my_current_low ) {
    variables ( iv_output_voltage, voltage, voltage1, \
                temperature );
    mapping(voltage, VDD1);
    mapping(voltage1, VDD2);
    variable_1_range (-1, 2);
    variable_2_range (1.4, 1.8);
    variable_3_range (1.1, 1.5);
    variable_4_range (-40, 125);
} /* end poly_template ( my_current_low ) */
poly_template ( my_current_high ) {
    variables ( iv_output_voltage, voltage, voltage1, \
                temperature );
    mapping(voltage, VDD1);
    mapping(voltage1, VDD2);
    variable_1_range (-1, 2);
    variable_2_range (1.4, 1.8);
    variable_3_range (1.1, 1.5);
    variable_4_range (-40, 125);
} /* end poly_template ( my_current_high ) */
/* Template definitions for propagated noise. Variables:
* input_noise_width
* input_noise_height
* input_peak_time_ratio
* total_output_net_capacitance */
poly_template(my_propagated_noise) {
    variables ( input_noise_width, input_noise_height, \
                input_peak_time_ratio \
                total_output_net_capacitance, voltage, \
                voltage1, temperature );
    mapping(voltage, VDD1);
    mapping(voltage1, VDD2);
    variable_1_range (0.01, 2);
    variable_2_range (0, 0.8);
    variable_3_range (0.0, 1.0);
    variable_4_range (0, 2);
}

```

```

variable_5_range (1.4, 1.8);
variable_6_range (1.1, 1.5);
variable_7_range (-40, 125);
} /* end poly_template (my_propagated_noise) */
/* INVERTER */
cell ( INV ) {
    area : 1 ;
    pin ( A ) {
        direction : input ;
        capacitance : 1 ;
        fanout_load : 1 ;
        /* DC noise margins.
         * These are used for compatibility of level shifters.
         * In noise analysis, they are the least accurate way
         * to define noise margins.
         * Compatible: can coexist in the pin group with any
         * other noise margin definition. */
        input_voltage : MY_CMOS_IN ;
        /* Noise group defines what is acceptable noise on input
         * pins. */
        /* Hyperbolic noise immunity.
         * Another way to specify noise immunity.
         * Mutually exclusive: noise_immunity_low cannot be
         * together with
         *hyperbolic_noise_immunity_low, and so on.
         * Defines pulse_height = height_coefficient +
         * area_coefficient / (width - width_coefficient)
         * Characterization recommendation: Use
         * hyperbolic_noise_immunity_*
         * if it can fit the curve, otherwise use
         * table noise_immunity_* */
        hyperbolic_noise_low() {
            height_coefficient : 0.4;
            area_coefficient : 1.1;
            width_coefficient : 0.1;
        }
        hyperbolic_noise_high() {
            height_coefficient : 0.3;
            area_coefficient : 0.9;
            width_coefficient : 0.1;
        }
        hyperbolic_noise_below_low() {
            height_coefficient : 0.1;
            area_coefficient : 0.3;
            width_coefficient : 0.01;
        }
        hyperbolic_noise_above_high() {
            height_coefficient : 0.1;

```

```

        area_coefficient : 0.3;
        width_coefficient : 0.01;
    }
} /* end pin (A) */
pin ( Y ) {
    direction : output ;
    max_fanout : 10 ;
    function : " !A ";
    output_voltage : MY_CMOS_OUT ;
    timing () {
        related_pin : A ;
        /* Steady state drive resistance */
        steady_state_resistance_high : 1500;
        steady_state_resistance_low : 1100;
        steady_state_resistance_above_high : 200;
        steady_state_resistance_below_low : 100;
        /* I-V curve.
        * Describes how much current the pin can deliver in a given state for
        * a given voltage on the pin.
        * Voltage is measured from the pin to ground, current is measured
        * flowing into the cell (both can be either positive or negative). */
        steady_state_current_low(my_current_low) {
            orders ("3, 3, 0, 0");
            coefs ("8.4165, 0.3198, -0.0004, 0.0000, \
1133.8274, 8.7287, -0.0054, 0.0000, \
139.8645, -60.3898, 0.0589, -0.0000, \
-167.4473, 95.7112, -0.1018, 0.0000");
        }
        steady_state_current_high(my_current_high) {
            orders ("3, 3, 0, 0");
            coefs ("10.9165, 0.2198, -0.0003, 0.0000, \
1433.8274, 8.7287, -0.0054, 0.0000, \
128.8645, -60.3898, 0.0589, -0.0000, \
-167.4473, 95.7112, -0.1018, 0.0000");
        }
        /* Noise immunity.
        * Defines maximum allowed noise height for given pulse width.
        * Pulse height is absolute value from the signal level.
        * Any of the 4 tables below are optional. */
        noise_immunity_low (my_noise_reject) {
            domain (typ) {
                orders ("3, 3, 0, 0, 0");
                coefs ("11.4165, 0.2198, -0.0003, 0.0000, \
1353.8274, 8.7287, -0.0054, 0.0000, \
149.8645, -60.3898, 0.0589, -0.0000, \
-167.4473, 95.7112, -0.1018, 0.0000");
            }
            domain (min) {

```

```

        orders ("3, 3, 0, 0");
        coefs ("6.964065, 0.134078, -0.000183, 0.0000, \
825.834714, 5.324507, -0.003294, 0.0000, \
91.417345, -36.837778, .035929, -0.0000, \
-102.142853, 58.383832, -.062098, 0.0000");
    }
    domain (max) {
        orders ("3, 3, 0, 0");
        coefs ("19.065555, 0.367066, -0.000501, 0.0000, \
2260.891758, 14.576929, -0.009018, 0.0000, \
250.273715, -100.850966, 0.098363, -0.0000, \
-279.636991, 159.837704, -0.170006, 0.0000");
    }
}
noise_immunity_high (my_noise_reject) {
    domain (typ) {
        orders ("3, 3, 0, 0, 0");
        coefs ("12.4165, 0.2198, -0.0003, 0.0000, \
1353.8274, 8.7287, -0.0054, 0.0000, \
129.8645, -60.3898, 0.0589, -0.0000, \
-147.4473, 95.7112, -0.1018, 0.0000");
    }
    domain (min) {
        orders ("3, 3, 0, 0");
        coefs ("6.364065, 0.134078, -0.000183, 0.0000, \
845.834714, 5.324507, -0.003294, 0.0000, \
91.417345, -36.837778, .035929, -0.0000, \
-103.142853, 58.383832, -.062098, 0.0000");
    }
    domain (max) {
        orders ("3, 3, 0, 0");
        coefs ("19.265555, 0.367066, -0.000601, 0.0000, \
2460.891758, 14.576929, -0.009018, 0.0000, \
250.273715, -130.850966, 0.098363, -0.0000, \
-279.636991, 159.837704, -0.170006, 0.0000");
    }
}
noise_immunity_below_low (my_noise_reject_outside_rail) {
    orders ("3, 3, 0, 0");
    coefs ("10.4165, 0.1198, -0.0003, 0.0000, \
1333.8274, 8.7287, -0.0054, 0.0000, \
149.8645, -60.3898, 0.0589, -0.0000, \
-167.4473, 95.7112, -0.1018, 0.0000");
}
noise_immunity_above_high (my_noise_reject_outside_rail) {
    orders ("3, 3, 0, 0");
    coefs ("12.4165, 0.2298, -0.0003, 0.0000, \
1253.8274, 8.7287, -0.0054, 0.0000, \

```

```

    149.8645, -60.3898, 0.0589, -0.0000, \
    -167.4473, 95.7112, -0.1018, 0.0000");
}

/* Propagated noise.
 * It is a function of input noise width and height and output
 * capacitance. Width and height are in separate tables. */
propagated_noise_width_high(my_propagated_noise) {
    orders ("1, 2, 1, 0, 0, 0");
    coefs ("8.4165, 0.3198, -0.0004, 0.2000, \
    1.8645, -6.3898, 0.0589, -0.03000, \
    -1.4473, 9.7112, -0.1018, 0.3500, ");
}

propagated_noise_height_high(my_propagated_noise) {
    orders ("1, 2, 1, 0, 0, 0");
    coefs ("0.4165, 0.3198, -0.0014, 0.2000, \
    0.8645, -6.3898, 0.0589, -0.03000, \
    -0.4473, 0.7112, -0.1018, 0.3500, ");
}

propagated_noise_peak_time_ratio_high(my_propagated_noise) {
    orders ("1, 2, 1, 0, 0, 0");
    coefs ("0.4165, 0.3198, 0.0014, 0.2000, \
    0.8645, 0.3898, 0.0589, 0.03000, \
    0.4473, 0.7112, 0.1018, 0.3500, ");
}

propagated_noise_width_low(my_propagated_noise) {
    orders ("1, 2, 1, 0, 0, 0");
    coefs ("8.4165, 0.3198, -0.0004, 0.2000, \
    1.8645, -6.3898, 0.0589, -0.03000, \
    -1.4473, 9.7112, -0.1018, 0.3500, ");
}

propagated_noise_height_low(my_propagated_noise) {
    orders ("1, 2, 1, 0, 0, 0");
    coefs ("0.4165, 0.3198, -0.0014, 0.2000, \
    0.8645, -6.3898, 0.0589, -0.03000, \
    -0.4473, 0.7112, -0.1018, 0.3500, ");
}

propagated_noise_peak_time_ratio_low(my_propagated_noise) {
    orders ("1, 2, 1, 0, 0, 0");
    coefs ("0.4165, 0.3198, 0.0014, 0.2000, \
    0.8645, 0.3898, 0.0589, 0.03000, \
    0.4473, 0.7112, 0.1018, 0.3500, ");
}

propagated_noise_width_above_high(my_propagated_noise) {
    orders ("1, 2, 1, 0, 0, 0");
    coefs ("8.4165, 0.3198, -0.0004, 0.2000, \
    1.8645, -6.3898, 0.0589, -0.03000, \
    -1.4473, 9.7112, -0.1018, 0.3500, ");
}

```

```

propagated_noise_height_above_high(my_propagated_noise) {
    orders ("1, 2, 1, 0, 0, 0, 0");
    coefs ("0.4165, 0.3198, -0.0014, 0.2000, \
0.8645, -6.3898, 0.0589, -0.03000, \
-0.4473, 0.7112, -0.1018, 0.3500, ");
}
propagated_noise_peak_time_ratio_above_high(my_propagated_noise) {
    orders ("1, 2, 1, 0, 0, 0, 0");
    coefs ("0.4165, 0.3198, 0.0014, 0.2000, \
0.8645, 0.3898, 0.0589, 0.03000, \
0.4473, 0.7112, 0.1018, 0.3500, ");
}
propagated_noise_width_below_low(my_propagated_noise) {
    orders ("1, 2, 1, 0, 0, 0, 0");
    coefs ("8.4165, 0.3198, -0.0004, 0.2000, \
1.8645, -6.3898, 0.0589, -0.03000, \
-1.4473, 9.7112, -0.1018, 0.3500, ");
}
propagated_noise_height_below_low(my_propagated_noise) {
    orders ("1, 2, 1, 0, 0, 0, 0");
    coefs ("0.4165, 0.3198, -0.0014, 0.2000, \
0.8645, -6.3898, 0.0589, -0.03000, \
-0.4473, 0.7112, -0.1018, 0.3500, ");
}
propagated_noise_peak_time_ratio_below_low(my_propagated_noise) {
    orders ("1, 2, 1, 0, 0, 0, 0");
    coefs ("0.4165, 0.3198, 0.0014, 0.2000, \
0.8645, 0.3898, 0.0589, 0.03000, \
0.4473, 0.7112, 0.1018, 0.3500, ");
}
cell_rise(scalar) { values("0");}
rise_transition(scalar) { values("0");}
cell_fall(scalar) { values("0");}
fall_transition(scalar) { values("0");}
} /* end of timing group */
} /* end of pin (Y) */
} /* end of cell (INV) */
} /* end of library (my_noise_lib)

```

---

## Nonlinear Delay Model Library With Noise Information

A nonlinear delay model noise library is limited to a fixed voltage.

```
library (my_noise_lib) {
    delay_model : "table_lookup";
    time_unit : "1ns";
    voltage_unit : "1V";
    current_unit : "1mA";
    capacitive_load_unit (1,PF);
    pulling_resistance_unit : 1kohm;
    nom_voltage : 1.6;
    nom_temperature : 40.0;
    nom_process : 1.0;
    /* Templates of input and output levels (used for DC noise margin) */
    input_voltage(MY_CMOS_IN) {
        vil : 0.3;
        vih : 1.1;
        vimin : -0.3;
        vimax : VDD + 0.3;
    }
    output_voltage(MY_CMOS_OUT) {
        vol : 0.1;
        voh : 1.4;
        vomin : -0.3;
        vomax : VDD + 0.3;
    }
    /* Template definitions for noise immunity. Variable:
     * input_noise_width */
    noise_lut_template(my_noise_reject) {
        variable_1 : input_noise_width;
        variable_2 : total_output_net_capacitance;
        index_1("0, 0.1, 0.3, 1, 2");
        index_2("0, 0.1, 0.3, 1, 2");
    }
    noise_lut_template(my_noise_reject_outside_rail) {
        variable_1 : input_noise_width;
        variable_2 : total_output_net_capacitance;
        index_1("0, 0.1, 2");
        index_2("0, 0.1, 2");
    }
    /* Template definitions for I-V characteristics. Variable:
     * iv_output_voltage */
    iv_lut_template(my_current_low) {
        variable_1 : iv_output_voltage
        index_1("-1, -0.1, 0, 0.1 0.8, 1.6, 2");
    }
}
```

```

iv_lut_template(my_current_high) {
    variable_1 : iv_output_voltage
    index_1("-1, 0, 0.3, 0.5, 0.8, 1.5, 1.6, 1.7, 2");
}
/* Template definitions for propagated noise. Variables:
 * input_noise_width
 * input_noise_height
 * total_output_net_capacitance */
propagation_lut_template(my_propagated_noise) {
    variable_1 : input_noise_width;
    variable_2 : input_noise_height;
    variable_3 : total_output_net_capacitance;
    index_1("0.01, 0.2, 2");
    index_2("0.2, 0.8");
    index_3("0, 2");
}
cell (tieoff_30_esd) {
    pin (high) {
        direction : output;
        capacitance : 0;
        function : "1";
        /* noise information */
        timing() {
            tied_off : true;
            steady_state_resistance_high : 1.22;
            steady_state_resistance_above_high : 1.00;
            steady_state_current_high(iv1x5){
                index_1("0.3,0.75,1.0,1.2,2");
                values("-513.2,-447.9,-359.3,-245.7,497.3");
            }
        }
    }
    pin (low) {
        direction : output;
        capacitance : 0;
        function : "0";
        /* noise information */
        timing() {
            tied_off : true;
            steady_state_resistance_low : 0.1;
            steady_state_resistance_below_low : 0.4;
            steady_state_current_low(iv1x5){
                index_1("-0.25,0.3,0.5,1.0,1.8");
                values("-595.4,555.4,690.5,774.75,822.5");
            }
        }
    }
}

```

```

}

/* INVERTER */
cell ( INV ) {
    area : 1 ;
    pin ( A ) {
        direction : input ;
        capacitance : 1 ;
        fanout_load : 1 ;
        /* DC noise margins.
         * These are used for compatibility of level shifters. In noise
         * analysis they are the least accurate way to define noise margins.
         * Compatible: can coexist in the pin group with any other noise margin
         * definition. */
        input_voltage : MY_CMOS_IN ;
        /* Timing group defines what is acceptable noise on input pins. */
        /* Hyperbolic noise immunity.
         * Another way to specify noise immunity.
         * Mutually exclusive: noise_immunity_low cannot be together with
         * hyperbolic_noise_immunity_low, etc.
         * Defines pulse_height = height_coefficient +
         * area_coefficient / (width - width_coefficient)
         * Characterization recommendation: use hyperbolic_noise_immunity_*
         * if can fit the curve, otherwise table noise_immunity_* */
        hyperbolic_noise_low() {
            height_coefficient : 0.4;
            area_coefficient : 1.1;
            width_coefficient : 0.1;
        }
        hyperbolic_noise_high() {
            height_coefficient : 0.3;
            area_coefficient : 0.9;
            width_coefficient : 0.1;
        }
        hyperbolic_noise_below_low() {
            height_coefficient : 0.1;
            area_coefficient : 0.3;
            width_coefficient : 0.01;
        }
        hyperbolic_noise_above_high() {
            height_coefficient : 0.1;
            area_coefficient : 0.3;
            width_coefficient : 0.01;
        }
    } /* end of pin A */
    pin ( Y ) {
        direction : output ;
        max_fanout : 10 ;
        function : " !A ";
    }
}

```

```

output_voltage : MY_CMOS_OUT
min_input_noise_width : 0.0;
max_input_noise_width : 2.0;
timing () {
    related_pin : A ;
    /* Steady-state drive resistance */
    steady_state_resistance_high : 1500;
    steady_state_resistance_low : 1100;
    steady_state_resistance_above_high : 200;
    steady_state_resistance_below_low : 100;
    /* I-V curve.
     * Describes how much current the pin can deliver in a given state for
     * a given voltage on the pin. The steady_state_resistance*_max is the
     * highest resistance in the I-V curve, the
     * steady_state_resistance*_min is the lowest.
     * Mutually exclusive: If steady_state_resistance_low* or
     * steady_state_resistance_max or steady_state_resistance_min is
     * specified, the I-V curve cannot be specified.
     * Characterization recommendation: Use steady_state_resistance* if
     * an I-V curve cannot be generated.
     * Voltage is measured from the pin to ground, current measured
     * flowing into the cell (both can be either positive or negative). */
    steady_state_current_low(my_current_low) {
        values("0.1, 0.05, 0, -0.1, -0.25, -1, -1.8");
    }
    steady_state_current_high(my_current_high) {
        values("2, 1.8, 1.7, 1.4, 1, 0.5, 0, -0.1, -0.8");
    }
    cell_rise(scalar) { values("0"); }
    rise_transition(scalar) { values("0"); }
    cell_fall(scalar) { values("0"); }
    fall_transition(scalar) { values("0"); }
    /* Noise immunity.
     * Defines the maximum allowed noise height for given pulse width.
     * Pulse height is the absolute value from the signal level.
     * Any of the following four tables are optional. */
    noise_immunity_low (my_noise_reject) {
        values ("1.5, 0.9, 0.8, 0.65, 0.6", \
                "1.5, 0.9, 0.8, 0.65, 0.6", \
                "1.5, 0.9, 0.8, 0.65, 0.6", \
                "1.5, 0.9, 0.8, 0.65, 0.6");
    }
    noise_immunity_high (my_noise_reject) {
        values ("1.3, 0.8, 0.7, 0.6, 0.55", \
                "1.5, 0.9, 0.8, 0.65, 0.6", \
                "1.5, 0.9, 0.8, 0.65, 0.6", \
                "1.5, 0.9, 0.8, 0.65, 0.6");
    }
}

```

```

    "1.5, 0.9, 0.8, 0.65, 0.6");
}
noise_immunity_below_low (my_noise_reject_outside_rail) {
    values ("1, 0.8, 0.5", \
"1, 0.8, 0.5", \
"1, 0.8, 0.5");
}
noise_immunity_above_high (my_noise_reject_outside_rail) {
    values ("1, 0.8, 0.5", \
"1, 0.8, 0.5", \
"1, 0.8, 0.5");
}
/* Propagated noise.
 * A function of input noise width, height, and output
 * capacitance. Width and height are in separate tables. */
propagated_noise_width_high(my_propagated_noise) {
    values ("0.01, 0.10", "0.15, 0.04", "0.14, 0.18", \
"0.05, 0.15", "0.24, 0.07", "0.17, 0.32");
}
propagated_noise_height_high(my_propagated_noise) {
    values ("0.01, 0.10", "0.15, 0.04", "0.14, 0.18", \
"0.05, 0.15", "0.24, 0.07", "0.17, 0.32");
}
propagated_noise_width_low(my_propagated_noise) {
    values ("0.01, 0.10", "0.15, 0.04", "0.14, 0.18", \
"0.05, 0.15", "0.24, 0.07", "0.17, 0.32");
}
propagated_noise_height_low(my_propagated_noise) {
    values ("0.01, 0.10", "0.15, 0.04", "0.14, 0.18", \
"0.05, 0.15", "0.24, 0.07", "0.17, 0.32");
}
propagated_noise_width_above_high(my_propagated_noise) {
    values ("0.01, 0.10", "0.15, 0.04", "0.14, 0.18", \
"0.05, 0.15", "0.24, 0.07", "0.17, 0.32");
}
propagated_noise_height_above_high(my_propagated_noise) {
    values ("0.01, 0.10", "0.15, 0.04", "0.14, 0.18", \
"0.05, 0.15", "0.24, 0.07", "0.17, 0.32");
}
propagated_noise_width_below_low(my_propagated_noise) {
    values ("0.01, 0.10", "0.15, 0.04", "0.14, 0.18", \
"0.05, 0.15", "0.24, 0.07", "0.17, 0.32");
}
propagated_noise_height_below_low(my_propagated_noise) {
    values ("0.01, 0.10", "0.15, 0.04", "0.14, 0.18", \
"0.05, 0.15", "0.24, 0.07", "0.17, 0.32");
} /*end propagated noise groups */
} /* end of timing group */

```

```
    } /* end of pin (Y)( */
} /* end of cell (INV) */
} /* end of library (my_noise_lib)
```

---

## Checking Noise Library Data

Use the noise data checker to screen for errors in your library noise data. To invoke the screener:

```
lc_shell> enable_timing_noise_signoff = 1
lc_shell> read_lib <library_name> -signoff_screening
```

The screener checks noise data to ensure it meets the requirements for static noise analysis. There are four types of noise data in a library: DC Noise Margin, IV Curves, Noise Immunity Curve and Noise Propagation Data. All the checks work on the above four data and you have the option of verifying all or some of them. By default, noise propagation data is not checked, because it is optional. The screener provides the following checks for noise.

### Basic Checks

This check scans the library for essential data that is required to do static noise analysis and reports any missing information. DC Noise Margin and IV Curves must be in any noise library. The screener reports an error if they are missing. The screener warns for missing data

## Recommended Checks

These following checks are recommended, but are not essential.

- Range Checks

Verify whether the IV Curve output voltage range index specified is large enough for wide coverage. This index must cover Vdd to 2\*Vdd.

- Verify whether the noise height for the minimum characterized noise width (for the noise immunity curve) is at least VDD.

There is a default tolerance of 5 percent VDD for this check.

- Verify whether the maximum characterized noise width for the Noise Immunity Curve is at least the value calculated using the following formula

$$\text{WidthMAX} = \text{SlewMAX} * \text{factor} / (\text{hstp} - \text{lstp})$$

where the factor is derived from SPICE simulations (default factor = 3). SlewMAX is the maximum transition design rule, hstp is high slew trip point, and lstp is the low slew trip point.

If max\_transition rule is absent in the library, this check is not performed.

- Verify whether the maximum characterized input noise width for Noise Propagation Data table is at least the value calculated using the following formula

$$\text{WidthMAX} = \text{SlewMAX} * \text{factor} / (\text{hstp} - \text{lstp})$$

where the factor is derived from SPICE simulations (the default factor = 3). SlewMAX is the max transition design rule, hstp is high slew trip point, and lstp is low slew trip point.

If the max\_transition rule is absent in the library then this check is not performed

- Verify whether the maximum characterized input noise height for noise propagation tables is at least VDD

## **Extrapolation Checks**

All extrapolation checks contain an option for you to specify for tolerance. Only data that exceeds the specified tolerance value is reported. The extrapolation checks are as follows:

- Check if noise immunity curves do not curl upward or level off at the end. There is a tolerance of 3 percent VDD for the curling-up check and 5 percent VDD for the leveling-off check.
- Noise Propagation data table should be such that an input noise of zero height or zero width should not propagate any noise at the output. The screener extrapolates Noise Propagation Data at zero input noise height/width and reports if the extrapolated value is greater than 0. The following tolerances are used:

```
Output noise height @ zero input noise height <= 3%Vdd  
Output noise width @ zero input noise height <= 0.1 ns  
Output noise height @ zero input noise width <= 5%Vdd  
Output noise width @ zero input noise width <= 0.1 ns
```

## **Current Polarity**

The current polarity check is as follows:

- Verify if all the characterized steady state current (IV Curves) have the correct polarity. Current flow into the cell is defined as positive.

## **Negative Table Values**

- Report any noise immunity or propagation table that has negative values in it.
- Report any negative height or area coefficients in the hyperbolic noise immunity function

## **Advanced Checks**

Advanced Checks reports any non-monotonous noise propagation table. For any given input noise width and height index, if the output load is increased the output noise (width and height) must decrease. There is a tolerance of 1 percent VDD.

You can also view a plot of noise height versus. load for any reported noise table.

# 11

## Generating Library Reports

---

You use the `report_lib` command to generate reports containing library information. When no command options are specified, the `report_lib` command displays the library-level data and information about the available cells, including

- Header information
- Operating conditions
- Wire load models
- Design rules
- A list of cells and their types

The command options let you control the reporting of

- Timing data
- Power data

- Electromigration data
- Noise data
- Functionality data
- FPGA data
- User-defined data
- Comprehensive data

Note:

For information about physical library reports, see the *Library Compiler User Guide: Physical Libraries*.

The following sections describe how to use the `report_lib` command options to generate the default report and the various specific reports.

- Generating Default Reports
- Generating Specific Reports
- Timing Reports
- Noise Reports
- Power Reports
- Electromigration Reports
- Functionality Reports
- FPGA Reports
- User-Defined Data Reports
- Comprehensive Reports

---

## Generating Default Reports

If you do not specify any command options, the `report_lib` command displays a default report containing the library-level data and information about the available cells such as the operating conditions, wire load models, design rules, and the types of cells.

Example 11-1 shows the header section of a default report.

### *Example 11-1 Timing Report Header*

```
*****
Report : library
Library: my_lib
Version: 2000.01
Date   : Tues Jan 14 14:05:20 2000
*****  
  
Library Type          : Technology
Tool Created          : 2000.01
Date Created          : Not Specified
Library Version       : Not Specified
Time Unit             : Not specified
Capacitive Load Unit : 1.000000pf
Pulling Resistance Unit : 1kilo-ohm
Voltage Unit          : 1V
Current Unit          : 1mA
Leakage Power Unit    : Not specified.
Bus Naming Style      : %s[%d] (default)  
  
Operating Conditions:  
  
Name Library Calc_mode Process Temp Volt Interconnect
Model  
-----  
NOM           my_lib 1.00  25.00  3.30 balanced_tree
WCCOM         my_lib 1.31  70.00  3.13 balanced_tree
WCIND         my_lib 1.31  85.00  3.13 balanced_tree  
  
Input Voltages:
No input_voltage groups specified.
```

Output Voltages:  
No output\_voltage groups specified.

Wire Loading Model:

Name	:	B0X0
Location	:	my_lib
Resistance	:	0
Capacitance	:	0
Area	:	0
Slope	:	0
Fanout	Length	Points Average Cap Std Deviation
-----		
1	0.00	
2	0.00	
3	0.00	
4	0.00	

Wire Loading Model Mode: enclosed.

Porosity information:  
No porosity information specified.

In\_place optimization mode: match\_footprint

Timing Ranges:  
No timing ranges specified.

Delay Threshold Trip-Points:  
input\_threshold\_pct\_rise: 50  
output\_threshold\_pct\_rise: 50  
input\_threshold\_pct\_fall: 50  
output\_threshold\_pct\_fall: 50

Slew Threshold Trip-Points:  
slew\_lower\_threshold\_pct\_rise: 20  
slew\_upper\_threshold\_pct\_rise: 80  
slew\_lower\_threshold\_pct\_fall: 20  
slew\_upper\_threshold\_pct\_fall: 80

slew\_derate\_from\_library: 1

Components:

```

Attributes:
  af - active falling
  ah - active high
  al - active low
  ar - active rising
  b - black box (function unknown)
  ce - clock enable
  d - dont_touch
  mo - map_only
  p - preferred
  r - removable
  s - statetable
  sa0 - dont_fault stuck-at-0
  sa1 - dont_fault stuck-at-1
    sa01 - dont_fault both stuck-at-0 and stuck-at-1
  sz - use_for_size_only
  t - test cell
  u - dont_use
  udp - usable for datapath generator

```

Cell	Footprint	Attributes
Cell1	"afhcin"	d, r, udp

## Generating Specific Reports

This section describes how to use the `report_lib` command options to generate specific reports.

---

## Timing Reports

You can use the following options to generate timing reports for the generic CMOS delay model, the CMOS nonlinear delay model, or the scalable polynomial delay model:

`-timing`

Displays pin-level timing information in terms of constraints or delays.

Note:

The `-timing` option becomes available only after you have read the library source file (.lib) into the tool's memory.

`-timing_arcs`

Displays all the timing arcs information in the library. When you use this option, the report lists the arc sense, type, from and to pins, and the `when` statement of all cells.

`-timing_label`

Displays all the timing arc label information.

## Library Defaults

The Synopsys design tool applies default values to timing-related attributes that are not defined in the technology library. The library defaults section of a default report lists timing-related attributes and the design rule constraint default values.

For timing-related attributes defined in the library, the default report shows the corresponding default values. For timing-related attributes that are not defined in the library, the default report shows a blank value field.

[Example 11-2](#) shows the Timing Attribute Defaults section of a default report.

### *Example 11-2 Timing Attribute Defaults*

Timing Attribute Defaults:

Attribute	Default
max_transition	
max_fanout	
fanout_load	1
max_capacitance	3.5
rise_pin_resistance	0
fall_pin_resistance	0
rise_delay_intercept	0
fall_delay_intercept	0
wire_capacitance	0
rise_wire_resistance	0
fall_wire_resistance	0
rise_nonpaired_twin	0
fall_nonpaired_twin	0

## Library-Level Templates

The nonlinear delay, load-dependent, and scalable polynomial delay models have library-level templates, which are written out in the timing report.

### Lookup Table Template

This is the format of the lookup table template information:

```
Template_name
-----
name of the template
  VARIABLE_1: variable_1
  VARIABLE_2: variable_2
  VARIABLE_3: variable_3
  INDEX_1: float ... float
  INDEX_2: float ... float
```

```
INDEX_3: float ... float
```

*variable\_2* and *variable\_3* are optional for all tables except the three-dimensional tables used to model output load. The lines containing INDEX\_2 and INDEX\_3 data are optional.

## **Polynomial Template**

This is the format of the scalable polynomial template information:

Poly Template:

```
Template_name
-----
name of the polynomial template
variables( variable_1, variable2, ...)
variable_1_range(min1_value, max1_value)
variable_2_range(min2_value, max2_value)
...
DOMAIN(domain1_name)
variables(variable_1, variable_2, ...)
variable_1_range(min1_value, max1_value)
variable_2_range(min2_value, max2_value)
DOMAIN(domain2_name)
CALC_MODE: name
```

The number of variable\_n\_ranges shown in the report depends on the number of variables in the template.

## **Template Examples**

[Example 11-3](#), [Example 11-4](#), and [Example 11-5](#) show the templates for the nonlinear delay, load-dependent, and scalable polynomial delay models, respectively.

### *Example 11-3 Nonlinear Delay Model Template*

```
Template_name
-----
prop
    VARIABLE_1: total_output_net_capacitance
```

```

VARIABLE_2: input_net_transition
INDEX_1: 0.0000    1.0000    2.0000    3.0000    4.0000
INDEX_2: 0.0000    1.0000    2.0000    3.0000    4.0000
tran
  VARIABLE_1: total_output_net_capacitance
  VARIABLE_2: input_net_transition
  INDEX_1: 0.0000    1.0000    2.0000    3.0000    4.0000

```

*Example 11-4 Load-Dependent Template*

```

Template_name
-----
load
  VARIABLE_1: constrained_pin_transition
  VARIABLE_2: related_pin_transition
  VARIABLE_3: related_out_total_output_net_capacitance
  INDEX_1: 0.1000    1.2000    2.3000    3.4000
  INDEX_2: 0.1000    1.2000    2.3000    3.4000
  INDEX_3: 0.1000    1.2000    2.3000    3.4000

```

*Example 11-5 Scalable Polynomial Delay Model Template*

```

Template_name
-----
scalar1
  variables(total_output_net_capacitance)
  variable_1_range( 1.00,    2.00)
ptable3x1
  variables(total_output_net_capacitance)
  variable_1_range( 1.00,   10.00)
  DOMAIN(D1)
    variables(total_output_net_capacitance)
    variable_1_range( 1.00,    5.00)
  DOMAIN(D2)
    variables(total_output_net_capacitance)
    variable_1_range( 5.00,   10.00)
constraint
  variable(constrained_pin_transition,
            related_pin_transition,
            related_out_total_output_net_capacitance)
  variable_1_range( 1.00,    2.00)
  variable_2_range( 2.00,    3.00)
  variable_3_range( 3.00,    4.00)

```

## Cell-Level Data

The timing report contains the following area and timing information for each cell.

The timing report includes a PIN section for every pin in the cell.

### Syntax

```
CELL(cell_name) : area, ...;  
  
PIN(pin_name) : attributes;  
timing_information  
END_PIN pin_name;  
  
PIN(pin_name) : attributes;  
timing_information  
END_PIN pin_name;  
  
END_CELL cell_name;
```

### Pin Attributes

Input, output, and bidirectional pins have different pin attributes.

These are the attributes for an input pin:

```
PIN(pin_name) : in, capacitance, fanout_load,  
max_transition, rise_capacitance, fall_capacitance ;
```

These are the attributes for an output pin:

```
PIN(pin_name) : out, capacitance, max_fanout,  
max_transition, max_capacitance, min_fanout,  
min_capacitance ;
```

These are the attributes for a bidirectional pin:

```
PIN(pin_name) : inout, capacitance, fanout_load,  
max_fanout, max_transition, max_capacitance,  
min_fanout, min_transition, min_capacitance,  
rise_capacitance, fall_capacitance ;
```

## **Operating Conditions**

Reports for scaled timing cells also include information about the `operating_conditions` group.

```
SCALED_CELL(cell_name, op_conditions) : area,  
...  
END_SCALED_CELL cell_name ;
```

## **Pin-Level Information**

Library Compiler lists timing information in terms of constraints or delays and pin capacitance ranges. Each category uses a different format. You generate such a report by using the `-timing` option of the `report_lib` command.

Library Compiler also lists cell degradation information under timing, because cell degradation is modeled like a `timing` group.

## **Constraints**

The constraints include setup and hold on sequential cells, nonsequential setup and hold, recovery time, removal, skew, and no-change.

### **Sequential Setup and Hold Constraint**

In sequential cells, the constrained pin is normally the data input pin of a latch or flip-flop. When the clock pin event defined by the transition type occurs, the pin must maintain a stable logic 1 or 0 for the rise or fall time (`intrinsic_rise` or `intrinsic_fall` in CMOS linear delay models; `rise_constraint` or `fall_constraint` in CMOS nonlinear and scalable polynomial delay models).

## Nonsequential Setup and Hold Constraint

Nonsequential setup and hold constraints differ from setup and hold constraints on sequential cells, in that the related pin is not a clock. These constraints define how long the data signal on an input pin remains stable before or after the related pin changes state.

## Recovery Time Constraint

The constrained pin is usually an asynchronous control pin, such as clear or preset. When the clock pin event defined by the transition type occurs, the pin must remain inactive for the `intrinsic_rise` or `intrinsic_fall` time.

## Removal Timing Constraint

This constraint defines the minimum separation between a clock active edge and an asynchronous control signal inactive edge.

## Skew Constraint

This constraint defines the maximum interval allowed between two clock-edge events.

## No-Change Constraint

This constraint defines the minimum period a signal must remain stable in relation to a related signal.

## Clock Insertion Delay Constraint

The arrival timing path defines the minimum and maximum timing constraint for a pin that is driving an internal clock tree for each input transition.

In conditional timing checks,

- The enable condition is the `when` attribute.
- The start enable condition is the `when_start` attribute.

- The end enable condition is the `when_end` attribute.

## Constraint Syntax

This is the syntax for each constraint in the timing report:

```

MPW : pin_name, polarity, 'enable_condition', value;
PERIOD : pin_name, 'enable_condition', value;
SETUP : constrained_pin, clock_pin, transition_type,
        related_output_pin, 'start_enable_condition',
        'end_enable_condition', (intrinsic_rise, intrinsic_fall);

MPW : pin_name, polarity, 'enable_condition', value;
PERIOD : pin_name, 'enable_condition', value;
SETUP : constrained_pin, clock_pin, transition_type,
        related_output_pin, enable_condition, (intrinsic_rise,
intrinsic_fall);

HOLD : clock_pin, transition_type, constrained_pin,
       related_output_pin, 'start_enable_condition',
       'end_enable_condition', (intrinsic_rise, intrinsic_fall);
NON_SEQ_SETUP : constrained_pin, clock_pin,
                 transition_type,
                 related_output_pin, 'start_enable_condition',
                 'end_enable_condition', (intrinsic_rise, intrinsic_fall);

HOLD : clock_pin, transition_type, constrained_pin,
       related_output_pin, 'start_enable_condition',
       'end_enable_condition', (intrinsic_rise, intrinsic_fall);
NON_SEQ_SETUP : constrained_pin, clock_pin,
                 transition_type, related_output_pin, enable_condition,
                 (intrinsic_rise, intrinsic_fall);

NON_SEQ_HOLD : clock_pin, transition_type, constrained_pin,
                 related_output_pin, 'start_enable_condition',
                 'end_enable_condition', (intrinsic_rise, intrinsic_fall);

NON_SEQ_HOLD : clock_pin, transition_type, constrained_pin,
                 related_output_pin, enable_condition, (intrinsic_rise,
intrinsic_fall);

RECOV : constrained_pin, pin_transition, clock_pin,
        clock_transition, related_output_pin,
        'start_enable_condition', 'end_enable_condition',

```

```

(intrinsic_rise, intrinsic_fall);

RECOV : constrained_pin, pin_transition, clock_pin,
         clock_transition, related_output_pin, enable_condition,
(intrinsic_rise, intrinsic_fall);

REMOVAL : clock_pin, clock_transition, constrained_pin,
           pin_transition, related_output_pin,
           'start_enable_condition', 'end_enable_condition',
(intrinsic_rise, intrinsic_fall);

REMOVAL : clock_pin, clock_transition, constrained_pin,
           pin_transition, related_output_pin, enable_condition,
(intrinsic_rise, intrinsic_fall);

SKEW : triggering_clock_pin, triggering_clock_transition,
       constrained_clock_pin, constrained_clock_transition,
       related_output_pin, 'start_enable_condition',
       'end_enable_condition', (intrinsic_rise, intrinsic_fall);

SKEW : triggering_clock_pin, triggering_clock_transition,
       constrained_clock_pin, constrained_clock_transition,
       related_output_pin, enable_condition, (intrinsic_rise,
intrinsic_fall);

NOCHANGE : constrained_pin, constrained_pin_level,
           related_pin, related_pin_level, related_output_pin,
           'start_enable_condition', 'end_enable_condition',
           (intrinsic_rise, intrinsic_fall);

NOCHANGE : constrained_pin, constrained_pin_level,
           related_pin, related_pin_level, related_output_pin,
           enable_condition, (intrinsic_rise, intrinsic_fall);

CLOCK_TREE : clock_pin, constraint_type, timing_sense,
             (intrinsic_rise,intrinsic_fall), (slope_rise, slope_fall),
             (rise_resistance, fall_resistance) ;

```

## Constraint Table Syntax

In the constraint table syntax, the template name can be either `rise_constraint` or `fall_constraint`. `INDEX_1`, `INDEX_2`, and `INDEX_3` appear only when the local table is defined to overwrite the value in the original template.

```
table (template_name) :  
    INDEX_1 : float ... float  
    INDEX_2 : float ... float  
    INDEX_3 : float ... float  
    VALUES : float ... float
```

[Example 11-6](#) and [Example 11-7](#) show pin constraints in different models. [Example 11-8](#) shows a report for a cell with nonsequential setup and hold constraints.

### *Example 11-6 Pin Constraints in a CMOS Linear Delay Model*

```
PIN(D) : in, 1, , , ;  
SETUP : D, CP, rise, '', '', (1.3, 1.3);  
HOLD : CP, rise, D, '', '', (0.3, 0.3);  
END_PIN D;  
  
PIN(CD) : in, 2, , , ;  
RECOV : CD, rise, CP, rise, '', '', (0.5, 0);  
END_PIN CD;  
  
PIN(EN) : in, 3, , , ;  
NOCHANGE : EN, low, CLK, high,, '', '', (1.0,1.0)  
END_PIN EN;  
  
PIN(CLK) : in, 0, , , ;  
MPW : CLK, l, '', 3;  
MPW : CLK, h, '', 3;  
END_PIN CLK;
```

### *Example 11-7 Pin Constraints in a CMOS Nonlinear Delay Model*

```
PIN(D) : in, 1, , , ;  
SETUP : D, CP, rise, '', '', ( , );  
    rise_constraint ( scalar ) :
```

```

        VALUES : 1.3
fall_constraint (scalar) :
        VALUES : 1.3
HOLD : CP, rise, D, '', '', ( , );
        rise_constraint (scalar)
        VALUES : 0.3
fall_constraint (scalar) :
        VALUES : 0.3
END_PIN D;

PIN(CD) : in, 2, , , ;
RECOV : CD, rise, CP, rise, '', '', ( , );
        rise_constraint (scalar) :
        VALUES : 0.5
END_PIN CD;

PIN(EN) : in, 3, , , ;
NOCHANGE : EN, low, CLK, high, , '', '', (,,)
        rise_constraint (scalar) :
        VALUES : 2.98000
fall_constraint (scalar) :
        VALUES : 0.9800
END_PIN EN;

PIN(CLK) : in, 0, , , ;
MPW : CLK, 1, '', 3;
        MPW : CLK, h, '', 3;
END_PIN CLK;

```

*Example 11-8 Nonsequential Setup and Hold Constraints*

```

PIN(d): in, 4, , , ;
NON_SEQ_HOLD: sel, rise, d, so, '', '', ( , );
        rise_constraint ( constraint ) :
        VALUES : 1.5000

        fall_constraint ( constraint ) :
        VALUES : 1.5000

NON_SEQ_SETUP: d, sel, rise, so, '', '', ( , );
        rise_constraint ( constraint ) :
        VALUES : 1.5000

        fall_constraint ( constraint ) :
        VALUES : 1.5000

```

```
END_PIN d;
```

## Delay Reports

Library Compiler reports delays for combinational timing arcs and for sequential timing arcs.

### Linear Combinational Delay

This is the format for the delay report for standard combinational timing:

```
DELAY: from_pin, to_pin, timing_type, timing_sense,
'enable_condition', (intrinsic_rise, intrinsic_fall),
(slope_rise, slope_fall), (rise_resistance,
fall_resistance);
```

The `intrinsic_rise` and `intrinsic_fall` attributes have values if specified, but the `slope_rise`, `slope_fall`, `rise_resistance`, and `fall_resistance` attributes should be empty.

### Piecewise Linear Combinational Delay

The delay report for piecewise linear combinational timing has the following format:

```
DELAY: from_pin, to_pin, timing_type,
timing_sense, 'enable_condition', (intrinsic_rise,
intrinsic_fall), (slope_rise, slope_fall),
(rise_delay_intercept, fall_delay_intercept),
(rise_pin_resistance, fall_pin_resistance),
(rise_wire_resistance, fall_wire_resistance),
(rise_nonpaired_twin, fall_nonpaired_twin);
```

If attributes are composed of pieces, each attribute type is enclosed in angle brackets and, within each attribute type, each piece is separated by a colon, as shown in [Example 11-9](#).

### *Example 11-9 Piecewise Linear Combinational Timing Pin Delays Report*

```
PIN(X) : out, 0, 20, , , 0.83, (0.02, 0.01), (0.045, -0.015);
  DELAY : A1, X, prop, neg_unate, '', (0.12, 0.1), ( , ),
    (<0 : 0.003 : -0.063>, <0 : -0.055 : -0.088>),
    (<0.006 : 0.006 : 0.006>, <0.011 : 0.011 : 0.011>),
    (<0.34 : 0.325 : 0.435>, <0.33 : 1.105 : 1.16>),
    (< : : >, < : : >);
END_PIN X;
```

### **Nonlinear Combinational Delay**

For the nonlinear delay model, Library Compiler reports the table information for combinational timing attributes.

The `intrinsic_rise` and `intrinsic_fall` attributes have values if specified, but the `slope_rise`, `slope_fall`, `rise_resistance`, and `fall_resistance` attributes should be empty.

In the delay table syntax, the template name can be either `rise_propagation`, `fall_propagation`, `cell_rise`, `cell_fall`, `rise_transition`, or `fall_transition`. You use `index_1` or `index_2` only when you define the local table to overwrite the index value in the original template.

### **Scalable Polynomial Combinational Delay**

For the scalable polynomial delay model, Library Compiler reports the coefficients and order information. The scalable polynomial delay model template uses the `rise_propagation`, `fall_propagation`, `cell_rise`, `cell_fall`, `rise_transition`, or `fall_transition` attributes. You use `variable_n_range` only when you define the local polynomial to overwrite the value in the original polynomial template.

## Example

```
cell_rise ( poly1 ) :  
    domain(best)  
    orders("1, 1, 0, 0")  
        "1.0000000+1.0000000a+1.0000000b+1.0000000ab"  
    domain(typ)  
    orders("1, 1, 0, 0")  
        "3.0000000+3.0000000a+3.0000000b+3.0000000ab"
```

## Rise and Fall Values

Rise and fall values have different definitions that depend on the timing type:

- For a combinational prop delay, a rise is the 0-to-1 transition and a fall is the 1-to-0 transition.
- If from\_pin is a three-state device-enable pin, a prop delay rise is the Z-to-1 transition, and a fall is the Z-to-0 transition.
- For a disable delay, from\_pin is a three-state device-enable pin. A rise is the 0-to-Z transition of to\_pin; a fall is the 1-to-Z transition.
- For a clear delay, only the fall is considered. A fall is the 1-to-0 transition of the to\_pin. The rise event is ignored.
- For a preset delay, only the rise is considered. A rise is the 0-to-1 transition of the to\_pin. The fall event is ignored.

[Example 11-10](#) and [Example 11-11](#) show two types of reports for combinational pin delays.

### *Example 11-10 Standard Combinational Pin Delays Report*

```
PIN(Q) : out, , , , , ( , ), ( , );  
SAMPLE : CP, rise, Q, '', (1.17, 1.35), (0, 0), (0.1523, 0.0589);  
DELAY : SD, Q, preset, neg_unate, '', (0.97, 1), (0, 0), (0.1523, 0);  
DELAY : CD, Q, clear, pos_unate, '', (1, 0.75), (0, 0), (0, 0.0589);  
END_PIN Q;
```

### *Example 11-11 Combinational Pin Delays Report With a Nonlinear Delay Model*

```
PIN(Q) : out, , , , , ( , ), ( , ) ;
DELAY : SD, Q, preset, neg_unate, '', ( , ), ( , ), ( , );
      rise_propagation ( test ) :
      VALUES : 1.0, 2.1, 3.4
      rise_transition ( test ) :
      VALUES : 0.5, 0.7, 1.9
DELAY : CD, Q, clear, pos_unate, '', ( , ), ( , ), ( , );
      fall_propagation ( test ) :
      VALUES : 1.0, 2.1, 3.4
      fall_transition ( test ) :
      VALUES : 0.5, 0.7, 1.9
END_PIN Q;
```

### **Linear Sequential Delay**

This is the format for a standard sequential delay report:

```
SAMPLE : clock_pin, transition_type, to_pin,
'enable_condition', (intrinsic_rise, intrinsic_fall),
(slope_rise, slope_fall), (rise_resistance,
fall_resistance) ;
```

[Example 11-12](#) shows standard sequential pin delays.

### *Example 11-12 Standard Sequential Pin Delays*

```
PIN(Q) : out, , , , , ( , ), ( , ) ;
SAMPLE : CP, rise, Q, '', (1.16, 1.44), (0, 0), (0.0653, 0.0347) ;
END_PIN Q ;
```

### **Piecewise Linear Sequential Delay**

This is the format for a piecewise linear sequential delay report:

```
SAMPLE : clock_pin, transition_type, to_pin,
'enable_condition', (intrinsic_rise, intrinsic_fall),
(slope_rise, slope_fall), (rise_delay_intercept,
fall_delay_intercept), (rise_pin_resistance,
fall_pin_resistance), (rise_wire_resistance,
```

```

fall_wire_resistance), (rise_nonaired_twin,
fall_nonaired_twin);

```

## Nonlinear Sequential Delay

This is the format for a nonlinear sequential delay report:

```

SAMPLE : CK, rise, Q, '', ( , ), ( , ), ( , ) ;
rise_propagation ( Template8 ) :
VALUES : 1.0394 1.1740 1.6733 1.0544 1.1897 1.6885
          1.0516 1.1868 1.6852 1.0360 1.1709 1.6693
          1.0024 1.1373 1.6356
fall_propagation ( Template8 ) :
VALUES : 1.4165 1.5473 2.0101 1.5046 1.6341 2.0969
          1.5584 1.6878 2.1506 1.6100 1.7402 2.2030
          1.6393 1.7694 2.2322
rise_transition ( Template1 ) :
VALUES : 0.0606 0.6828

fall_transition ( Template1 ) :
VALUES : 0.1596 0.6372

```

## Scalable Polynomial Sequential Delay

This is the format for a scalable polynomial sequential delay report:

```

SAMPLE : CK, rise, Q, '', ( , ), ( , ), ( , ) ;
rise_propagation ( PolyTemplate8 ) :
domain(best)
orders("1, 1, 0, 0")
"1.0000000+1.0000000a+1.0000000b+1.0000000ab"
domain(typ)
orders("1, 1, 0, 0")
"3.0000000+3.0000000a+3.0000000b+3.0000000ab"
fall_propagation ( Template8 ) :
domain(best)
orders("1, 1, 0, 0")
"2.0000000+2.0000000a+2.0000000b+2.0000000ab"
domain(typ)
orders("1, 1, 0, 0")
"4.0000000+4.0000000a+4.0000000b+4.0000000ab"

```

## Pin Capacitance Ranges

This is the format for a pin capacitance ranges report:

```
PIN(A) : in, float, , , float, float;  
        RISE_CAPACITANCE_RANGE(float, float)  
        FALL_CAPACITANCE_RANGE(float, float)  
END_PIN A ;
```

## Cell Degradation Design Rule

The cell degradation design rule defines the maximum capacitive load a cell can drive without cell performance degradation.

This is the syntax for the cell degradation information:

```
cell_degradation (template_name) ;  
    INDEX_1 (float ... float) ;  
    VALUES : (float ... float) ;
```

This is an example of the cell degradation information in a library report after you issue the `report_lib -timing` command:

```
cell_degradation (template_1) :  
    INDEX_1      :   1.0  2.5  3.75  
    VALUES       :   1.0394  1.1740  1.6733  
                  1.0516  1.1868  1.6852  
                  1.0024  1.1373  1.6356
```

## Timing Arcs Report

Use the `-timing_arcs` option to generate a report like the one shown in [Example 11-13](#).

### *Example 11-13 Reporting Timing Arcs Information*

```
lc_shell> report_lib -timing_arcs  
...  
...
```

Attributes:

```
af - active falling
ah - active high
al - active low
ar - active rising
b - black box (function unknown)
ce - clock enable
d - dont_touch
mo - map_only
p - preferred
r - removable
s - statetable
sa0 - dont_fault stuck-at-0
sa1 - dont_fault stuck-at-1
sa01 - dont_fault both stuck-at-0 and stuck-at-1
sz - use_for_size_only
t - test cell
u - dont_use
```

Cell	Attributes	#	Arc	Arc Pins		When
			Sense/Type	From	To	
<hr/>						
EO		0	pos unate	A	Z	
		1	pos unate	B	Z	
		2	neg unate	A	Z	
		3	neg unate	B	Z	
EO1		0	pos unate	A	Z	
		1	pos unate	B	Z	
		2	neg unate	A	Z	
		3	neg unate	B	Z	
EOa		0	pos unate	A	Z	

## Timing Label Reports

Library Compiler reports all labels associated with the timing arcs when you use the `-timing_label` option. This is the format for a timing label report:

```
lc_shell> report_lib -timing_label
```

Cell	Arc	Pins	Labels
	From	To	
<hr/>			

cell1	X[0]	C	X0_C
	X[1]	C	X1_C
	X[2]	C	X2_C
	X[3]	C	X3_C
	A	D	A_D

## Timing Report Statistics

The summary section at the end of a library timing report contains simple statistics like the following:

```
Number of library cells: 18
Number of reported cells: 16
Number of reported scaled cells: 6
```

## Noise Reports

Use the following options to generate noise reports:

`-noise`

Displays noise immunity, noise propagation, and current-voltage (I-V) characteristic information, as shown in [Example 11-14](#).

`-noise_arcs`

Displays noise arc information, as shown in [Example 11-15](#).

*Example 11-14 Noise Immunity, Noise Propagation, and I-V Characteristics in a Noise Report*

```
lc_shell> report_lib -noise
```

Noise Immunity Lookup Table Template:

```
Template_name
-----
mynoise_reject
  VARIABLE_1: input_noise_width
  VARIABLE_2: total_output_net_capacitance
  INDEX_1:  0.0000  0.1000  0.3000  1.0000  2.0000
```

```

INDEX_2: 0.0000 0.1000 0.3000 1.0000 2.0000
mynoise_reject_outside_rail
VARIABLE_1: input_noise_width
VARIABLE_2: total_output_net_capacitance
INDEX_1: 0.0000 0.1000 2.0000
INDEX_2: 0.0000 0.1000 2.0000

```

I-V Characteristics Lookup Table Template:

```

Template_name
-----
my_current_low
    VARIABLE_1: iv_output_voltage
    INDEX_1: -1.0000 -0.1000 0.0000 0.1000 0.8000 1.6000 2.0000
my_current_high
    VARIABLE_1: iv_output_voltage
    INDEX_1: -1.0000 -0.1000 0.3000 0.1000 0.5000 0.8000 1.6000 1.7000
2.0000

```

Noise Propagation Lookup Table Template:

```

Template_name
-----
my_noise_propagation
    VARIABLE_1: total_output_net_capacitance
    VARIABLE_2: input_noise_width
    VARIABLE_3: input_noise_height
    INDEX_1: 0.0000 1.0000 2.0000 3.0000 4.0000 5.0000 6.0000
    INDEX_2: 0.0000 1.0000 2.0000 3.0000

INDEX_3: 0.0000 1.0000 2.0000 3.0000

CELL(INV): 1;

PIN(A): in, 1, 1, , , ;
hyperbolic_noise_low : (0.400000 1.100000 0.100000) ;
hyperbolic_noise_high : (0.300000 0.900000 0.100000) ;
hyperbolic_noise_above_high : (0.100000 0.300000 0.010000) ;
hyperbolic_noise_below_low : (0.100000 0.300000 0.010000) ;
END_PIN A;

PIN(Y): out, 0, 10, , , , ;
steady_state_resistance_low : 1100.000000 ;
steady_state_resistance_high : 1500.000000 ;
steady_state_resistance_above_high : 200.000000 ;
steady_state_resistance_below_low : 100.000000 ;
```

```

PIN(y): in, 1, , , , ;
noise_immune_high ( my_noise_reject ) :
VALUES :   1.3000  0.8000  0.7000  0.6000  0.5500  1.5000
           0.9000  0.8000  0.6500  0.6000  1.5000  0.9000
           0.8000  0.6500  0.6000  1.5000  0.9000  0.8000
           0.6500  0.6000  1.5000  0.9000  0.8000  0.6500
           0.6000

noise_immune_low ( my_noise_reject ) :
VALUES :   1.5000  0.9000  0.8000  0.6500  0.6000  1.5000
           0.9000  0.8000  0.6500  0.6000  1.5000  0.9000
           0.8000  0.6500  0.6000  1.5000  0.9000  0.8000
           0.6500  0.6000  1.5000  0.9000  0.8000  0.6500
           0.6000

noise_immune_below_low ( my_noise_reject_outside_rail ) :
VALUES :   1.0000  0.8000  0.5000  1.0000  0.8000  0.5000
           1.0000  0.8000  0.5000

noise_immune_above_high ( my_noise_reject_outside_rail ) :
VALUES :   1.0000  0.8000  0.5000  1.0000  0.8000  0.5000
           1.0000  0.8000  0.5000

steady_state_current_low ( my_current_low ) :
VALUES :   0.1000  0.0500  0.0000 -0.1000 -0.2500 -1.0000
           -1.8000

steady_state_current_high ( my_current_high ) :
VALUES :   2.0000  1.8000  1.7000  1.4000  1.0000  0.5000
           0.0000 -0.1000 -0.8000

END_PIN y;

END_CELL INV;

```

Number of library cells: 1  
Number of reported cells: 1  
1

### *Example 11-15 Noise Arcs in a Noise Report*

```
lc_shell> report_lib -noise_arcs
```

```
...
```

Components:

Arc Types:

```
nih - noise_immunity_high
nil - noise_immunity_low
niah - noise_immunity_above_high
nibl - noise_immunity_below_low
hnh - hyperbolic_noise_high
hn1 - hyperbolic_noise_low
hnah - hyperbolic_noise_above_high
hnbl - hyperbolic_noise_below_low
ssch - steady_state_current_high
sscl - steady_state_current_low
ssct - steady_state_current_tristate
pnwh - propagated_noise_width_high
pnwl - propagated_noise_width_low
pnhh - propagated_noise_height_high
pnhl - propagated_noise_height_low
pnwah - propagated_noise_width_above_high
pnwbl - propagated_noise_width_below_low
pnhah - propagated_noise_height_above_high
pnhbl - propagated_noise_height_below_low
```

Cell	Arc		Arc Pins
	#	TypeFromToWhen	
INV	0	hnhA	
	1	hn1A	
	2	hnahA	
	3	hmbla	
	4	nihAY	
	5	nilAY	
	6	niahAY	
	7	niblAY	
	8	sschAY	
	9	ssclAY	
	10	pnwlAY	
	11	pnhlAY	
	12	pnwblAY	

## Power Reports

Use the following options to generate power reports:

-power

Displays power-related information, including internal power, leakage power, and multiple power supplies.

-power\_label

Displays all the power label information.

## Library Defaults

Power tools use the default values for power-related attributes if they are not defined in the technology library.

The library defaults section of the library report lists what default is being used for each power-related attribute. The power-related attributes not defined in the library are blank in the power report.

**Example 11-16** represents the power attribute defaults section in a power report.

### *Example 11-16 Power Attribute Defaults*

Power Attribute Defaults:

Attribute	Default
<hr/>	
default_cell_leakage_power	1
default_leakage_power_density	0

## Library-Level Templates

The internal power models have library-level templates, which are written out in the power report. This is the format of the internal power lookup table template information:

```
Template_name
-----
name of the template
    VARIABLE_1: variable_1
    VARIABLE_2: variable_2
    VARIABLE_3: variable_3
    INDEX_1 : float ... float
    INDEX_2 : float ... float
    INDEX_3 : float ... float
```

*variable\_2* and *variable\_3* are optional for all tables except the three-dimensional tables used to model output load. The lines containing *INDEX\_2* and *INDEX\_3* data are optional.

[Example 11-17](#) shows an example of a lookup table template for the internal power models.

### *Example 11-17 Internal Power Template*

Internal Power Lookup Table Template:

```
Template_name
-----
power_load_64
    VARIABLE_1: input_transition_time
    VARIABLE_2: total_output_net_capacitance
    INDEX_1:  0.0400  0.1200  0.2000  0.4000  0.6000  0.8000
    INDEX_2:  0.0638  0.1276  0.2552  0.5104  1.0208  2.0416
power_3_d
    VARIABLE_1: total_output_net_capacitance
    VARIABLE_2: equal_or_opposite_output_net_capacitance
    VARIABLE_3: input_transition_time
    INDEX_1:  0.1000  5.0000  20.0000
    INDEX_2:  0.1000  5.0000  20.0000
    INDEX_3:  0.1000  0.5000  0.8000
```

Library Compiler lists power information in terms of internal power and leakage. Each category uses a different format. You generate this report by using the `-power` option for `report_lib` command.

The leakage power format looks like the following:

`LEAKAGE_POWER : when_string, value;`

*Example 11-18 Reporting Power Modeling Information*

```
lc_shell> report_lib -power
```

Attributes:

a - average power specification  
i - internal power  
l - leakage power  
rf - rise and fall power specification

Cell	#	Attr	Power		Source of path	when
			Toggling	pin		
EO	0	i,a	Z		B	A'
	1	i,a	Z		B	A
	2	i,a	Z		B	
	3	i,a	Z		A	B'
	4	i,a	Z		A	B
E01	0	i,a	Z		B	A'
	1	i,a	Z		B	A
	2	i,a	Z		A	
	3	i,a	Z		B	
	4	i,a	Z		A	B'

## Library-Level Multiple Power Supply Information

This section shows the format and provides examples of multiple power supply information at the library level, including

- `power_supply` group information
- `power_rail` information about the operating conditions

This is the report format for information about the power\_supply group:

```
Power Supply Group
  default_power_rail : name

  Power Rail          Value
-----
  name_1             float
  name_2             float
```

This is the format for information about the power rail in the operating\_conditions group:

```
Name      :          operating condition name
  Power Rail          Value
-----
  name_1             float
  name_2             float
```

Example 11-19 shows a power supply report.

#### *Example 11-19 Power Supply Information Report*

```
Power Supply Group:
  default_power_rail : name

  Power Rail          Value
-----
  VDD1               5.0
  VDD2               3.30
```

Example 11-20 shows a power rail report.

#### *Example 11-20 Power Rails for Operating Conditions Report*

```
Name      :          WCCOM
  Power Rail          VALUE
-----
```

VDD1	4.80
VDD2	2.90

## Cell-Level Information

### Leakage Power

This is the format for cell leakage information at the cell level:

```
LEAKAGE_POWER : when_string, value;
```

[Example 11-21](#) shows a cell leakage power report.

#### *Example 11-21 Cell Leakage Power Report*

```
CELL(cell4): 2, 0.02;
  LEAKAGE_POWER : 'A', 0.05;

CELL(cell5): 2, 0.02;
  LEAKAGE_POWER : 'A' & B', 0.05;
  LEAKAGE_POWER : 'A & B'', 0.05;

PIN(A): in, 0.032, , , , ;
END_PIN A;

PIN(B): in, 0.033, , , , ;
END_PIN B;

PIN(Z): out, 0, , , 0.5742, , ;
END_PIN Z;
END_CELL cell5;
```

### Multiple Power Supply Information

This is the format of multiple power supply information at the cell level:

```
Name      : cell name
RAIL CONNECTION (CONN NAME) : RailpowerName
-----
name_1          float
```

*name\_2*                                   *float*

**Example 11-22** shows a cell-level multiple power supply report.

### *Example 11-22 Multiple Power Supply Information Report*

```
Name              :                 IBUF1
RAIL CONNECTION (PV1)  : VDD1
RAIL CONNECTION (PV2)  : VDD2
```

## **Pin-Level Information**

### **Internal Power Data**

The power report contains the following area and power information for each cell:

```
CELL(cell_name) : area, ...;
  AREA : area_value

  PIN(pin_name_1) : attributes;
  ...power_information...
  END_PIN pin_name_1;

  PIN(pin_name_n) : attributes;
  ...power_information...
  END_PIN pin_name_n;

END_CELL cell_name;
```

**Example 11-23** is an example of a CMOS cell description in a power report.

### *Example 11-23 Cell Description in a Power Report*

```
CELL(cell1): 2, 0.02;
  LEAKAGE_POWER : 'A', 0.05;
  PIN(A): in, 0.029, , , , ;
  INTERNAL_POWER: , , 'C & D';
    rise_power ( power_ramp ) :
  VALUES : 0.5342 0.7481 1.0494 1.9451 2.9057 3.8904
```

```

    fall_power ( power_ramp ) :
VALUES : 0.5342 0.7481 1.0494 1.9451 2.9057 3.8904

END_PIN A;
...
END_CELL cell1;

CELL(cell2): 2, 0.02;
  LEAKAGE_POWER : 'A' & B', 0.05;
  LEAKAGE_POWER : 'A & B'', 0.05;

PIN(A): in, 0.032, , , , ;
END_PIN A;

PIN(B): in, 0.033, , , , ;
END_PIN B;

PIN(Z): out, 0, , , 0.5742, , ;
END_PIN Z;
END_CELL cell2;

```

## Multiple Power Supply Signal Information

The power report generates signal-level information for all pins in cells with multiple power supplies.

This is the format of the multiple power supply information at the pin level:

```

PIN (input_pin_name) :
  INPUT_SIGNAL_LEVEL (input_pin_name) : power_rail_name
END_PIN (input_pin_name)
PIN (output_pin_name) :
  OUTPUT_SIGNAL_LEVEL (input_pin_name) : power_rail_name
END PIN (output_pin_name);

```

If the pin is an I/O pin, the power report generates signal-level information as follows:

```

PIN (inout_pin_name) :
    INPUT_SIGNAL_LEVEL (inout_pin_name) : power_rail_name
    OUTPUT_SIGNAL_LEVEL (inout_pin_name) ; power_rail_name
END_PIN (inout_pin_name) ;

```

## **Example**

```

PIN (A) :
    INPUT_SIGNAL_LEVEL (A) : VDD1
END_PIN (A) ;

PIN (Z) :
    OUTPUT_SIGNAL_LEVEL (Z) : VDD2
END_PIN (Z) ;

PIN (Z1) ;
    INPUT_SIGNAL_LEVEL (Z1) : VDD1
    OUTPUT_SIGNAL_LEVEL (Z1) : VDD2
END_PIN (Z1) ;

```

## **Electromigration Reports**

Use the following option to generate an electromigration report.

-em

Displays electromigration-related information.

## **Library Defaults**

[Example 11-24](#) shows the timing attribute defaults section in an electromigration report.

### *Example 11-24 Timing Attribute Defaults*

Timing Attribute Defaults:

Attribute	Default
<hr/>	
em_temp_degradation_factor	: 0.0

## Library-Level Templates

The electromigration data has library-level templates, which are written out in the electromigration report.

This is the format of the lookup table electromigration template information:

```
Template_name
-----
name of the template
    VARIABLE_1: variable_1
    VARIABLE_2: variable_2
    INDEX_1 : float ... float
    INDEX_2 : float ... float
```

[Example 11-25](#) shows examples of the lookup table templates for the electromigration models.

## Pin-Level Information

The electromigration report includes a PIN section for every pin in the cell.

```
CELL(cell_name) : area, ...;
    PIN(pin_name) : attributes,
        electromigration_information
    END_PIN pin_name;
END_CELL cell_name;
```

### *Example 11-25 Template Examples*

```
CELL(sample): 2;
    PIN(y): out, 0, , , 1.812, , ;
    ELECTROMIGRATION: a;
        em_max_toggle_rate ( output_by_cap_and_trans ) :
            VALUES : 2.0000 1.0000 0.5000 1.5000 0.7500 0.3300 1.0000 0.5000 0.1500

    ELECTROMIGRATION: b;
        em_max_toggle_rate ( output_by_cap_and_trans ) :
```

```

VALUES : 2.0000 1.0000 0.5000 1.5000 0.7500 0.3300 1.0000 0.5000 0.1500

ELECTROMIGRATION: c;
  em_max_toggle_rate ( output_by_cap_and_trans ) :
VALUES : 2.0000 1.0000 0.5000 1.5000 0.7500 0.3300 1.0000 0.5000 0.1500

END_PIN Y;

```

## Functionality Reports

Functionality reports provide statetable and VHDL name information.

### Statetable Report

Use the following options to generate statetable reports.

**-table**

Displays the statetable data in a compact form. [Example 11-26](#) provides an example of this option.

**-full\_table**

Displays the statetable model in a table form and memory information about the ports. [Example 11-27](#) provides an example of this option.

**Note:**

The statetable options are enabled only when the technology library is read in. The asterisk suffix (\*) in a statetable report denotes a delayed port.

*Example 11-26 Statetable Report in Compact Form.*

```
lc_shell> report_lib -table asic.db dffz
...
```

```

Statetable descriptions
CELL(dffz):

PIN(QN) : CP*, QN*, D, CP
    Table : NNNNNNNNNHHNLNHN
END_PIN QN;

PIN(Q) :
    state_function : QN'
END_PIN Q;

PIN(QNZ) :
    state_function : QN
    three_state : EN
END_PIN QZ;
END_CELL dffz;

```

*Example 11-27 Statetable Report in Table Form*

```

lc_shell> report_lib -full_table asic.db dff
...
Statetable descriptions:
CELL(dff):

PIN(QN) : CP*, QN*, D, CP
Table :
 0000      N
 0001      H
 0010      N
 0011      L
 0100      N
 0101      H
 0110      N
 0111      H
 1000      N
 1001      N
 1010      N
 1011      N
 1100      N
 1101      N
 1110      N

```

```

      1111      N
END_PIN QN;
END_CELL dff;
```

## VHDL Name Report

Use the `-vhdl_name` option to generate the VHDL name report.

`-vhdl_name`

Displays the list of changed VHDL names at the end of the report.

[Example 11-28](#) provides an example of this option.

### *Example 11-28 Table of Changed VHDL Names*

```
lc_shell> report_lib -vhdl_name my_lib
...
```

Cell	PortVHDL name
<hr/>	
INV	ININb
XOR	XORB
TINV	T_INV
	INA
NAND4	NAND4b
	Ab
	BA
	CCb

## Test Cell Information

Additional error messages and warnings result when the timing and test cell of the sequential model are closely checked.

It is important to update the technology libraries by accurately defining the full function of each library cell and the nontest mode of each test cell. Use the statetable format and, if possible, the ff/latch format. If the ff/latch format is adequate and you do not supply statetable information, you risk specification error.

[Example 11-29](#) is an example of a test cell report.

*Example 11-29 Test Cell Report*

```
lc_shell> report_lib -table asic.db dff_test
...
CELL(dff_test):
  PIN(QN) : CP*, S, D0, D1, CP
  TABLE : NHNLNHNHNNNNNNNNNN
  END_PIN QN;

  TEST_CELL
    PIN(QN) : CP*, D0, CP
    TABLE : NHNLNHNHNNNNNNNNNN
    END_PIN QN;
  END_TEST_CELL;
END_CELL dff_test;
```

---

## FPGA Reports

Use the following option to generate the FPGA report:

`-fpga`

Displays the library defaults and, if they exist, the part group and the I/O attributes. [Example 11-30](#), [Example 11-31](#), and [Example 11-32](#) show these three sections in the order they appear in a report:

## Library Defaults

*Example 11-30 Library Defaults Section of the FPGA Report*

```
lc_shell> report_lib -fpga sample
...
*****
Report : library
```

```

Version: 2003.12
Library: sample
Date    : Tue Oct  2 15:22:55 2003
*****
Library Type          : FPGA Technology
Tool Created          : 2002.05
Date Created          : Not Specified
Library Version       : Not Specified
Time Unit             : 1ns
default_part ("AUTO", "-5")
FPGA Technology       : "my_technology"

```

## Parts Information

**Example 11-31** shows the PART section of the FPGA report, which lists the number of pins, rows, columns, flip-flops, blockrams, and constraints, as well as the speed grades, if they are present in the library.

### *Example 11-31 PART Section of the FPGA Report*

```

PART(V100CS144)
pin_count : 94;
num_rows : 20;
num_cols : 30;
num_ffs : 2760;
num_blockrams : 10;
valid_speed_grades ("-6", "-5", "-4");
valid_step_levels ( "STEP0", "STEP1", "STEP2" )
default_step_level : "STEP0" ;
max_count(BUFGTS, 4);
END PART V100CS144;

```

## Cell-Level Information

**Example 11-32** shows the information contained in the cell-level section of the FPGA report.

### *Example 11-32 Cell-Level Section of the FPGA Report*

FPGA IO Attributes:

Cell	io_type	drive_type	slew_type	edif_name
IBUF	LVTTL			IBUF
IBUF_LVCMOS2	LVCMSO2			IBUF
IBUF_PCI33_3	PCI33_3			IBUF
IBUF_PCI33_5	PCI33_5			IBUF
IBUF_PCI66_3	PCI66_3			IBUF
OBUF	LVTTL	12	fast	OBUF
OBUF_S_2	LVTTL	2	fast	OBUF
OBUF_S_4	LVTTL	4	none	OBUF
OBUF_S_6	LVTTL	6	none	OBUF

#### FPGA Resource Usage:

Cell	resource_name	number_resource
IBUF	DCM	6
IBUF_LVCMOS2	DCM	16
IBUF_PCI33_3	DCM	26
IBUF_PCI33_5	DCM	36
IBUF_PCI66_3	DCM	46
OBUF	DCL	6
OBUF_S_2	DCL	16
OBUF_S_4	DCL	26
OBUF_S_6	DCL	26

## User-Defined Data Reports

Use the following option to generate a user-defined data report.

*-user\_defined\_data*

Displays all the user-defined attributes and groups in the technology library.

Note:

When you specify both the `-user_defined_data` option and a list of cells (see “[Comprehensive Reports](#)” on page 11-45), the report displays only the user-defined attributes and groups for the specified cells.

*Example 11-33 User-Defined Data in a Report*

```
lc_shell> report_lib -user_defined_data sample
...
*****
Report : library
Library: sample
Version: 2003.12
Date   : Fri Sep 26 09:11:17 2003
*****
*****
```

Library Type : Technology  
Tool Created : 2003.12  
Date Created : 06-JUL-90  
Library Version : 0.930000  
User-defined Data Template:

```
-----
-----
timing
  timing_attr : string
  timing_udg
    timing_udg_attr : string
pin
  float_attr : float
  pin_attr : string
  pin_udg1
    pin_udg_attr : string
  pin_udg2
    pin_udg_attr : string
cell
  cell_attr : string
  cell_udg1
    cell_udg_attr : string
  cell_udg2
    cell_udg_attr : string
```

```

library
    lib_attr : string
    float_attr : float
    lib_udg1
        lib_udg_attr : string
    lib_udg2
        lib_udg_attr : string
User-defined Data:

-----
-----
LIBRARY(sample) :
    float_attr : 1e-09
    lib_attr : lib_test
    LIB_UDG1(lib_udg1) :
        lib_udg_attr : test
END_LIB_UDG1 lib_udg1;
    LIB_UDG2(lib_udg2) :
        lib_udg_attr : test
END_LIB_UDG2 lib_udg2;
    CELL(INV) :
        cell_attr : inv_test
    CELL_UDG1(cell_udg1) :
        cell_udg_attr : test
END_CELL_UDG1 cell_udg1;
    CELL_UDG2(cell_udg2) :
        cell_udg_attr : test
END_CELL_UDG2 cell_udg2;
    PIN(A) :
        float_attr : 0.123457
        pin_attr : pin_test
        PIN_UDG1(pin_udg1) :
            pin_udg_attr : test
        END_PIN_UDG1 pin_udg1;
    END_PIN A;
    PIN(Y) :
        PIN_UDG2(pin_udg2) :
            pin_udg_attr : test
        END_PIN_UDG2 pin_udg2;
    TIMING() :
        timing_attr : timing_test
        TIMING_UDG(timing_test) :
            timing_udg_attr : timing_udg_test
        END_TIMING_UDG timing_test;
    END_TIMING ;

```

```

    END_CELL INV;
CELL(INV1) :
    CELL_UDG1(cell_udg1) :
        cell_udg_attr : test
    END_CELL_UDG1 cell_udg1;
    CELL_UDG2(cell_udg2) :
        cell_udg_attr : test
    END_CELL_UDG2 cell_udg2;
PIN(A) :
    PIN_UDG1(pin_udg1) :
        pin_udg_attr : test
    END_PIN_UDG1 pin_udg1;
END_PIN A;
PIN(Y) :
    PIN_UDG2(pin_udg2) :
        pin_udg_attr : test
    END_PIN_UDG2 pin_udg2;
END_PIN Y;
END_CELL INV1;
END_LIBRARY sample;

```

---

## Comprehensive Reports

Use the following options to generate comprehensive library reports.

**-all**

Enables all report options.

**{list of cells}**

A list of cells separated by commas. The report will include data only for the specified set of cells.

*Example 11-34 Command for Generating a Report of a List of Cells*

```
lc_shell> report -lib mylib {AND, OR}
```



# 12

## Building Environments

---

Variations in operating temperature, supply voltage, and manufacturing process cause performance variations in electronic networks.

Library Compiler gives you the ability to model these variations. Then Design Compiler 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 Power Supply Cells
- Defining Wire Load Groups
- Selecting Wire Load Groups Automatically
- Specifying Delay Scaling Attributes

---

## Library-Level Default Attributes

Global default values 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;
```

Library Compiler issues a warning if the library has `cell_leakage_power` information but does not have the `default_cell_leakage_power` attribute defined.

Library Compiler looks for a definition for both the `default_cell_leakage_power` and `default_leakage_power_density` attributes. (For details about the `default_leakage_power_density` attribute, see “[default\\_leakage\\_power\\_density Simple Attribute](#)” on page 12-4). In checking these attributes, Library Compiler proceeds in the following manner:

- If you have not defined both attributes, Library Compiler issues a warning and sets the value for both attributes to 0.0.

- If you define both attributes, Library Compiler takes no action.
- If you define the `default_cell_leakage_power` attribute but not the `default_leakage_power_density` attribute, Library Compiler issues a warning and creates a `default_leakage_power_density` attribute set to 0.0.
- If you define the `default_leakage_power_density` attribute but not the `default_cell_leakage_power` attribute, Library Compiler issues a warning and creates a `default_cell_leakage_power` attribute set to 0.0.

## **default\_leakage\_power\_density Simple Attribute**

This library-level attribute provides the mean static leakage power per area unit in the given technology. This attribute must be a nonnegative floating-point number. If it is not defined, this attribute defaults to 0.0.

### **Example**

```
default_leakage_power_density : 0.5;
```

Library Compiler looks for a definition for both the `default_leakage_power_density` and `default_cell_leakage_power` attributes. (For details about the `default_cell_leakage_power` attribute, see “[default\\_cell\\_leakage\\_power Simple Attribute](#)” on page 12-3). In checking for these attributes, Library Compiler proceeds in the following manner:

- If you have not defined both attributes, Library Compiler issues a warning and sets the value for both attributes to 0.0.
- If you define both attributes, Library Compiler takes no action.

- If you define the `default_leakage_power_density` attribute but not the `default_cell_leakage_power` attribute, Library Compiler issues a warning and creates a `default_cell_leakage_power` attribute set to its default, 0.0.
- If you define the `default_cell_leakage_power` attribute but not the `default_leakage_power_density` attribute, Library Compiler issues a warning and creates a `default_leakage_power_density` attribute set to its default, 0.0.

---

## 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. The CMOS nonlinear delay model does not support default timing attributes. Use only the default attributes that apply to your timing model, which can be one of the following:

- CMOS generic delay model
- CMOS piecewise linear delay model

See Chapter 2, “Delay Models,” for details of these delay models.

## All Delay Models

These are the defaults that apply to all pins in a library. The attributes described in this section apply to all delay models.

```

default_inout_pin_cap : valuefloat ;

Sets a default value for capacitance for all I/O pins in the
library.

default_input_pin_cap : valuefloat ;

Sets a default value for capacitance for all input pins in the
library.

default_output_pin_cap : valuefloat ;

Sets a default value for capacitance for all output pins in the
library.

default_max_fanout : valuefloat ;

Sets a default value for max_fanout for all output pins in the
library.

default_max_transition : valuefloat ;

Sets a default value for max_transition for all output pins in
the library.

default_fanout_load : valuefloat ;

Sets a default value for fanout_load for all input pins in the
library.

```

The following example shows the default pin attributes in a CMOS library:

#### *Example 12-1 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 ;
    ...
}
```

## CMOS Generic Delay Model

These are the default timing attributes for a library that uses a CMOS generic delay model. In each attribute, *value* is a floating-point number.

```
default inout pin fall res : valuefloat ;
```

Sets a default value for `fall_resistance` for all I/O pins in a library.

```
default output pin fall res : valuefloat ;
```

Sets a default value for `fall_resistance` for all output pins in the library.

```
default inout pin rise res : valuefloat ;
```

Sets a default value for `rise_resistance` for all I/O pins in the library.

```
default output pin rise res : valuefloat ;
```

Sets a default value for `rise_resistance` for all output pins in the library.

```
default slope fall : valuefloat ;
```

Sets a default value for `slope_fall` for all output pins in the library.

```
default slope rise : valuefloat ;
```

Sets a default value for `slope_rise` for all output pins in the library.

```
default_intrinsic_fall : valuefloat ;
```

Sets a default value for `intrinsic_fall` for all timing arcs in the library.

```
default_intrinsic_rise : valuefloat ;
```

Sets a default value for `intrinsic_rise` for all timing arcs in the library.

**Example 12-2** sets the default timing attributes for a generic delay model.

### *Example 12-2 Setting Standard Timing Default Attributes*

```
library (example) {  
    ...  
    /* default timing attributes */  
    default inout pin fall res : 12.0;  
    default output pin fall res : 12.0;  
    default inout pin rise res : 15.2;  
    default output pin rise res : 15.3;  
    default intrinsic fall : 1.0;  
    default intrinsic rise : 1.0;  
    ...  
}
```

## Piecewise Linear Delay Model

These are the default timing attributes for a library that uses a piecewise linear delay model. In each attribute, `value` is a floating-point number.

```
default rise delay intercept : valuefloat ;
```

Sets a default value for `rise_delay_intercept` on all output pins in the library.

```

default_fall_delay_intercept : valuefloat ;

Sets a default value for fall_delay_intercept on all output
pins in the library.

default_rise_pin_resistance : valuefloat ;

Sets a default value for rise_pin_resistance on all output
pins in the library.

default_fall_pin_resistance : valuefloat ;

Sets a default value for fall_pin_resistance on all output
pins in the library.

default_intrinsic_fall : valuefloat ;

Sets a default value for intrinsic_fall for all timing arcs in
the library.

default_intrinsic_rise : valuefloat ;

Sets a default value for intrinsic_rise for all timing arcs in
the library.

```

**Example 12-3** illustrates the default timing attributes setting for a piecewise linear timing delay model.

### *Example 12-3 Setting Piecewise Linear Default Timing Attributes*

```

library (example) {
    ...
    /* default timing attributes */
    default_rise_delay_intercept : 1.0;
    default_fall_delay_intercept : 1.0;
    default_rise_pin_resistance : 1.5;
    default_fall_pin_resistance : 0.4;
    default_intrinsic_fall      : 1.0;
    default_intrinsic_rise      : 1.0;
    ...
}

```

---

## Setting Wire Load Defaults

Use the following library-level attributes to set wire load defaults.

### **default\_wire\_load Attribute**

Assigns the default values 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 or floorplan manager 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_resistance : value;
```

## **Example**

```
default_wire_load_area : 0.33;
```

---

## **Setting Other Environment Defaults**

Use the following library-level attributes to set other environment defaults.

## **default\_max\_utilization Attribute**

Defines the upper limit placed on the utilization of a chip.

## Syntax

```
default_max_utilization : value;
```

## Example

```
default_max_utilization : 0.08;
```

Utilization is the percentage of total die size taken up by the total cell area. For example, if the total cell area is 100,000 and the total die size on which these cells will be placed is 150,000, then utilization is 66.6 percent. The utilization of a chip implicitly defines how much room there is for routing between layers of silicon.

Generally, the higher the utilization you place on a chip, the more difficult a design is to route, because there is less physical area available for doing the routing.

See the Design Compiler reference manuals for more information on using `default_max_utilization`.

## **default\_min\_porosity Attribute**

Defines the routability optimization constraint, with this particular library as the target library for optimizing a design for routability.

## Example

```
default_min_porosity : 1.03 ;
```

You can override `default_min_porosity` by using the `set_min_porosity` command in `dc_shell`.

## **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 12-15.

### **Syntax**

```
default_operating_conditions : operating_condition_name;
```

### **Example**

```
default_operating_conditions : WCCOM1;
```

## **default\_connection\_class Attribute**

Sets a default value for connection\_class for all pins in a library.

### **Example**

```
default_connection_class : name1 [name2 name3 ...];
```

---

## **Examples of Library-Level Default Attributes**

[Example 12-4](#) 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 12-4 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 intrinsic fall : 1.0;
default intrinsic rise : 1.0;
default output pin_cap : 0.0;
default slope fall : 0.0;
default slope rise : 0.0;
default fanout load : 1.0;
default max fanout : 10.0;

/* default timing attributes */

default inout pin fall res : 0.2;
default output pin fall res : 0.2;
default inout pin rise res : 0.33;
default output pin rise res : 0.4;

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:

`calc_mode : name ;`

An optional attribute that you use to identify the associated process mode.

`power_rail (name , value)`

Identifies all power supplies that have the nominal operating conditions (defined in the `operating_conditions` group) and the nominal voltage values.

`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.

`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 6 in *Library Compiler User Guide: Methodology and Modeling Functionality in Technology Libraries* for more information.

```
tree_type : model ;
```

The definition for the environment interconnect model. Design Compiler 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.

For additional information about the interconnect models, see [“Interconnect Delay” on page 3-15](#).

---

## Determining Operating Conditions

Design Compiler determines which operating conditions to use, according to the following hierarchy:

1. The `operating_conditions` group defined 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
- 

## Defining Operating Conditions at Runtime

To define an `operating_conditions` group at runtime, use the `set_operating_conditions` command in `dc_shell`.

### Syntax

```
set_operating_conditions operating_cond_name
```

`operating_cond_name`

Specifies the name of an `operating_conditions` group defined in the library. For example, to use the operating conditions group `MY_BEST`, enter

```
dc_shell> set_operating_conditions MY_BEST
```

For additional information, see the `set_operating_conditions` command in the Synopsys man pages.

---

## **timing\_range Group**

A `timing_range` group models statistical fluctuations in the defined operating conditions defined for your design during the optimization process. A `timing_range` group is defined at the library-group level:

```
library (lib_name) {  
    timing_range (name) {  
        ... timing_range description ...  
    }  
}
```

A `timing_range` group defines two multipliers that scale the signal arrival times computed by the timing analyzer when it evaluates timing constraints. In the following attributes, *multiplier* is a floating-point number:

```
faster_factor : multiplier;
```

Scaling factor applied to the signal arrival time to model the fastest-possible arrival time.

```
slower_factor : multiplier;
```

Scaling factor applied to the signal arrival time to model the slowest-possible arrival time.

**Example 12-5** describes the `SLOW_RANGE` and `FAST_RANGE` timing ranges.

### *Example 12-5 Defining Timing Ranges*

```
library (example) {  
    ...  
    timing_range (SLOW_RANGE) {  
        faster_factor : 1.05;  
        slower_factor : 1.2;  
    }  
    timing_range (FAST_RANGE) {  
        faster_factor : 0.90;  
        slower_factor : 0.96;  
    }  
    ...  
}
```

In the `SLOW_RANGE` timing range, the possible delays are from 1.05 to 1.2 times the delay calculated by the timing analyzer. In the `FAST_RANGE` timing range, the possible delays are 0.90 to 0.96 times the delay calculated by the timing analyzer.

---

## 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:

The modeling method previously described produces accurate results in cases in which all the delays have a linear dependency on the operating parameters. In cases in which delays do not scale linearly with operating conditions, this method provides only a first-order approximation. Remember that timing ranges influence constraint evaluation only; they do not affect timing reports in Design Compiler. You can obtain information on 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
<hr/>			
FAST_RANGE	test	0.9600	0.9000
SLOW_RANGE	test	1.2000	1.0500

---

# Defining Power Supply Cells

Use the power\_supply group to model multiple power supply cells.

---

## power\_supply group

The power\_supply group captures all nominal information on voltage variation. It is defined before the operating\_conditions group and before the cell groups. All the power supply names defined in the power\_supply group exist in the operating\_conditions group.

Define the power\_supply group at the library level.

### Syntax

```
power_supply () {  
    default_power_rail : string ;  
    power_rail (string, float) ;  
    power_rail (string, float) ;  
    ...  
}
```

### Example

```
power_supply () {  
    default_power_rail ; VDD0;  
    power_rail (VDD1, 5.0) ;  
    power_rail (VDD2, 3.3) ;  
}
```

---

## 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 in order 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 ( fanoutint, lengthfloat, average_capacitancefloat \ standard_deviationfloat , number_of_netsint );`

`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.

The Floorplan Manager tool is the only Synopsys product that uses `average_capacitance`, `standard_deviation`, and `number_of_nets`. For more information, see the *Floorplan Manager User Guide*.

When you create a wire load manually, define only *fanout* and *length*. When you generate the wire load with the `create_wire_load` command, the *fanout* and *length* values are generated automatically.

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 Design Compiler 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,...) ;  
}
```

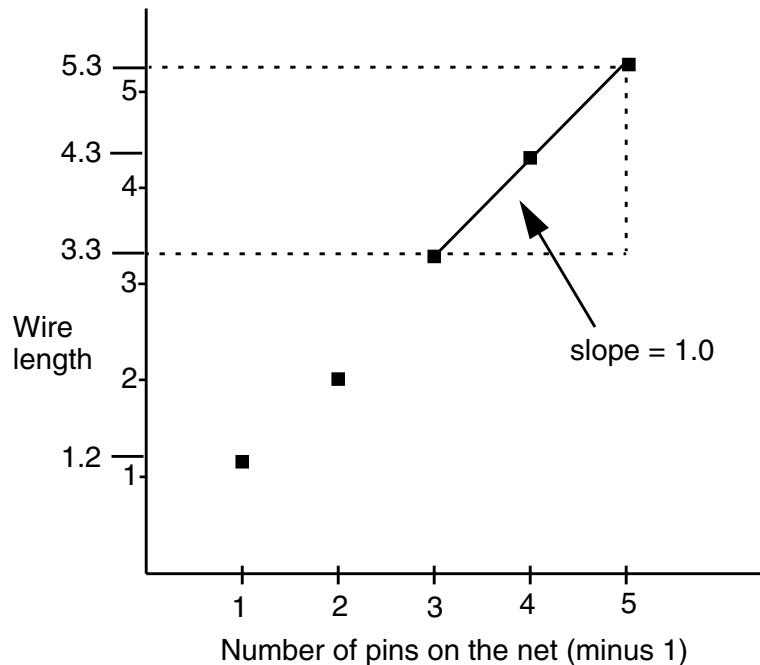
The `interconnect_delay` group specifies the lookup table template and the wire delay values.

Specify the `interconnect_delay` values as you would for any Library Compiler table.

To overwrite the default index values, specify the new index values before the `interconnect_delay` values, as shown here.

[Figure 12-1](#) 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 12-1 Wire Load Definition Graph*



Example 12-6 gives the library description for the model in Figure 12-1.

Note:

In Example 12-6 and Example 12-7, the name 90x90 is enclosed in quotation marks because it is a string that begins with a number.

*Example 12-6 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 12-7](#) shows how to estimate the wire delay by using a 3-D lookup table.

### *Example 12-7 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 12-6](#), 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, Library Compiler 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 12-8 shows the report for a wire load model containing new *fanout* and *length* information:

**Example 12-8 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

Library Compiler 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 12-9 shows a wire load model.

### *Example 12-9 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/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, Library Compiler 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

$$\begin{aligned}\text{Net area}[n] &= \text{Net area}[last \text{ fanout}] + ((fanout - last \text{ fanout}) \times \\ &\quad \text{slope} \times \text{area/unit\_length}) \\ \text{Net area}[6] &= \{(Net area[5]) + ((6-5) \times 0.311 \times 0.5)\} \\ \text{Net area}[5] &= \text{Area/unit\_length} \times \text{fanout\_length} \text{ (for fanout of 5)} \\ &= 0.5 \times 0.83 = 0.42\end{aligned}$$

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, Library Compiler uses interpolation. For a fanout of 3, there is no value in the model, so the calculation is

$$\begin{aligned}\text{Slope[between 2 and 5]} &= (\text{length}[5]-\text{length}[2]) / (5-2) \\ &= (0.83-0.63) / (5-2) = 0.0666 \\ \text{Net area}[2] &= 0.5 \times 0.63 = 0.32 \\ \text{Net area}[n] &= \text{Net area}[last fanout] + ((fanout-last fanout) \times \\ &\quad \text{slope} \times \text{area/unit\_length}) \\ \text{Net area}[3] &= \text{Net area}[2] + ((3-2) \times \text{slope[between 2 and 5]} \times \\ &\quad \text{area/unit\_length}) \\ \text{Net area}[3] &= 0.32 + (1 \times 0.0666 \times 0.5) = 0.3533\end{aligned}$$

The two fanouts closest to 3 are 2 and 5. Library Compiler 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 12-10](#) shows a source file containing a `wire_load_table` group and the report for the table:

#### *Example 12-10 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` library variable 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 12-11](#)) to allow for different amounts of wire load pessimism or in cases where the same library is used for different fabrication processes.

### *Example 12-11 Specification of Multiple wire\_load\_selection Groups*

```
wire_load_selection(really_pessimistic) {
    wire_load_from_area(min_areal,max_areal,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. Without multiple `wire_load_selection` groups, you must update your library with `update_lib` if you use a library compiled with a single `wire_load_selection` group and want to use a different group. When using multiple `wire_load_selection` groups, you must include the `default_wire_load_selection` library variable 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_area1,max_area1,Wire_Load_Name1);
    min_area2,max_area2,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 12-12](#). A design with an area outside the listed range is assigned the `wire_load` group of the next area range. In [Example 12-12](#), a design with an area of 120 gets the `10x10` `wire_load` group.

#### *Example 12-12 Wire Load Group Selection*

```
wire_load_selection(name1) {  
    wire_load_from_area(0,100,"05x05");  
    wire_load_from_area(150,200,"10x10");  
}
```

In [Example 12-13](#), a design with an area of 20 gets the `05x05` `wire_load` group.

#### *Example 12-13 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` attributes 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> 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` Design Compiler selects is reported by `report_lib`, 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

Design Compiler 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.

Library Compiler models the effects of process, temperature, and voltage variations on circuit timing, using the following:

- k-factors that apply to the entire library and are defined at the library level. Library Compiler 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 (see “[Scaling Factors for Individual Cells](#)” on page 12-59).

The scaling factors you define for your library depend on the timing delay model you use.

The following k-factors are specific to timing delay models:

- Intrinsic delay factors
- Slope sensitivity factors (CMOS generic delay model)
- Drive capability factors (CMOS generic delay model)

- Pin and wire capacitance factors
- Wire resistance factors
- Pin resistance factors (CMOS piecewise linear delay model)
- Intercept delay factors (CMOS piecewise linear delay model)
- Power scaling factors

Note:

Scaling factors have no effect on the scalable polynomial delay model.

- Timing constraint factors

Using these values, Design Compiler uniformly scales all timing numbers for timing analysis. See [Chapter 3, “Delay Models,”](#) for descriptions of the delay analysis equations Design Compiler uses.

## **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 value for any unspecified k-factor is 0.

## **Calculating Voltage Delay Factors**

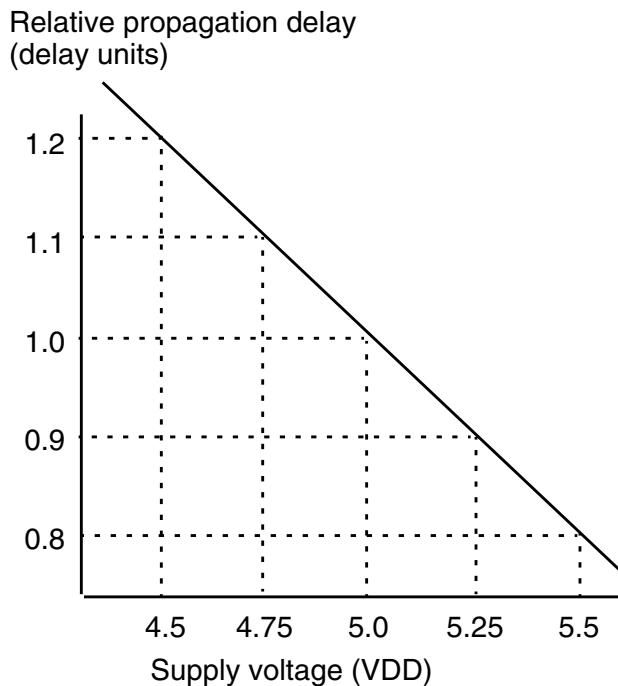
[Figure 12-2](#) illustrates the relationship between supply voltage and propagated delay for a typical technology library.

[Figure 12-2](#) 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  $k_{\text{volt}}$  delay factor is  $-0.4$ .

*Figure 12-2 Propagation Delay as a Function of Supply Voltage*



## Calculating Temperature Delay Factors

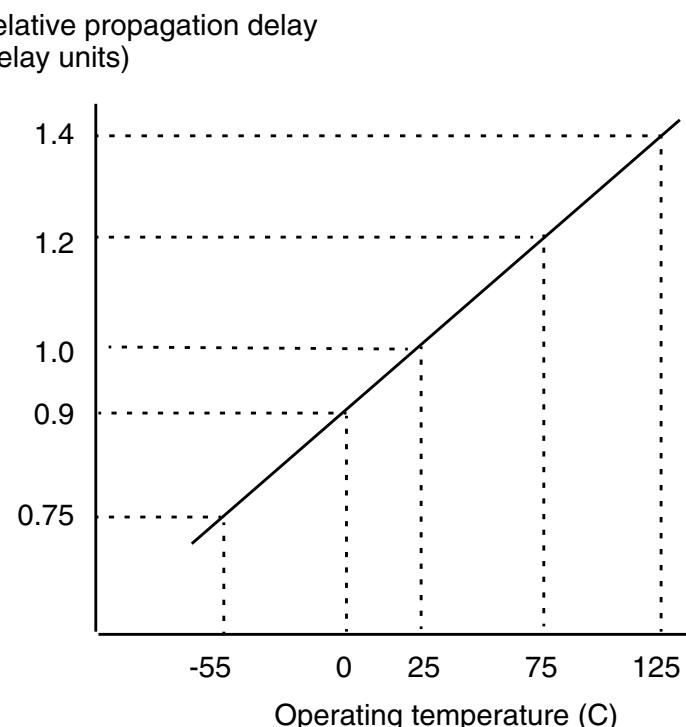
[Figure 12-3](#) illustrates the relationship between operating temperature and propagated delay for a typical technology library. The figure shows that the scaling factor for the delays in the timing equation increases as temperature increases.

Determine the k-factor values related to temperature by calculating the slope of the line as follows:

$$\begin{aligned}k_{\text{temp}} &= \text{delay scaling units} / \text{degree Centigrade} \\&= (1.4 - 0.9) / (125 - 0) \\&= 0.004\end{aligned}$$

If the slope of the line is 0.004 delay units per degree centigrade, the value of the  $k_{\text{temp}}$  delay factor is 0.004.

*Figure 12-3 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.

---

## Intrinsic Delay Factors

Intrinsic delay factors scale the intrinsic delay according to process, temperature, and voltage variations. Each attribute gives the multiplier for a certain portion of the intrinsic-rise delay or intrinsic-fall delay of all cells in the library. In the following syntax, *multiplier* is a floating-point number:

```
k_process_intrinsic_fall : multiplier ;
```

Scaling factor applied to the intrinsic fall delay of a timing arc to model process variation.

```
k_process_intrinsic_rise : multiplier ;
```

Scaling factor applied to the intrinsic rise delay of a timing arc to model process variation.

```
k_temp_intrinsic_fall : multiplier ;
```

Scaling factor applied to the intrinsic fall delay of a timing arc to model temperature variation.

```
k_temp_intrinsic_rise : multiplier ;
```

Scaling factor applied to the intrinsic rise delay of a timing arc to model temperature variation.

```
k_volt_intrinsic_fall : multiplier ;
```

Scaling factor applied to the intrinsic fall delay of a timing arc to model voltage variation.

```
k_volt_intrinsic_rise : multiplier ;
```

Scaling factor applied to the intrinsic rise delay of a timing arc to model voltage variation.

---

## Slope Sensitivity Factors

You can define slope sensitivity factors only in a library using a CMOS linear delay model. See Chapter 2, “Delay Models,” for details on CMOS linear delay analysis equations.

The slope sensitivity factors scale the delay slope according to process, temperature, and voltage variations. Each attribute gives the multiplier for a certain portion of the rise-delay slope or fall-delay slope of all cells in the library. In the following syntax, *multiplier* is a floating-point number:

`k_process_slope_fall : multiplier ;`

Scaling factor applied to timing arc fall slope sensitivity to model process variation.

`k_process_slope_rise : multiplier ;`

Scaling factor applied to timing arc rise slope sensitivity to model process variation.

`k_temp_slope_fall : multiplier ;`

Scaling factor applied to timing arc fall slope sensitivity to model temperature variation.

`k_temp_slope_rise : multiplier ;`

Scaling factor applied to timing arc rise slope sensitivity to model temperature variation.

`k_volt_slope_fall : multiplier ;`

Scaling factor applied to timing arc fall slope sensitivity to model voltage variation.

```
k_volt_slope_rise : multiplier ;
```

Scaling factor applied to timing arc rise slope sensitivity to model voltage variation.

---

## Drive Capability Factors

You can define drive capability factors only in a library using a CMOS linear delay model. See [Chapter 3, “Delay Models,”](#) for details on CMOS linear delay analysis equations.

The drive capability factors scale the drive capability of a pin according to process, temperature, and voltage variations. Each attribute gives the multiplier for a certain portion of a pin’s drive capability in rise-delay or fall-delay analysis. In the following syntax, *multiplier* is a floating-point number:

```
k_process_drive_current : multiplier ;
```

Scaling factor applied to timing arc `drive_current` to model process variation.

```
k_process_drive_fall : multiplier ;
```

Scaling factor applied to timing arc `fall_resistance` to model process variation.

```
k_process_drive_rise : multiplier ;
```

Scaling factor applied to timing arc `rise_resistance` to model process variation.

```
k_temp_drive_current : multiplier ;
```

Scaling factor applied to timing arc `drive_current` to model temperature variation.

```
k_temp_drive_fall : multiplier ;
```

Scaling factor applied to timing arc `fall_resistance` to model temperature variation.

```
k_temp_drive_rise : multiplier ;
```

Scaling factor applied to timing arc `rise_resistance` to model temperature variation.

```
k_volt_drive_current : multiplier ;
```

Scaling factor applied to timing arc `drive_current` to model voltage variation.

```
k_volt_drive_fall : multiplier ;
```

Scaling factor applied to timing arc `fall_resistance` to model voltage variation.

```
k_volt_drive_rise : multiplier ;
```

Scaling factor applied to timing arc `rise_resistance` to model voltage variation.

---

## 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.

---

## CMOS Wire Resistance Factors

Wire resistance factors scale wire resistance according to process, temperature, and voltage variations. Each attribute gives the multiplier for a certain portion of the wire resistance. In the following syntax, *multiplier* is a floating-point number:

```
k_process_wire_res : multiplier ;
```

Scaling factor applied to wire resistance to model process variation.

```
k_temp_wire_res : multiplier ;
```

Scaling factor applied to wire resistance to model temperature variation.

```
k_volt_wire_res : multiplier ;
```

Scaling factor applied to wire resistance to model voltage variation.

---

## Pin Resistance Factors

You can define pin resistance factors only in libraries that use a CMOS piecewise linear delay model. See [Chapter 3, “Delay Models,”](#) for details on CMOS piecewise linear delay analysis equations.

Pin-resistance factors scale resistance according to process, temperature, and voltage variations. Each attribute gives the multiplier for a certain portion of the pin resistance. In the following syntax, *multiplier* is a floating-point number:

```
k_process_rise_pin_resistance : multiplier ;
```

Scale factor applied to `rise_pin_resistance` to model process variation.

```
k_process_fall_pin_resistance : multiplier ;  
    Scale factor applied to fall_pin_resistance to model  
    process variation.  
  
k_temp_rise_pin_resistance : multiplier ;  
    Scale factor applied to rise_pin_resistance to model  
    temperature variation.  
  
k_temp_fall_pin_resistance : multiplier ;  
    Scale factor applied to fall_pin_resistance to model  
    temperature variation.  
  
k_volt_rise_pin_resistance : multiplier ;  
    Scale factor applied to rise_pin_resistance to model  
    voltage variation.  
  
k_volt_fall_pin_resistance : multiplier ;  
    Scale factor applied to fall_pin_resistance to model  
    voltage variation.
```

---

## Intercept Delay Factors

You can define intercept delay factors only in libraries that use a CMOS piecewise linear delay model. See [Chapter 3, “Delay Models,”](#) for details on CMOS piecewise linear delay analysis equations.

Intercept delay factors scale the delay intercept according to process, temperature, and voltage variations. These factors are used with slope- or intercept-type timing equations.

Each attribute gives the multiplier for a certain portion of the rise and fall intercepts of all cells in the library. In the following syntax, *multiplier* is a floating-point number:

```
k_process_rise_delay_intercept : multiplier ;
```

**Scale factor applied to `rise_delay_intercept` to model process variation.**

```
k_process_fall_delay_intercept : multiplier ;
```

**Scale factor applied to `fall_delay_intercept` to model process variation.**

```
k_temp_rise_delay_intercept : multiplier ;
```

**Scale factor applied to `rise_delay_intercept` to model temperature variation.**

```
k_temp_fall_delay_intercept : multiplier ;
```

**Scale factor applied to `fall_delay_intercept` to model temperature variation.**

```
k_voltage_rise_delay_intercept : multiplier ;
```

**Scale factor applied to `rise_delay_intercept` to model voltage variation.**

```
k_voltage_fall_delay_intercept : multiplier ;
```

**Scale factor applied to `fall_delay_intercept` to model voltage variation.**

---

## Power Scaling Factors

You can define power factors only in libraries that use a CMOS technology. Power scaling factors scale the power computation in Design Compiler by process, temperature, and voltage. The power scaling factors are listed below. In the following syntax, *multiplier* is a floating-point number:

`k_process_cell_leakage_power : multiplier ;`

Specifies environmental derating factors for the  
`cell_leakage_power` attribute.

`k_process_internal_power : multiplier ;`

Specifies environmental derating factors for the  
`internal_power` attribute.

`k_temp_cell_leakage_power : multiplier ;`

Specifies environmental derating factors for the  
`cell_leakage_power` attribute.

`k_temp_internal_power : multiplier ;`

Specifies environmental derating factors for the  
`internal_power` attribute.

`k_volt_cell_leakage_power : multiplier ;`

Specifies environmental derating factors for the  
`cell_leakage_power` attribute.

`k_volt_internal_power : multiplier ;`

Specifies environmental derating factors for the  
`internal_power` attribute.

---

## Timing Constraint Factors

Timing-constraint factors scale the following timing constraint values to account for the effects of changes in process, temperature, and voltage:

- Setup time
- Hold time
- No-change time
- Recovery time
- Removal time
- Minimum pulse width
- Minimum clock period
- Skew

An example of a timing constraint factor is `k_volt_hold_rise`, which is the scaling factor applied to hold constraints to model voltage variation. For setup, hold, and recovery time constraints, the `k`-factors containing the rise suffix are applied to the related `intrinsic_rise` value of the constraint timing group. Those with the fall suffix are applied to the related `intrinsic_fall` value.

For minimum pulse width constraints, the factors with the high suffix are applied to the `min_pulse_width_high` constraint. Those with the low suffix are applied to the `min_pulse_width_low` constraint.

In the following syntax examples, the scaling factor (multiplier) for temperature and voltage constraints is a floating-point number; for process constraints, the factor is a nonnegative floating-point number.

`k_process_hold_rise : multiplier ;`

Scaling factor applied to hold constraints to model process variation.

`k_process_hold_fall : multiplier ;`

Scaling factor applied to hold constraints to model process variation.

`k_process_removal_fall : multiplier ;`

Scaling factor applied to removal constraints to model process variation.

`k_process_removal_rise : multiplier ;`

Scaling factor applied to removal constraints to model process variation.

`k_temp_hold_rise : multiplier ;`

Scaling factor applied to hold constraints to model temperature variation.

`k_temp_hold_fall : multiplier ;`

Scaling factor applied to hold constraints to model temperature variation.

`k_temp_removal_fall : multiplier ;`

Scaling factor applied to removal constraints to model temperature variation.

`k_temp_removal_rise : multiplier ;`

Scaling factor applied to removal constraints to model temperature variation.

`k_volt_hold_rise : multiplier ;`

Scaling factor applied to hold constraints to model voltage variation.

`k_volt_hold_fall : multiplier ;`

Scaling factor applied to hold constraints to model voltage variation.

`k_volt_removal_fall : multiplier ;`

Scaling factor applied to removal constraints to model voltage variation.

`k_volt_removal_rise : multiplier ;`

Scaling factor applied to removal constraints to model voltage variation.

`k_process_setup_rise : multiplier ;`

Scaling factor applied to setup constraints to model process variation.

`k_process_setup_fall : multiplier ;`

Scaling factor applied to setup constraints to model process variation.

`k_temp_setup_rise : multiplier ;`

Scaling factor applied to setup constraints to model temperature variation.

`k_temp_setup_fall : multiplier ;`

Scaling factor applied to setup constraints to model temperature variation.

`k_volt_setup_rise : multiplier ;`

Scaling factor applied to setup constraints to model voltage variation.

`k_volt_setup_fall : multiplier ;`

Scaling factor applied to setup constraints to model voltage variation.

`k_process_nochange_rise : multiplier ;`

Scaling factor applied to no-change constraints to model process variation.

`k_process_nochange_fall : multiplier ;`

Scaling factor applied to no-change constraints to model process variation.

`k_temp_nochange_rise : multiplier ;`

Scaling factor applied to no-change constraints to model temperature variation.

`k_temp_nochange_fall : multiplier ;`

Scaling factor applied to no-change constraints to model temperature variation.

`k_volt_nochange_rise : multiplier ;`

Scaling factor applied to no-change constraints to model voltage variation.

`k_volt_nochange_fall : multiplier ;`

Scaling factor applied to no-change constraints to model voltage variation.

```
k_process_recovery_rise : multiplier ;
```

Scaling factor applied to recovery constraints to model process variation.

```
k_process_recovery_fall : multiplier ;
```

Scaling factor applied to recovery constraints to model process variation.

```
k_temp_recovery_rise : multiplier ;
```

Scaling factor applied to recovery constraints to model temperature variation.

```
k_temp_recovery_fall : multiplier ;
```

Scaling factor applied to recovery constraints to model temperature variation.

```
k_volt_recovery_rise : multiplier ;
```

Scaling factor applied to recovery constraints to model voltage variation.

```
k_volt_recovery_fall : multiplier ;
```

Scaling factor applied to recovery constraints to model voltage variation.

```
k_process_min_pulse_width_high : multiplier ;
```

Scaling factor applied to minimum pulse width constraints to model process variation.

```
k_process_min_pulse_width_low : multiplier ;
```

Scaling factor applied to minimum pulse width constraints to model process variation.

k\_temp\_min\_pulse\_width\_high : *multiplier* ;

Scaling factor applied to minimum pulse width constraints to model temperature variation.

k\_temp\_min\_pulse\_width\_low : *multiplier* ;

Scaling factor applied to minimum pulse width constraints to model temperature variation.

k\_volt\_min\_pulse\_width\_high : *multiplier* ;

Scaling factor applied to minimum pulse width constraints to model voltage variation.

k\_volt\_min\_pulse\_width\_low : *multiplier* ;

Scaling factor applied to minimum pulse width constraints to model voltage variation.

k\_process\_min\_period : *multiplier* ;

Scaling factor applied to minimum period constraints to model process variation.

k\_temp\_min\_period : *multiplier* ;

Scaling factor applied to minimum period constraints to model temperature variation.

k\_volt\_min\_period : *multiplier* ;

Scaling factor applied to minimum period constraints to model voltage variation.

k\_process\_skew\_fall : *multiplier* ;

Scaling factor applied to skew constraints to model process variation.

```
k_process_skew_rise : multiplier ;
```

Scaling factor applied to skew constraints to model process variation.

```
k_temp_skew_fall : multiplier ;
```

Scaling factor applied to skew constraints to model temperature variation.

```
k_temp_skew_rise : multiplier ;
```

Scaling factor applied to skew constraints to model temperature variation.

```
k_volt_skew_fall : multiplier ;
```

Scaling factor applied to skew constraints to model voltage variation.

```
k_volt_skew_rise : multiplier ;
```

Scaling factor applied to skew constraints to model voltage variation.

Table 12-1 shows how the constraints are calculated.

*Table 12-1 Timing Constraint Equations*

---

Setup Rise

$$\begin{aligned} \text{intrinsic\_rise } & (1 + \Delta\text{voltage k\_volt\_setup\_rise}) \\ & (1 + \Delta\text{temp k\_temp\_setup\_rise}) (1 + \Delta\text{process k\_process\_setup\_rise}) \end{aligned}$$

Setup Fall

$$\begin{aligned} \text{intrinsic\_fall } & (1 + \Delta\text{voltage k\_volt\_setup\_fall}) \\ & (1 + \Delta\text{temp k\_temp\_setup\_fall}) (1 + \Delta\text{process k\_process\_setup\_fall}) \end{aligned}$$

Hold Rise

$$\begin{aligned} \text{intrinsic\_rise } & (1 + \Delta\text{voltage k\_volt\_hold\_rise}) \\ & (1 + \Delta\text{temp k\_temp\_hold\_rise}) (1 + \Delta\text{process k\_process\_hold\_rise}) \end{aligned}$$

Hold Fall

$$\begin{aligned} \text{intrinsic\_fall } & (1 + \Delta\text{voltage k\_volt\_hold\_fall}) \\ & (1 + \Delta\text{temp k\_temp\_hold\_fall}) (1 + \Delta\text{process k\_process\_hold\_fall}) \end{aligned}$$

Recovery Rise

$$\begin{aligned} \text{intrinsic\_rise } & (1 + \Delta\text{voltage k\_volt\_recovery\_rise}) \\ & (1 + \Delta\text{temp k\_temp\_recovery\_rise}) (1 + \Delta\text{process k\_process\_recovery\_rise}) \end{aligned}$$

*Table 12-1 Timing Constraint Equations (Continued)*

---

Recovery Fall

intrinsic\_fall (1 + Δvoltage k\_volt\_recovery\_fall)  
(1 + Δtemp k\_temp\_recovery\_fall) (1 + Δprocess k\_process\_recovery\_fall)

Minimum Pulse Width High

min\_pulse\_width\_high (1 + Δvoltage k\_volt\_min\_pulse\_width\_high)  
(1 + Δtemp k\_temp\_min\_pulse\_width\_high)  
(1 + Δprocess k\_process\_min\_pulse\_width\_high)

Minimum Pulse Width Low

min\_pulse\_width\_low (1 + Δvoltage k\_volt\_min\_pulse\_width\_low)  
(1 + Δtemp k\_temp\_min\_pulse\_width\_low)  
(1 + Δprocess k\_process\_min\_pulse\_width\_low)

Minimum Clock Period

min\_period (1 + Δvoltage k\_volt\_min\_period)  
(1 + Δtemp k\_temp\_min\_period) (1 + Δprocess k\_process\_min\_period)

---

---

## Delay Scaling Factors Example

[Example 12-14](#) shows delay scaling factors for a CMOS generic delay model.

### *Example 12-14 Setting k-Factors*

```
library (example) {
    ...
    k_process_drive_fall      : 1.0;
    k_process_drive_rise      : 1.0;
    k_process_hold_rise       : 1.0;
    k_process_hold_fall       : 1.0;
    k_process_intrinsic_fall  : 1.0;
    k_process_intrinsic_rise  : 1.0;
    k_process_pin_cap         : 0.0;
    k_process_slope_fall      : 1.0;
    k_process_slope_rise      : 1.0;
    k_process_wire_cap        : 0.0;
    k_process_wire_res        : 1.0;
    k_temp_drive_fall         : 0.004;
    k_temp_drive_rise         : 0.004;
    k_temp_hold_rise          : .0037;
    k_temp_hold_fall          : .0037;
    k_temp_intrinsic_fall     : 0.004;
    k_temp_intrinsic_rise     : 0.004;
    k_temp_pin_cap            : 0.0;
    k_temp_slope_fall         : 0.0;
    k_temp_slope_rise         : 0.0;
    k_temp_wire_cap           : 0.0;
    k_temp_wire_res           : 0.0;
    k_volt_drive_fall         : -0.4;
    k_volt_drive_rise         : -0.4;
    k_volt_hold_rise          : -0.26;
    k_volt_hold_fall          : -0.26;
    k_volt_intrinsic_fall     : -0.4;
    k_volt_intrinsic_rise     : -0.4;
    k_volt_pin_cap            : 0.0;
    k_volt_slope_fall         : 0.0;
    k_volt_slope_rise         : 0.0;
    k_volt_wire_cap           : 0.0;
    k_volt_wire_res           : 0.0;
    ...
}
```

```
}
```

In [Example 12-14](#), only the intrinsic-rise, intrinsic-fall, rise-resistance, and fall-resistance delays are affected by a change in operating voltage. Setting the other voltage factors to 0.0 negates changes to them.

---

## Scaling Factors for Individual Cells

The k-factors you define for a library as a whole do not produce accurate timing for certain cells, because not all cells in the same library scale uniformly: Pads scale differently from core cells, the timing of voltage-level pads varies with temperature, and some cells are designed to produce a constant delay and do not scale at all.

Other cells do not scale in a linear manner for process, voltage, or temperature. You can define a special set of scaling factors in a library-level group called `scaling_factors` and apply the k-factors in this group to selected cells by using the `scaling_factors` attribute.

You can apply library-level k-factors to the majority of cells in your library and use this construct to provide additional accurate scaling factors for special cells. [Example 12-15](#) uses the special scaling factors.

### *Example 12-15 Individual Scaling Factors*

```
library (example) {
    k_volt_intrinsic_rise : 0.987 ;
    ...
    scaling_factors("IO_PAD_SCALING") {
        k_volt_intrinsic_rise : 0.846 ;
        ...
    }
    cell (INPAD_WITH_HYSTERESIS) {
```

```
    area : 0 ;
    scaling_factors : IO_PAD_SCALING ;
    ...
}
...
}
```

[Example 12-15](#) defines a scaling factor group called `IO_PAD_SCALING` that contains k-factors that are different from the library-level k-factors.

You can use any k-factors in a `scaling_factors` group. The `scaling_factors` attribute in the `INPAD_WITH_HYSTERESIS` cell is set to `IO_PAD_SCALING`, so all k-factors set in the `IO_PAD_SCALING` group are applied to the cell.

By default, all cells without a `scaling_factors` attribute continue to use the library-level k-factors.

You can model cells that do not scale at all, by creating a `scaling_factors` group in which all the k-factor values are set to 0.0.

---

## Scaling Factors Associated With the Nonlinear Delay Model

As with the other delay models, the CMOS nonlinear delay model scaling factors scale the delay based on the variation in process, temperature, and voltage. The following scaling factors are specific 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

The process, temperature, and voltage scaling factors for other CMOS delay models are not valid with the CMOS nonlinear timing model. Library Compiler checks these attributes as invalid and issues an error message. Only the scaling factors listed above are valid with the nonlinear timing model.

Design Compiler uses the same factors described for the CMOS generic delay model to scale pin and wire capacitance and wire resistance for the nonlinear delay model.

Define the scaling factors for setup, hold, recovery, removal, and skew in the nonlinear delay model as you do for the other delay models. For the nonlinear delay model, however, they apply to the values given in the associated constraint table.

For example, the timing constraint equation for setup rise is

```
rise_constraint * (1 + delta_voltage * k_volt_setup_rise)  
* (1 + delta_temp * k_temp_setup_rise)  
* (1 + delta_process * k_process_setup_rise)
```

where `rise_constraint` is a value in the `rise_constraint` table of the timing arc.

These are the scaling factors for the `setup_rising` timing constraint:

- `k_volt_setup_rise`
- `k_temp_setup_rise`
- `k_process_setup_rise`

The `scaling_factors` group is not affected by the change from linear to nonlinear delay model. The scaling factors for setup rise are valid in the `scaling_factors` group.

## Specifying Nonlinear Delay Tables

These are two methods for specifying nonlinear delay tables.

## **Method 1**

Use rise/fall propagation with rise/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.



# 13

## Verifying CMOS Libraries

---

Design Compiler uses information from technology libraries to drive its optimization strategies and to check that solutions adhere to the designer's specifications. The results of optimization are only as accurate as the technology library used.

To verify and validate CMOS technology and symbol libraries, you must understand the information in the following sections:

- Library Verification
- Checking Library Consistency
- Preparing a Test Circuit
- Verifying Functionality
- Verifying Timing Parameters

---

## Library Verification

To ensure optimal results, you must make sure technology libraries have the following properties:

- Accuracy: The timing values and functional descriptions must be correct and consistent.
- Completeness: Describe all cells available from the ASIC vendor in the technology library. Use a complete delay calculation model.
- Consistency: All cells in the technology library must have an equivalent graphic representation in the symbol library.

If you already have the information needed to create a library file in another database, you can develop tools that automatically create a technology library from the information in your simulation libraries. Synopsys encourages the development of such tools. An automated system for creating technology libraries ensures

- Accuracy, by eliminating manual errors
- Completeness, by deriving information from the core database
- Consistent results, by using a single source of data

You can test a synthesis library by using the Synopsys VHDL Simulator. Library Compiler automatically generates the simulation models, test benches, test vectors, and a shell script. See *Library Compiler Reference Manual: VHDL Libraries* for additional information.

To verify a library, you need

- The new library you want to verify

- A known, *golden* library
- A specially designed test circuit
- The simulator and timing analyzer from your development environment

Both the library to be verified and the known golden library are used in synthesis and optimization under very controlled circumstances. By comparing the results, you can determine whether the new library is accurate.

The procedures work well on combinational logic and simple cells. Libraries that contain complex, sequentially modeled cells can be tested, but the results may not be robust. For additional support, contact the Synopsys hotline at (800) 245-8005.

---

## Checking Library Consistency

The `compare_lib` command compares a technology library and a symbol library for consistency. These libraries must already be loaded in Design Compiler. Use the `list` command to display the libraries that are resident in Design Compiler.

This is the syntax for the `compare_lib` command:

`compare_lib filename1 filename2`

*filename1, filename2*

The names of the libraries to be compared. One must be a technology library and the other a symbol library. You can write these names in any order.

The `compare_lib` command performs two checks. First, it verifies that each cell in the technology library has a corresponding symbol definition in the symbol library. Second, it checks that the pin names of each cell in the technology library match the pin names defined for the cell's corresponding symbol.

In the following example, the `list` command verifies that the libraries you want to compare are already resident. Then the `compare_lib` command compares the `cmos.db` technology library with the `cmos.sdb` symbol library.

```
dc_shell> list -libraries  
  
Library      File      Path  
-----      ----      ----  
cmos        cmos.db    /home/user/libraries  
cmos        cmos.sdb   /home/user/libraries  
  
dc_shell> compare_lib cmos.db cmos.sdb
```

Design Compiler reports discrepancies between the libraries.

---

## Preparing a Test Circuit

You must prepare a test circuit design for each technology library you want to test. Most of the test and evaluation procedures described in this chapter assume that you are using a test circuit design with a specific topology. After you create this design, you can use it in each of the verification procedures.

The test design contains at least one of each cell in the technology library being tested. Connect each cell's input in a design to separate input pins in the module, and connect its output pins directly to a

separate module output. Each clock pin should be driven from a separate pin. You need multiple clock pins if cells in the library require more than a single clock.

**Example 13-1** shows a sample VHDL format netlist created for a library that contains the cells INV, NAND2, OR2, MUX21, AOI22, and DFF.

*Example 13-1 Sample VHDL Format Netlist*

```
entity TEST_CIRCUIT is
port( I1, I2, I3, I4, CLK : in BIT; O1, O2, O3, O4, O5, O6, O7 : out BIT);

end TEST_CIRCUIT;

architecture STRUCTURAL_VIEW of TEST_CIRCUIT is

component DFF
    port( D, CP : in BIT; Q, QN : out BIT);
end component;

component AOI22
    port( A, B, C, D : in BIT; Z : out BIT);
end component;

component NAND2
    port( A, B : in BIT; Z : out BIT);
end component;

component MUX21
    port( A, B, S : in BIT; Z : out BIT);
end component;

component OR2
    port( A, B : in BIT; Z : out BIT);
end component;
component INV
    port( A : in BIT; Z : out BIT);
end component;

begin
    O1_label : INV port map( A => I1, Z => O1);
    O2_label : NAND2 port map( A => I1, B => I2, Z => O2);
    O3_label : OR2 port map( A => I1, B => I2, Z => O3);
    O4_label : MUX21 port map( A => I1, B => I2, S => I3, Z => O4);
```

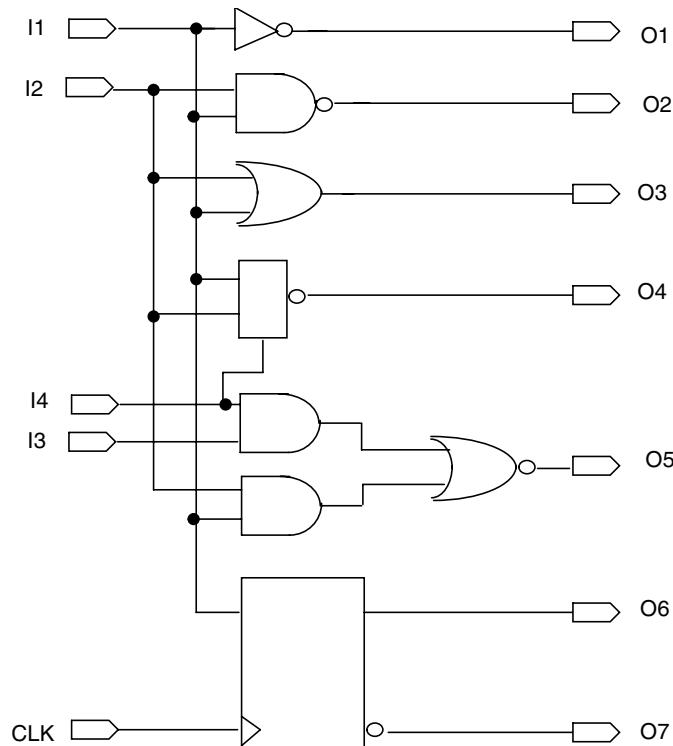
```

O5_label : AOI22 port map( A => I1, B => I2, C => I3, D => I4, Z => O5);
U6 : DFF port map( D => I1, CP => CLK, Q => O6, QN => O7);
end STRUCTURAL_VIEW;

```

[Figure 13-1](#) shows the six-gate schematic for [Example 13-1](#).

*Figure 13-1 Test Circuit Netlist Network*



Sharing the input pins makes creating input test vectors easier. In this example, it is practical for a single netlist to contain all cells of the technology library being verified. When you are testing a large library, however, you might want to use more than one netlist.

---

## Verifying Functionality

To optimize logic networks properly, Design Compiler requires a database that describes the Boolean function of the elements in the target technology library. If these Boolean functions are incorrectly described in the library, Design Compiler can create invalid circuits. To prevent this, you must ensure that the Boolean function attributes given in the technology library match the functions described in the corresponding simulation library.

---

### Preparing for Function Verification

To follow the function verification procedure, use an accurate simulator and a simulation library that is known to be functionally correct.

To prepare for testing, you need the following items:

- The Design Compiler software
- A technology library to test
- The test circuit netlist described in the previous section
- The simulator from the resident CAE environment
- The simulation library corresponding to the technology library you want to test
- The simulation test vectors
- The simulation output comparison program (optional)
- The test-mapping library created specifically for function verification

When you are testing the function descriptions in the technology library, you need to include only cells with `function` attributes in the test circuit netlist. Avoid including cells without function statements, unless you have simulation models. When the simulation comparison is performed, unmapped gates are compared against copies of themselves.

To verify the functionality of a technology library,

1. Create the test-mapping library
2. Map the test circuit netlist onto the test-mapping library

Note:

At this point in the process, your design is described only in terms of the cells from the test-mapping library. The design remains functionally identical to the design described in the test circuit netlist, but it is described solely in terms of the gate types in the test-mapping library.

3. Simulate both the original test circuit netlist and the mapped design
4. Compare the results of both simulations

---

## **Creating the Test-Mapping Library**

The first new item to prepare is a test-mapping library. Use this as the target library for mapping the test circuit netlist you created. This library is as simple and small as possible, containing only two cells: an inverter and a 2-input NOR gate.

These two gates constitute the absolute minimal library used for mapping and optimization with Design Compiler. By using a simple library, you can ensure that the results of the mapping reflect the validity of the technology library being tested. [Example 13-2](#) is a sample test-mapping library.

*Example 13-2 Sample Test-Mapping Library*

```
library(test_map_lib) {
    /* 2-input NOR gate */
    cell(NOR_2) {
        pin(A) {
            direction : input ;
        }
        pin(B) {
            direction : input ;
        }
        pin(X) {
            direction : output ;
            function : "(A+B)' " ;
            timing() {
                related_pin : "A B" ;
            }
        }
    }
    /* INVERTER gate */
    cell(INV) {
        pin(A) {
            direction : input ;
        }
        pin(X) {
            direction : output ;
            function : "A' " ;
            timing() {
                related_pin : "A" ;
            }
        }
    }
}
```

Include only the minimum information required for each cell in the test library file. Verify this file by visual inspection.

Note:

Design Compiler reads in a technology library containing any cell. Any library can be used to generate a VHDL library. The technology library must contain at least a 2-input NOR gate and an inverter. The minimum, however, is not sufficient for all designs. The minimum sufficient technology library to handle a design consists of the set of cells described in “[Verifying Technology and VHDL Libraries Together](#)” on page 13-13.

The following commands prepare the library for the function verification test and compile and save it:

```
lc_shell> read_lib test_map.lib.lib
lc_shell> write_lib test_map.lib
```

The result is a technology library named test\_map.lib.db.

---

## Mapping the Test Circuit Netlist

The next step is to map the test circuit netlist of library cells onto the two cells described in test\_map.lib.db. To do this, follow these steps:

1. Set the following variables in dc\_shell:

```
dc_shell> link_library = {your_technology_library.db,test_map.lib.db }
dc_shell> target_library = test_map.lib.db
```

2. Read in the test circuit netlist. This example uses the VHDL format, but you can use any supported netlist format.
3. Map the netlist onto the test library with the following commands:

```
dc_shell> read -f vhdl TEST_CIRCUIT.NET
dc_shell> translate
```

Note:

At this point in the process, you have a design containing only the cells from the test-mapping library. The design is functionally identical to the design described in the test circuit netlist, and the design is described solely in terms of the two gate types in the test-mapping library.

#### 4. Export the design to the simulation environment:

```
dc_shell> write -f vhdl TEST_CIRCUIT -o TEST_MAPPED.NET
```

This command saves the mapped design in a file called TEST\_MAPPED.NET.

---

### **Creating the Simulation Test Vectors**

You now have two netlists:

- TEST\_CIRCUIT.NET, which uses the technology library you are verifying
- TEST\_MAPPED.NET, which is the result of mapping your test circuit netlist onto the two cells that you know are accurately modeled

The next step is to simulate both of the netlists and compare the reported functionality. To do this,

1. Create a set of simulation input vectors to use in both simulations.
2. Generate the pattern file by using the language of your simulation environment. You can simulate all possible input patterns in cases in which the number of input pins is small.

3. Separately test those cells having more than 20 input pins. Use the same methodology, coupled with a targeted function test-vector set.
- 

## **Simulating the Mapped and Original Test Netlists**

Run the simulator on both netlists. For convenience, you can create a super netlist that contains both the mapped and the original netlists. On this netlist, each pair of corresponding output pins connected with an XOR gate identifies all situations where two corresponding output pins are not identical.

The process is taken one step further by connecting the output pins of the exclusive-OR gates with OR gates. With this connection, there is a single super module output to monitor. The library is functionally correct if the output of this OR gate is not high after all events from each input vector have settled.

---

## **Identifying Function Description Errors**

After you have the results of both simulations, compare the results and search for discrepancies. Many vendors provide a tool for comparing the results of two simulation output pins. If you have this capability, run that tool.

If you do not have a simulation-output comparison program, you must perform the discrepancy check manually. Every discrepancy in the simulations is a functional inconsistency between your simulation library and your technology library.

When no discrepancies exist, the technology and simulation libraries are functionally consistent.

Combinational cells with the `dont_touch` attribute are not verified by this procedure; like black boxes, these cells are not mapped.

---

## Verifying Technology and VHDL Libraries Together

You can verify the functionality of your technology library and the corresponding VHDL library at the same time. The steps are the same as those listed in “[Preparing for Function Verification](#)” on [page 13-7](#). Variations on this basic procedure are needed to test the two libraries concurrently.

### Creating the Test-Mapping Library

You need the following set of cells in the test-mapping library to support all technology libraries for this procedure:

- An inverter.
- A 2-input NOR gate.
- A three-state buffer, if used in the library.
- A D flip-flop with preset, clear, and two complementary output pins—when both control signals are active, the output pins are both 0.
- A D flip-flop with preset, clear, and two complementary output pins—when both control signals are active, the output pins are both 1.

When you create the DB image of the test-mapping library, create a VHDL library as well and analyze it:

```
lc_shell> read_lib test_map_lib.lib
lc_shell> write_lib test_map_lib
lc_shell> write_lib -f vhdl test_map_lib
```

## Mapping the Test Circuit Netlist

To map the test circuit netlist onto your test map library, use these commands:

```
dc_shell> vhdlout_write_components = FALSE
dc_shell> vhdlout_bit_type = "MVL7";
dc_shell> vhdlout_bit_vector_type = "MVL7_VECTOR";
dc_shell> vhdlout_use_packages = {synopsys.types.all
    technology.COMPONENTS.all}
dc_shell> link_path={technology.db, test_map_lib.db}
dc_shell> target_library={test_map_lib.db}
dc_shell> symbol_library={generic.sdb}
dc_shell> read -f tdl test_circuit.tdl
dc_shell> write -f vhdl test_circuit -output net_lib.vhd
dc_shell> max_area 0
dc_shell> set_structure false -timing false
dc_shell> set_flatten -phase true -effort low
dc_shell> compile
dc_shell> write -f vhdl test_circuit -output
    net_test_map.vhd
```

## Completing the Verification

After you create the simulation test vectors (see “[Creating the Simulation Test Vectors](#)” on page 13-11), run VHDL simulation on both net\_lib.vhd and net\_test\_map.vhd and compare the results. Simulations differ in unknown behavior; see “[Ambiguity Resolution](#)” in the *Library Compiler Reference Manual: VHDL Libraries* for additional information.

---

## Verifying Timing Parameters

You need to verify timing parameters for two reasons:

1. When you work with several different tools in a single design environment, you must ensure that all the tools use consistent technology information.
2. Design Compiler must have accurate timing information to use when it synthesizes and optimizes your designs. Inaccurate timing information can cause Design Compiler to create less-than-optimal designs. Bad timing information can even cause Design Compiler to create new circuits that represent a degradation in performance from the original.

Both of these requirements are met by a determination of how closely the results from the Design Compiler timing calculations track the results generated by the timing simulator.

---

## Testing Parameters

The following sections describe the method for verifying the timing parameters in a technology library, describing important parameters to test, and devising tests for them:

- General Testing Methodology
- Debugging With the `report_delay_calculation` Command
- Testing Basic Cell-Timing Parameters
- Testing the Interconnect Models
- Testing Environmental Scaling

To verify that the parameter value entry is correct, use the `report_lib -timing` command, described in [Chapter 6, “Advanced Composite Current Source Modeling.”](#)

The divide-and-conquer approach to parameter testing is important in determining sources of discrepancies and in differentiating algorithmic differences in the tools from library parameter errors.

To verify the timing parameters, you need

- The test circuit netlist
- The Design Compiler software
- A timing simulator or analyzer

This is the procedure:

1. Run a Design Compiler timing report on the test circuit netlist.
2. Run a simulation timing analyzer on the test circuit netlist.
3. Compare the results.

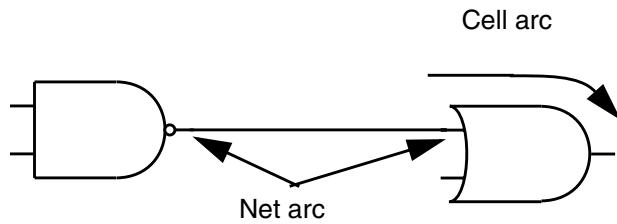
---

## General Testing Methodology

This section describes a general methodology for testing timing arcs associated with nets and cells with the `report_delay_calculation` command.

Use the `report_delay_calculation` command to display detailed timing calculation information about both cell arcs and net arcs, as shown in [Figure 13-2](#). This information is especially helpful for debugging or verifying timing data.

Figure 13-2 Net and Cell Arcs



Note:

If you want the `report_delay_calculation` command to display confidential library data, you must load the library in source format. You can do this with the `read_lib` command.

This is the syntax of the `report_delay_calculation` command:

```
report_delay_calculation -from pin_name -to pin_name
```

`-from pin_name`

Defines the starting point of a timing arc within your design.

`-to pin_name`

Defines the ending point of a timing arc within your design.

For a *cell* timing arc, use the input and output pins of a common leaf cell that has a timing arc defined between them in the library.

For a *net* timing arc, use a driver and a load on a common net. Port names are allowed in place of a pin name for net arcs.

If there is a cell timing arc between pins, the details of the *cell* arc delay calculation will be displayed in the timing arc report. If there are pins that are part of the same net, the details of the *net* arc delay calculation will be displayed in the timing arc report.

The following conditions produce an error with no delay information displayed:

- A technology library was not loaded in source format.
- There are no cell or net delay arcs between the specified pins.
- You specified an undefined pin.
- The specified pin is associated with a nonleaf cell.
- You specified more than one pin for the `-from` or `-to` option.
- You specified an arc type between pins that is not supported.

The format of the timing report output varies, depending on the type of delay model and the interconnect delay tree type (for net delay arcs). If delay values have been back-annotated for a specified arc, the annotated delay, rather than calculated delay, is given.

[Example 13-3](#) shows one type of timing arc report that is generated for the output of a cell arc in a generic CMOS library. The timing reports for different types of libraries, such as nonlinear, look similar.

### *Example 13-3 Cell Arc Output Report*

```

From : U1/A
To : U1/X
arc type: cell
arc sense: inverting
Input net transition times: Dt_rise = 0, Dt_fall = 0

Rise Delay computation:
rise_intrinsic          0 +
rise_slope * Dt_fall      0 * 0 +
rise_resistance * (pin_cap + wire_cap) / driver_count
0.2 * (0 + 0.53) / 1
rise_transition_delay :    0.106
-----
Total                  0.106

Fall Delay computation:
fall_intrinsic          0 +
fall_slope * Dt_rise      0 * 0 +
fall_resistance * (pin_cap + wire_cap) / driver_count

```

```

0.15 * (0 + 0.53) / 1
fall_transition_delay :          0.0795
-----
Total                      0.0795

```

---

## Debugging With the report\_delay\_calculation Command

You can use the `report_delay_calculation` command to help debug delay calculations along a critical path. To do this, first locate a critical path in a design with the `report_timing -path full -input_pins` command to get a listing similar to this one:

Point	Incr	Path
input external delay	0.00	0.00 f
i2 (in)	0.00	0.00 f
cell1/i2 (lower1)	0.00	0.00 f
cell1/C/B (AN2)	0.00	0.00 f
cell1/C/Z (AN2)	0.82	0.82 f
cell1/o1 (lower1)	0.00	0.82 f
cell2/i1 (lower2)	0.00	0.82 f
cell2/C/A (IV)	0.00	0.82 f
cell2/C/Z (IV)	0.38	1.20 r
cell2/o1 (lower2)	0.00	1.20 r
o1 (out)	0.00	1.20 r
data arrival time		1.20

The `-input_pins` option causes the input pins to be listed in the path, in addition to the output pins typically seen. Next, you can use the `report_delay_calculation` command to print the details of a cell delay arc, by giving it the input and output pin of a leaf cell along the path. This is an example:

```
report_delay_calculation -from cell1/C/B -to cell1/C/Z
```

To print the details of a net delay arc, use a driver and a load pin on the same net along the path. The pins must be associated with leaf cells. This is an example that is valid, because both cells are leaf cells:

```
report_delay_calculation -from cell1/C/Z -to cell2/C/A
```

The following command is invalid, because there is no net delay arc associated with the from pin, because it is not on a leaf cell:

```
report_delay_calculation -from cell2/i1 -to cell2/C/A
```

Operating conditions and wire loads are considered when you generate a timing arc report, but timing ranges are not. This is because timing ranges typically apply to an entire path, as opposed to a single timing arc.

---

## Testing Basic Cell-Timing Parameters

Test all timing parameters and related information used to describe cells in a technology library. These parameters include

- Intrinsic delay
- Resistance
- Capacitance
- Input slope
- Setup
- Hold
- Rising and falling edge
- Area

- Ranges for piecewise linear model

## Intrinsic Delays

To test intrinsic delays that are unloaded-gate delay values, run a Design Compiler timing report on the test design described earlier. Use the following commands:

```
dc_shell> read -f vhdl TEST_CIRCUIT.NET
dc_shell> report_timing > test.intrinsic
```

Then run a timing simulation or analysis on the test design.

Compare the results of these tests. If the output capabilities of the simulator allow it, write a small program to compare the two output values. The development of such a program is strongly encouraged.

If you have a `default_wire_load` or  
`default_operating_conditions` attribute defined in your  
library, use the `set_wire_load` or  
`set_operating_conditions` command to set the wire load and  
operating conditions to null, as in the following example:

```
dc_shell> set_wire_load ""
dc_shell> set_operating_conditions ""
```

## Resistance (Generic Delay Model)

Test the resistance parameters described in the technology library for each cell under at least two loading conditions. These are the resistance parameters used in the standard timing delay model:

- `rise_resistance`
- `fall_resistance`

Using the same design as for intrinsic delay, vary the load at each output pin with the Design Compiler `set_load` command for each output pin, as illustrated in [Example 13-4](#).

*Example 13-4 Testing Generic-Delay Resistance Parameters*

```
dc_shell> read -f vhdl TEST_CIRCUIT.NET
dc_shell> set_load 2 all_outputs()
dc_shell> report_timing > test.resis_load_2
dc_shell> set_load 4 all_outputs()
dc_shell> report_timing > test.resis_load_4
```

Run a simulation by using the same two loading conditions if your simulator lets you set loading at output pins. If the simulator cannot set alternative load values at module output pins, you must generate this effect by adding more load gates to each module output.

To add more load gates to each module output:

1. Add one inverter.
2. Add two inverters to each output pin by connecting the input pins and connecting the output pins.

## **Resistance (Piecewise Linear Delay Model)**

Test the resistance parameters for each cell under at least two loading conditions. These are the resistance parameters used in the piecewise linear timing delay model:

- `rise_pin_resistance`
- `fall_pin_resistance`
- `rise_delay_intercept`
- `fall_delay_intercept`

Use the test design described earlier in this chapter. Vary the wire length and load at each output pin. To vary the wire length,

1. Connect the output of the test circuit to a load circuit. Use a small gate such as an inverter for the load circuit.
2. Make sure you check each range defined by the `piece_define` statement. For example, pieces 0, 1, and 2 are tested in the example below. The wire load model defines the wire length added for each extra loading gate. This is an example:

```
dc_shell> read -f vhdl TEST_CIRCUIT.NET
dc_shell> set_wire_load default
dc_shell> report_timing > test.resis_load_piece_0
dc_shell> read -f vhdl TEST_CIRCUIT_piece_1.NET
dc_shell> set_wire_load default
dc_shell> report_timing > test.resis_load_piece_1
dc_shell> read -f vhdl TEST_CIRCUIT_piece_2.NET
dc_shell> set_wire_load default
dc_shell> report_timing > test.resis_load_piece_2
```

3. Run a simulation on each of the different netlists.
4. Compare the results of the simulation with the timing reports generated by Design Compiler.

## Capacitance

Testing capacitance output pin values, which are often 0 in technology libraries, is implicit in the verification of both intrinsic delay and resistance. You need to devise additional tests only for input pin capacitance values. Use the same design as for intrinsic delay, but vary the drive value at each input pin with the Design Compiler `set_drive` command.

### Example

```
dc_shell> read -f vhdl TEST_CIRCUIT.NET
dc_shell> set_drive 2 all_inputs()
```

```
dc_shell> report_timing > test.incap_drive_2
dc_shell> set_drive 4 all_inputs()
dc_shell> report_timing > test.incap_drive_4
```

If your simulator allows you to vary the effective drive at design input pins, input pin capacitance validation is not difficult. If the simulator cannot vary the effective drive at the input pins, you must add inverters or buffers at every input pin to generate a drive at the pin.

The pin's delay value, in turn, depends on the input pin capacitance under test. For this test, you need only a single simulation with a nonzero drive value at the input pins.

## Input Slope

Use the same basic design used for function verification to begin testing each gate's sensitivity to changes in rise time and input fall time. The test for the input sensitivity to step functions is implicit in the tests for intrinsic delay and resistance parameters.

To test nonzero rise and fall times, use the Design Compiler `set_load` and `set_drive -rise` or `set_drive -fall` commands to change the input rise and fall time, as in the following example:

## Example

```
dc_shell> read -f vhdl TEST_CIRCUIT.NET
dc_shell> set_load 2 all_inputs()
dc_shell> set_drive 2 all_inputs()
dc_shell> report_timing > test.slope_rc_4
dc_shell> set_load 2 all_inputs()
dc_shell> set_drive 4 all_inputs()
dc_shell> report_timing > test.slope_rc_8
```

## Setup

Setup times are treated as constraint equation parameters by Design Compiler. To check the parameters of constraint equations, devise dynamic tests that check both the violating condition and the complying condition at the boundary of the violation.

Implement the test in Design Compiler by using a combination of the `set_input_delay` command and the `max_period` constraint to set up the dynamic conditions for a setup check.

For example, assume you have a rising-edge-triggered D flip-flop with a setup time requirement of 4. For simplicity, assume that only integer time values are assigned. To test the boundary violation condition, use the following commands:

```
dc_shell> set_input_delay -fall 1 d
Performing set_input_delay on port 'd'.
1
dc_shell> create_clock 3 CLK
Performing create_clock on clock 'CLK'
1
dc_shell> report_constraint -verbose
Information: Updating design information... (UID-85)

*****
Report : constraint
    -verbose
Design : DESIGN
Version : v3.0
Date   : Tues Jan 14 14:05:20 2002
*****
1. max_period 3.000000 { CLK }

Worst:          q_reg/D
Max Data Path      1.00
+ Library Setup     4.00
- Min Active Clock   0.00
- (Clock Period *
    Period Multiplier) 1.00
+ Minus Clock Skew   0.00
```

```

-----
Delta Cost           2.00 (VIOLATED)

The minimum clock period is 5.00 for signal 'CLK' (on 'q_reg/D') .
1

```

In the simulator, you can use input vectors that generate the `set_input_delay` and `max_period` signal behavior for the setup check. If setup and hold checking is provided by the simulator, you can use the output listing to verify that both systems have found that the signal complies with, or violates, the timing requirements of the cell. In the simulator, the vector specification might look something like this:

```

time 0 : D_PORT = 0, CLOCK_PORT = 0; /*init signals */
time 100: D_PORT = 1;      /*apply data signal change */
time 103: CLOCK_PORT = 1; /*apply clk at period boundary of
3*/

```

The simulator output will also show a violation.

Validation of the boundary condition for compliance looks like this:

```

dc_shell> set_input_delay -fall 1 D
Performing set_input_delay on port 'D'.
1
dc_shell> create_clock 5 CLK
Performing create_clock on clock 'CLK'.
1
dc_shell> report_constraint -verbose
Information: Updating design information... (UID-85)

*****
Report : constraint
        -verbose
Design : flop
Version : v3.0
Date    : Tues Jan 14 14:05:20 2002
*****

```

```

1. max_period 5.000000 { CLK }

Worst:           q_reg/D
Max Data Path   1.00
+ Library Setup 4.00
- Min Active Clock 0.00
- (Clock Period *
  Period Multiplier) 5.00
+ Minus Clock Skew 1.00
+ Minus Clock Skew 0.00
-----
Delta Cost       0.00 (MET)

The minimum clock period is 5.00 for signal 'CLK' (on 'q_reg/
d')
1

```

In the simulator, the vector specification looks like this:

```

time 0 : D_PORT = 0, CLOCK_PORT = 0; /*init signals */
time 100: D_PORT = 1;      /*apply data signal change */
time 104: CLOCK_PORT = 1; /*apply clk at period boundary of
4*/

```

The simulator output will show compliance. In some simulators, you may have to run tools that postprocess the simulation results to determine if they adhere to timing rules.

## Hold

Hold times are treated as constraint equation parameters by Design Compiler. To check the values of constraint equation parameters in the technology library, devise dynamic tests that check both the violating condition and the complying condition at the boundary of the violation.

For hold time verification, use the same approach you did for setup time verification. Use the same example as described for setup time validation, but assign a hold time constraint of 2 and test the boundary violation condition with the following commands:

```
dc_shell> set_input_delay 1 D_PORT
dc_shell> create_clock 4 CLOCK_PORT
dc_shell> report_constraint -verbose

2. Clocks_at_hold 4.000000 { CLOCK_PORT }

Worst:                      U1/D
Max Active Clock           0.00
+ Library Hold              2.00
- Min Data Path             1.00
-----
Delta Cost                  1.00 (VIOLATED)
```

In the simulator, the equivalent vector specification looks like this:

```
time 0 : D_PORT = 0, CLOCK_PORT = 0; /* init signals */
time 100: CLOCK_PORT = 1; /* apply clk */
time 101: D_PORT = 1;      /* apply data signal change */
```

The simulator output will also show a violation.

Use the following commands to validate the boundary condition for compliance in Design Compiler:

```
dc_shell> set_input_delay 2 D_PORT
dc_shell> create_clock 4 CLOCK_PORT;
dc_shell> report_constraint -verbose

2. Clocks_at_hold 4.000000 { CLOCK_PORT }

Worst:                      U1/D
Max Active Clock           0.00
+ Library Hold              2.00
- Min Data Path             2.00
-----
```

Delta Cost

0.00 (MET)

In the simulator, the equivalent vector specification looks like this:

```
time 0 : D_PORT = 0, CLOCK_PORT = 0; /* init signals */
time 100: CLOCK_PORT = 1; /* apply clk */
time 102: D_PORT = 1; /* apply data signal change */
```

The simulator output will also show compliance. In some simulators, you might have to run tools that postprocess the simulation results to determine if they adhere to timing rules.

## Rising and Falling Edges

You can use the `timing_type` command to validate pins that have timing arcs, by using a combination of the Design Compiler reporting features and the logic simulator of the target environment. For example, use the Design Compiler `set_input_delay -rise` and `set_input_delay -fall` commands, so that events occur at the input pins at the same time as they occur in the simulation. Then, in the simulator, design a two-vector input set that applies both the active edge and the inactive edge to the edge-sensitive input pin.

If you specify the event trigger correctly in the technology library, only the trigger event propagates through the cell in the Design Compiler timing report. The nontriggering event is masked. These results will be reflected in the simulator's output.

## Area

You can use the test circuit netlist to verify the area parameters of the technology library you want to test. The `report -area` command in Design Compiler reports the total area of the design. To compare this value with the actual area by manual inspection, use the following command:

```
dc_shell> report_area
```

## Ranges for Piecewise Linear Model

Using the `piece_define` attribute statement, you can define the ranges for pieces in the piecewise linear timing delay model at the library level. To verify the ranges, use the `report_lib` command:

```
lc_shell> report_lib TEST_CIRCUIT
```

Look for the following section in the library report:

Piecewise Linear Delay Model:

Piece	Length
-----	
1	0.0 <= length < 50.0
2	50.0 <= length < 100.0
3	100.0 <= length

The piecewise linear equations in this library use three pieces with breakpoints at 50 and 100 mils.

When the `piece_type` is capacitance, the report displays the following:

Piecewise Linear Delay Model:

Piece	Pin Capacitance
-----	
1	0.0 <= capacitance < 1.5
2	1.5 <= capacitance < 4.3
3	4.3 <= capacitance

---

## Testing the Interconnect Models

You must test each `wire_load` group in the technology library. You can use a simple inverter tree design to test the wire load specifications. Set up the test design to test a range of fanout counts.

For example, design a circuit in which one input inverter supplies the input signals for two inverters, one of which supplies the input signal for three inverters, and the other supplies the input signal for four, and so on, out to the highest fanout value defined in the `wire_load` group.

[Example 13-5](#) shows a possible test circuit in netlist format, and [Figure 13-3](#) shows the same test circuit as a schematic.

### *Example 13-5 Fanout Test Netlist*

```
entity FANOUT_TEST is
    port( I1, I2 : in BIT;  O1, O2, O3, O4, O5 : out BIT);

end FANOUT_TEST;

architecture STRUCTURAL_VIEW of FANOUT_TEST is
    component INVH
        port( A : in BIT;  Z : out BIT);
    end component;

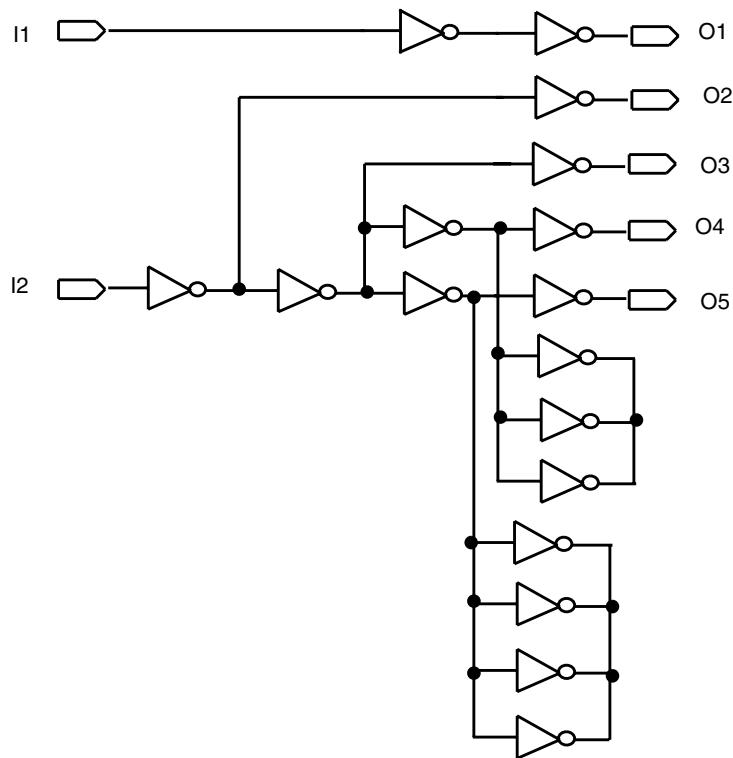
begin

    O1 <= I1;
    O2 <= I2;
    O4 <= I2;
    O5 <= I2;

    U1 : INVH  port map( A => I2,  Z => O3);

end STRUCTURAL_VIEW;
```

*Figure 13-3 Fanout Test Circuit Schematic*



In this example, the library under test has two wire load models defined: SMALL\_CHIP\_WIRE and BIG\_CHIP\_WIRE. Assume that the test circuit inverter tree is also in a VHDL netlist named FANOUT\_TEST.NET.

First, collect Design Compiler timing reports on the test circuit netlist, by using both of the defined wire load models:

```
dc_shell> read -f vhdl FANOUT_TEST.NET
dc_shell> set_wire_load SMALL_CHIP_WIRE
dc_shell> report_timing > fanout_test.small_chip
dc_shell> set_wire_load BIG_CHIP_WIRE
dc_shell> report_timing > fanout_test.big_chip
```

Then run the timing simulator on the fanout test circuit netlist and compare the results with those generated by Design Compiler.

---

## Testing Environmental Scaling

Use a design with a single inverter to test the environmental scaling performed under all defined conditions. Generate a Design Compiler timing report for each `operating_conditions` group in the library.

In the following example, the three operating condition groups, BCMIL, NOM, and WCMIL, are tested by use of the test circuit netlist `SINGLE_INV.NET`, which contains one inverter:

```
dc_shell> read -f vhdl SINGLE_INV.NET
dc_shell> set_operating_conditions BCMIL
dc_shell> report_timing > single_inv.bcmil
dc_shell> set_operating_conditions NOM
dc_shell> report_timing > single_inv.nom
dc_shell> set_operating_conditions WCMIL
dc_shell> report_timing > single_inv.wcmil
```

Compare the results with those from simple 2-input vector simulations, where the simulation delay values were computed with the values from the corresponding delay environments. This test also performs an implicit check on the environmental scaling factors in the technology library.



# Index

---

## A

advanced composite current source modeling  
  base curves 6-4  
  example 6-13  
Advanced composite current source power  
  modeling  
  Gate leakage current 8-6  
aggressor net, defined 10-2  
attributes, technology library  
  cell\_name 9-9  
  short 9-10

## B

balanced\_tree value of tree\_type 3-11, 3-49,  
  12-17  
best\_case\_tree value of tree\_type 12-17  
blocks  
  complex interface timing 9-3, 9-21  
  describing with interface timing 9-7  
Boolean  
  function verification 13-7  
  operators 1-12  
boundary condition  
  validation 13-26  
  violation 13-25, 13-28

## C

calc\_mode attribute 12-16  
capacitance, pin  
  delay modeling 3-15  
  in delay calculation 3-12, 3-46, 3-48  
  scaling 12-43  
  setting default 12-6  
capacitance, wire  
  in delay calculation 3-12, 3-46, 3-48  
  scaling 12-44  
capacitive power 1-5  
cell\_fall group  
  defined 4-69  
  defining delay arcs 4-69  
  in nonlinear delay models 4-33  
cell\_leakage\_power attribute 1-10  
cell\_name attribute 9-9  
cell\_rise group  
  defined 4-69  
  defining delay arcs 4-69  
  in nonlinear delay models 4-33  
cells 11-10, 11-33, 11-36  
  modeling for power 1-7  
  sequential timing 9-4  
  timing  
    differences in specifying 9-8  
    interpreting 9-4

checking libraries 13-4  
 clock gating  
     benefits of 1-96  
     circuits that benefit 1-96  
     clock tree synthesis 1-107  
     illustration without 1-97  
     latch-based 1-98, 1-99  
     register bank, definition 1-96  
     with integrated cells 1-96  
 clock pin, setup and hold checks 4-96, 4-97  
`clock_gate_clock_pin` attribute 1-101  
`clock_gate_enable_pin` attribute  
     in pin group 1-102  
`clock_gate_obs_pin` attribute 1-103  
`clock_gate_out_pin` attribute 1-104  
`clock_gate_test_pin` attribute 1-103  
 clock-pin attribute 9-12  
 CMOS nonlinear delay model  
     submicron delay model 3-20  
 combinational path  
     output propagation delay 9-20  
 combinational timing arc  
     definition 4-4  
     use of 4-3  
 composite current source  
     lookup table model 5-3, 5-8  
     `output_current_template` group 5-3  
     receiver capacitance group 5-9  
     receiver information 5-8  
     `reference_time` simple attribute 5-5  
     representing driver information 5-2  
     template variables 5-4  
     vector group 5-5  
 composite current source modeling, example  
     5-20  
 Composite current source power modeling 8-2  
     Cell leakage current 8-3  
     Dynamic power 8-14  
     Examples 8-28  
     Intrinsic parasitic 8-10  
 Composite current source signal integrity noise model 7-2  
 Compiling 7-2  
 Sample noise library 7-9  
 Screening 7-4  
 Syntax 7-3  
 conditional timing constraints, attributes and groups 4-120  
 connect delay 3-38  
     balanced case equation 3-49  
     best case equation 3-48  
     in interconnect delay 3-15, 3-52  
     worst case equation 3-49  
`constrained_pin_transition`, value for transition constraint 4-100  
 constraint  
     attribute 4-126  
     evaluation 12-20  
     in timing report 11-11  
     load-dependent 4-25  
`constraint_high` attribute 4-124  
`constraint_low` attribute 4-124

## D

D flip-flop  
     CMOS piecewise linear delay model 4-133  
     CMOS scalable polynomial delay model 4-140  
     CMOS standard delay model 4-131, 4-137  
     timing arcs 4-107  
`dc_shell` commands  
     `report_lib` 12-20  
     `set_operating_conditions` 12-18  
     `set_timing_ranges` 12-20  
     `set_wire_load` 12-26  
 DCM  
     delay model description 3-56  
     IEEE1481.1 standard 3-56  
     timing information ignored 3-57  
`default_cell_leakage_power` attribute 1-17, 12-3

default\_leakage\_power\_density attribute,  
     coexistence with 1-17, 12-3  
 default\_connection\_class attribute 12-13  
 default\_fall\_delay\_intercept attribute 12-9  
 default\_fall\_pin\_resistance attribute 12-9  
 default\_fanout\_load attribute 12-6  
 default\_inout\_pin\_rise\_res attribute 12-7  
 default\_input\_pin\_cap attribute 12-6  
 default\_intrinsic\_fall attribute 12-8, 12-9  
 default\_intrinsic\_rise attribute 12-8, 12-9  
 default\_leakage\_power\_density attribute 1-18,  
     12-4  
     default\_cell\_leakage\_power attribute,  
         coexistence with 1-18, 12-4  
 default\_max\_fanout attribute 12-6  
 default\_max\_transition attribute 12-6  
 default\_max\_utilization attribute 12-11  
 default\_min\_porosity attribute 12-12  
 default\_operating\_conditions attribute 12-13  
 default\_output\_pin\_cap attribute 12-6  
 default\_output\_pin\_fall\_res attribute 12-7  
 default\_output\_pin\_rise\_res attribute 12-7  
 default\_rise\_pin\_resistance attribute 12-9  
 default\_slope\_fall attribute 12-7  
 default\_slope\_rise attribute 12-7  
 default\_wire\_load attribute 12-10  
 default\_wire\_load\_area attribute 12-11  
 default\_wire\_load\_capacitance attribute 12-11  
 default\_wire\_load\_resistance attribute 12-11  
 degradation tables  
     for transition delay 4-79  
     variables 4-80  
 delay  
     calculation example 3-17, 3-54  
     connect 3-4, 3-10, 3-38  
     interconnect 3-15, 3-52  
     intrinsic 3-4, 3-8, 3-38, 3-43  
     model 4-21  
         and timing group attributes 4-32  
         piecewise linear 12-8  
     polynomial 4-30  
     use of 4-21  
 propagation, defined 4-70  
 scaling 3-18, 3-40, 3-55  
     wire capacitance 12-43  
 slope 3-4, 3-7, 3-38, 3-44  
     transition 3-4, 3-9, 3-38, 3-45  
 delay information, describing 11-11  
 DELAY Line 11-17  
 delay noise, defined 10-2  
 delay\_model attribute 4-23, 4-32  
 delays, timing report 11-17  
 delta  
     process 3-6, 3-30, 3-40  
     temp 3-6, 3-30, 3-40  
     voltage 3-6, 3-30, 3-40  
 divided\_by attribute 9-12  
 Dpropagation 3-24  
 drive capability 4-64  
 Dtransition 3-21, 3-24  
 duty\_cycle attribute 9-12  
 dynamic power, defined 1-3

## E

edges attribute 9-15  
 edge-sensitive timing arcs 4-54  
 electromigration  
     group 1-118  
     modeling 1-111  
 em\_lut\_template group 1-115  
 em\_max\_toggle\_rate group 1-120  
 em\_temp\_degradation\_factor attribute 12-14  
 enable pin of three-state function 4-51  
 environment, describing 12-1  
 environmental derating factors 1-22  
 equal\_or\_opposite\_output attribute 1-57  
 equal\_or\_opposite\_output\_net\_capacitance  
     1-38

## F

factors, power-scaling 1-75  
fall\_constraint group  
    modeling load dependency 4-84  
    no-change constraint hold time 4-115, 4-116  
    setup and hold constraints 4-100  
fall\_delay\_intercept attribute  
    in piecewise linear delay models 4-33  
    scaling 12-47  
    specifying default value 12-9  
fall\_pin\_resistance attribute  
    in piecewise linear delay models 4-33  
    scaling 12-46  
    specifying default value 12-9  
    verification 13-22  
fall\_power group  
    attributes 1-67  
    internal\_power group 1-66  
fall\_propagation group  
    defined 4-70  
    defining delay arcs 4-69  
    in nonlinear delay models 4-33  
fall\_resistance attribute  
    definition 4-65  
    in linear delay models 4-65  
    specifying default value 12-7  
    verification 13-21  
fall\_transition\_degradation group 4-79  
falling\_edge value for timing\_type attribute  
    4-54  
falling\_edge value, of timing\_type 4-41  
falling\_together\_group attribute 1-57  
fanout  
    and wire length estimation 12-22  
    in interconnect delay 3-15, 3-52  
    test netlist example 13-31  
fanout\_area attribute 12-30  
fanout\_capacitance attribute 12-30  
fanout\_length attribute 12-30  
    in delay calculation 3-10

fanout\_load attribute, specifying default value 12-6  
fanout\_resistance attribute 12-30  
fixed delay 3-8, 3-43  
flip-flops, register bank of 1-96  
function attribute, verification 13-8  
functional noise, defined 10-2

## G

generated\_clock group 9-11

## H

hold constraints  
    and VHDL library generator 4-98  
    defined 4-95  
    hold\_falling value 4-98  
    hold\_rising value 4-98  
    in linear delay models 4-98  
    in nonlinear delay models 4-100, 4-103  
hold\_falling value  
    defined 4-98  
    for timing\_type attribute 4-101  
hold\_falling, value of timing\_type 4-41  
hold\_rising value  
    defined 4-98  
    for timing\_type attribute 4-101  
hold\_rising value, of timing\_type 4-41  
hyperbolic\_noise groups 10-32

## I

index\_1 attribute  
    fall\_power group 1-67  
    in em\_max\_toggle\_rate group 1-121  
    power\_lut\_template group 1-39  
    rise\_power group 1-69, 1-71  
index\_2 attribute  
    fall\_power group 1-67  
    in em\_lut\_template group 1-117

in em\_max\_toggle\_rate group 1-121  
 power\_lut\_template group 1-39  
 rise\_power group 1-69, 1-71  
**index\_3** attribute  
 fall\_power group 1-67  
 power\_lut\_template group 1-39  
 rise\_power group 1-69, 1-71  
**input**  
 hold constraint 9-17  
 vectors, in simulation 13-11  
**input noise width ranges**, defined 10-30  
**input\_net\_transition** variable, power 1-42  
**input\_transition\_time** 1-38  
**input\_voltage** group 1-94  
**integrated cells**  
 clock gating 1-96, 1-100  
**integrated clock gating cell** 1-98  
**interconnect**  
 delay 3-15, 3-52  
 model, defining 12-17  
 power 1-31  
 tree topologies 3-13, 3-50  
**Interface Timing Specification**  
 interface timing relationships  
 examples 9-17  
 using blocks 9-26  
**internal power**  
 calculating 1-26, 1-27, 1-29  
 defined 1-4, 1-23  
 examples 1-76  
 library example 1-74  
 modeling choices 1-24  
 template information 11-7, 11-36  
**internal\_power** group  
 definition 1-45  
 equal\_or\_opposite\_output attribute 1-57  
 fall\_power attribute 1-66  
 falling\_together\_group attribute 1-57  
 one-dimensional table 1-73  
 power group attribute 1-14, 1-69  
 related\_pin attribute 1-61  
 rise\_power attribute 1-71  
 rising\_together\_group attribute 1-62  
 switching\_interval attribute 1-63  
 switching\_together\_group attribute 1-64  
 two-dimensional table 1-73  
 when attribute 1-65  
**intrinsic delay** 3-4, 3-8, 3-38, 3-43  
 defined 4-69  
 for input pin 4-61  
 for output pin 4-61  
 in linear delay models 4-62  
 in nonlinear delay models 4-63, 4-64  
 in piecewise linear models 4-63  
**intrinsic\_fall** attribute  
 in linear delay models 4-33, 4-62  
 specifying default value 12-8, 12-9  
**intrinsic\_rise** attribute  
 in linear delay models 4-33, 4-62  
 specifying default value 12-8, 12-9  
**invert** attribute 9-13  
**inverter with three-state output** 4-52  
**ITS** see **Interface Timing Specification**  
**I-V characteristics curve**  
 polynomial model 10-16  
**iv\_lut\_template** group 10-13

## J

JK flip-flop, example 4-55

## K

**k-factors**  
 defined 3-6, 3-30, 3-41, 12-35  
 example of 12-58  
 pin resistance 12-45  
**process attributes**  
 cell power 12-48  
 drive fall/rise 12-42  
 fall delay intercept 12-47  
 hold rise/fall constraints 12-50

internal power 12-48  
intrinsic fall/rise delay 12-40  
minimum period 12-54  
minimum pulse width 12-53  
pin capacitance 12-43  
recovery rise/fall constraints 12-53  
rise delay intercept 12-47  
rise pin resistance 12-45  
setup rise constraints 12-62  
setup rise/fall constraints 12-51, 12-52  
slope fall/rise 12-41  
wire capacitance 12-44  
wire resistance 12-45  
slope-sensitivity 12-41  
temperature attributes  
  cell leakage power 12-48  
  drive fall/rise 12-43  
  fall delay intercept 12-47  
  fall pin resistance 12-46  
  hold rise/fall constraints 12-50  
  internal power 12-48  
  intrinsic fall/rise delay 12-40  
  minimum period 12-54  
  minimum pulse width 12-54  
  pin capacitance 12-44  
  recovery rise/fall constraints 12-53  
  rise pin resistance 12-46  
  setup rise constraints 12-62  
  setup rise/fall constraints 12-51, 12-52  
  slope fall/rise 12-41  
  wire capacitance 12-44  
  wire resistance 12-45  
voltage attributes  
  cell leakage power 12-48  
  drive fall/rise 12-43  
  fall delay intercept 12-47  
  fall pin resistance 12-46  
  hold rise/fall constraints 12-51  
  internal power 12-48  
  intrinsic fall/rise delay 12-40  
  minimum period 12-54  
  minimum pulse width 12-54

pin capacitance 12-44  
recovery rise/fall constraints 12-53  
rise pin resistance 12-46  
setup rise/fall constraints 12-52, 12-62  
slope fall/rise 12-41  
wire capacitance 12-44  
wire resistance 12-45  
wire capacitance 12-43

## L

latch-based clock gating 1-99  
leakage power modeling 1-9  
leakage\_power group 1-10, 1-13  
leakage\_power\_unit attribute 1-16  
library  
  file, comparing 13-3  
  identifying errors 13-12  
library group (technology library)  
  default pin attributes 12-5  
  default timing attributes 12-7, 12-8  
  em\_temp\_degradation\_factor attribute 1-122  
  intercept delay scaling factors 12-46  
  intrinsic delay scaling factors 12-40  
  multiple power supplies 1-89  
  pin resistance scaling factors 12-45  
  power\_supply 1-84, 1-87  
  slope-sensitivity scaling factors 12-41  
library group in technology library  
  input\_voltage group 1-94  
  voltage\_unit attribute 1-94  
link\_library, dc\_shell variable 9-27  
load dependency modeling 4-83  
load-dependent constraints, template  
  variables 4-25  
lookup tables  
  for modeling load dependency 4-84, 4-85  
power  
  one-dimensional table 1-73  
  three-dimensional table 1-74  
  two-dimensional table 1-73

syntax for lookup table group 4-28  
lu\_table\_template group  
breakpoints 1-39, 4-27  
defining 5-9, 5-13  
modeling load dependency 4-83, 4-85  
variables 4-24

## M

mapping 13-8  
master\_pin attribute 9-14  
max\_clock\_tree\_path  
    timing\_type value 4-43  
max\_fanout attribute  
    specifying default value 12-6  
max\_input\_noise\_width 10-30  
max\_transition attribute  
    specifying default value 12-6  
min\_clock\_tree\_path  
    timing\_type value 4-44  
min\_input\_noise\_width 10-30  
min\_pulse\_width group  
    defined 4-124  
minimum\_period  
    timing\_type value 4-43  
minimum\_period group  
    defined 4-126  
minimum\_pulse\_width  
    timing\_type value 4-43  
mode attribute 4-46  
model group 9-9  
modeling electromigration 1-111  
modeling timing arcs 4-6, 4-7  
multiple paths 4-36  
multiplied\_by attribute 9-14

## N

negative\_unate value  
    for timing\_sense attribute 4-57, 4-59  
    in clear arc 4-59

    in preset arc 4-57  
no-change constraints  
    and conditional attributes 4-127  
    defined 4-111  
    in linear delay models 4-114  
    in nonlinear delay models 4-115, 4-116  
noise  
    characteristics 10-11  
    characterizing 10-4  
    defining steady-state current groups 10-15  
    failure voltage criteria 10-8  
    hyperbolic model 10-10  
    immunity 10-6  
    immunity curve characterization 10-8  
    input noise width ranges 10-30  
    I-V characteristics and drive resistance 10-4  
    I-V characteristics curve, polynomial model  
        10-16  
    I-V characteristics lookup table model 10-13  
    iv\_lut\_template group 10-13  
    lookup template variables 10-14, 10-28  
    modeling cells 10-4  
    noise calculation defined 10-3  
    noise immunity defined 10-3  
    noise propagation defined 10-3  
    nonlinear delay model, example 10-52  
    propagated noise 10-34  
    propagated noise groups  
        for polynomial, defined 10-41  
    propagated noise polynomial model 10-39  
    propagated noise table groups, defined  
        10-36  
    propagation 10-10  
    representing calculation information 10-12  
    scalable polynomial model, example 10-44  
    summary of library requirements 10-11  
    template variables 10-14  
noise calculation, defined 10-3  
noise glitch, calculating 10-4  
noise hyperbolic model 10-10  
noise immunity curve characterization 10-8  
noise immunity curve, modeling 10-7

noise immunity lookup table model 10-23  
noise immunity polynomial groups, defined 10-29  
noise immunity polynomial model 10-27  
noise immunity table groups, defined 10-21, 10-25  
noise immunity, defined 10-3  
noise library requirements 10-11  
noise propagation, defined 10-3  
noise steady\_state\_resistance simple attributes 10-20  
noise template variables for poly\_template group 10-40  
noise\_tied\_off attribute 10-21  
noise\_lut\_template group 10-23, 10-24  
nom\_process attribute 3-6, 3-30, 3-40  
nom\_temperature attribute 3-6, 3-30, 3-40  
nom\_voltage attribute 3-6, 3-30, 3-40  
non\_seq\_hold\_falling value defined 4-105  
for timing\_type attribute 4-101  
non\_seq\_hold\_rising value defined 4-105  
for timing\_type attribute 4-101  
non\_seq\_setup\_falling value defined 4-105  
for timing\_type attribute 4-101  
non\_seq\_setup\_rising value defined 4-105  
for timing\_type attribute 4-101  
non\_unate value for timing\_sense attribute 4-57, 4-59  
in clear arc 4-59  
in preset arc 4-57  
nonlinear delay model in report information 11-7  
NOR gate example 4-67

## O

operating\_conditions group 3-6, 3-11, 3-30, 3-40, 3-48  
calc\_mode 12-16  
effect on input voltage groups 1-94  
effect on output voltage groups 1-95  
group statement 12-15  
power\_rail 12-16  
process attribute 12-16  
setting default group 12-13  
specifying at runtime 12-17  
temperature attribute 12-16  
tree\_type attribute 12-17  
voltage attribute 12-16  
optimization  
register transfer level 1-96  
strategies 13-1  
output current groups, defining 5-4  
output propagation 9-18  
output\_voltage group 1-95

## P

pads, input voltage levels 1-94  
Parasitics modeling in a macro cell 8-12  
path tracing  
arcs 4-37  
arcs and timing analysis 4-3  
edge-sensitive timing arcs 4-54  
piece\_define attribute 3-42, 3-43, 4-66  
verification 13-23  
piecewise linear delay model  
description 3-41  
verification of resistance 13-22  
pin group statement  
clock\_gate\_enable\_pin attribute 1-102  
internal\_power group  
equal\_or\_opposite\_output attribute 1-57  
fall\_power attribute 1-66  
falling\_together\_group attribute 1-57  
power group 1-14, 1-69

**related\_pin** attribute 1-61  
**rise\_power** attribute 1-71  
**rising\_together\_group** attribute 1-62  
**switching\_interval** attribute 1-63  
**switching\_together\_group** attribute 1-64  
 when attribute 1-65  
**pins**, transitioning together 1-25  
**polarity** transition 4-36  
**poly\_template** group 4-30, 10-17, 10-27,  
 10-40  
**polynomial delay model** 4-30  
**positive\_unate** value  
 for **timing\_sense** attribute 4-57, 4-59  
 in clear arc 4-59  
 in preset arc 4-57  
**postprocessing tools** 13-27  
**power**  
 capacitive power defined 1-5  
 cell modeling 1-7  
 dynamic power defined 1-3  
 internal power, calculating 1-76  
**internal\_power** group 1-56  
 leakage power defined 1-9  
 lookup template variables 1-38, 1-42  
 equal\_or\_opposite\_output\_net\_capacitan  
 ce 1-38  
 input\_transition\_time 1-38  
 total\_output\_net\_capacitance 1-38, 1-42  
 total\_output2\_net\_capacitance 1-42  
 pins transitioning together, lower  
 consumption 1-25  
**power\_lut\_template** group 1-37  
 static power defined 1-3  
 switching activity defined 1-6  
**power group**  
 in **internal\_power** group 1-14, 1-69  
 values attribute 1-69  
**power lookup table template example** 1-39  
**power lookup tables**  
 one-dimensional table 1-73, 1-76  
 three-dimensional table 1-80  
 two-dimensional table 1-73, 1-78  
**power\_lut\_template** group 1-37  
**power\_poly\_template** variables  
 input\_net\_transition 1-42  
 temperature 1-42  
 total\_output\_net\_capacitance 1-42  
 voltage 1-42  
**power\_rail** attribute 12-16  
**power\_supply** group statement 1-84, 1-87  
**power-scaling factors** 1-75  
**process** attribute 3-6, 3-30, 3-40, 12-16  
**propagation delay** 3-24  
 as function of supply voltage 12-37  
 as function of temperature 12-38  
**propagation\_lut\_template** group 10-35

## R

**RAM**, specifying with interface timing 9-29  
**ramp time nominal, in slope delay** 3-7, 3-44  
**range of bus members**, in **related\_pin** 4-34  
**receiver\_capacitance** group, defining  
 at pin level 5-9  
 at timing level 5-14  
**recovery constraints** 4-107  
**recovery\_falling** value  
 for **timing\_type** attribute 4-42, 4-102  
 using 4-108  
**recovery\_rising** value  
 for **timing\_type** attribute 4-42, 4-101  
 using 4-108  
**reference\_time simple** attribute 5-5  
**related\_bus\_pins** attribute  
 defined 4-35  
 in all delay models 4-32  
**related\_output\_pin** attribute in nonlinear delay  
 models 4-33  
**related\_pin** attribute  
 defined 4-34  
 in all delay models 4-32

in hold arcs 4-98  
 internal\_power group 1-61  
**related\_pin\_transition** value for transition constraint 4-101  
 removal constraints, defined 4-109  
**removal\_falling** value  
     for timing\_type attribute 4-43, 4-102  
     using 4-110  
**removal\_rising** value  
     for timing\_type attribute 4-43, 4-102  
     using 4-110  
**report command** 13-29  
**report\_delay\_calculation command** 13-16  
**report\_timing command** 13-19  
**resistance**  
     specifying default value 12-45  
     wire attribute  
         in delay calculation 3-12, 3-47  
**retaining time**  
     delay 4-78  
     for nonlinear delay model only 4-74, 4-76  
**retaining\_fall group** 4-73, 4-76  
**retaining\_rise group** 4-73, 4-76  
**retaining\_fall group**  
     defined 4-73, 4-76  
     example 4-78  
**retaining\_rise group**  
     defined 4-73, 4-76  
     illustration 4-74, 4-77  
**rise\_constraint group**  
     modeling load dependency 4-84  
     no-change constraint setup time 4-115, 4-116  
     setup and hold constraints 4-100  
**rise\_delay\_intercept attribute**  
     in piecewise linear delay models 4-33  
     scaling 12-47  
     specifying default value 12-8  
     verification 13-22  
**rise\_pin\_resistance attribute** 3-42, 3-43  
     in piecewise linear delay models 4-33  
     scaling 12-45  
     specifying default value 12-9  
**rise\_power group**  
     attributes 1-69, 1-71  
     internal\_power group 1-71  
**rise\_propagation group**  
     defined 4-70  
     defining delay arcs 4-69  
     in nonlinear delay models 4-33  
**rise\_resistance attribute**  
     defined 4-64  
     in linear delay models 4-33  
     specifying default value 12-7  
     verification 13-21  
**rise\_transition group** in nonlinear delay models 4-33  
**rise\_transition\_degradation group** 4-79  
**rising\_edge** value for timing\_type attribute 4-41, 4-54  
**rising\_together\_group attribute** 1-62  
**RTL optimization** 1-96

## S

**scalable polynomial delay model** 4-30  
     description 3-33  
     template 4-30  
**scalar power\_lut\_template group** 1-40  
**scaled\_cell group** 11-11  
**scaling factors**  
     for nonlinear delay model 12-60  
     intrinsic delay 12-40  
     power 1-75, 12-48  
     slope-sensitivity 12-41  
**SDF file, generating** 4-128  
**sdf\_cond attribute** 4-89, 4-92  
     conditional timing constraint 4-121  
     defined 4-121  
     in min\_pulse\_width group 4-125  
     in minimum\_period group 4-127  
     state-dependent timing constraint 4-92

- with no-change constraints 4-127
- sdf\_cond\_end** attribute
  - defined 4-123
  - with no-change constraints 4-127
- sdf\_cond\_start** attribute
  - defined 4-122
  - with no-change constraints 4-127
- sdf\_edges** attribute
  - defined 4-123
  - with no-change constraints 4-127
- sequential timing arc**
  - defined 4-4
  - types 4-5
  - use of 4-3
- set\_drive** command 13-23
- set\_input\_delay** command 13-25
  - fall option 13-29
- set\_load** command 13-22, 13-24
- set\_wire\_load** command 12-26
- setup and hold checking** 13-26
- setup constraints**
  - and VHDL library generator 4-98
  - defined 4-95
  - in linear delay models 4-96
  - in nonlinear delay models 4-100, 4-103
  - setup\_falling** value 4-96
  - setup\_rising** value 4-96
- setup\_falling** value for **timing\_type** attribute 4-42, 4-101
- setup\_rising** value for **timing\_type** attribute 4-41, 4-101
- shifts** attribute 9-15
- short** attribute 9-10
- short-circuit power** 1-23
- signal behavior** 13-26
- skew constraints** 4-118
  - defined 4-118
- skew\_falling** value
  - defined 4-119
  - for **timing\_type** attribute 4-42, 4-101
- skew\_rising** value for **timing\_type** attribute 4-42, 4-101, 4-119
- slope**
  - delay** 3-4, 3-7, 3-38, 3-44
  - equation** 3-44
  - describing sensitivity 4-87
- slope\_fall** attribute 4-88
  - in linear delay models 4-33
  - in piecewise linear delay models 4-33
- slope\_rise** attribute
  - in linear delay models 4-33
  - in piecewise linear delay models 4-33
- standard delay model, verification of resistance** 13-21
- startpoint, timing arc** 4-3
- state-dependent timing attributes** 4-88
- static power**, defined 1-3
- steady\_state\_current groups** 10-15
- steady-state current groups, polynomial** 10-18
- submicron delay modeling** 3-20
- switching**
  - activity** 1-6
  - power** 1-31
- switching\_together\_group** attribute 1-64
- synchronous load-enable**
  - figure 1-96
  - in a register bank 1-96
- synthesis**
  - achieving higher quality results 9-6

## T

- target\_library, dc\_shell variable** 9-27
- technology library**
  - cell internal power attributes** 1-45
  - em\_temp\_degradation\_factor** 1-122
- temperature attribute** 3-6, 3-30, 3-40, 12-16
- temperature variable, power** 1-42
- template**
  - for nonlinear delay models 4-23
  - in report information 11-7, 11-36

report for nonlinear delay model 11-7, 11-36  
 test circuit, preparation for verification 13-4  
 test vectors 13-11  
 three\_state\_disable timing arc 4-51  
 three\_state\_enable timing arc 4-52  
 three-state cell, timing\_sense attribute 4-36  
 tied\_off attribute, defining usage 10-21  
 tied\_off cells 10-21  
 time borrowing 9-28  
 time-of-flight delay 3-10, 3-25  
 timing
 

- DCM delay model, timing information ignored 3-57
- delays, template variables 4-24
- describing delay 4-1
- information
  - in timing report 11-11
  - statistics 11-24
- interface 9-27
- model 12-7
- ranges 12-18
- setting constraints 4-2
- transparent latch 4-146

timing arcs
 

- clear 4-58
- constraint arc defined 4-4
- defining identical 4-34
- delay arc defined 4-4
- diagram 4-3
- modeling 4-6
- naming 1-45, 4-8
- preset 4-56
- with VHDL library generator 4-8

timing group
 

- defined 4-7
- fall\_delay\_intercept attribute 4-66
- fall\_pin\_resistance attribute 4-67
- fall\_resistance attribute 4-65
- intrinsic\_fall attribute 4-62
- intrinsic\_rise attribute 4-62
- related\_pin attribute 4-34

rise\_delay\_intercept attribute 4-66  
 rise\_pin\_resistance attribute 4-66  
 rise\_resistance attribute 4-64  
 slope\_fall attribute 4-88  
 slope\_rise attribute 4-88  
 timing\_sense attribute 4-36  
 timing\_type attribute 4-37
 

- values 4-40

timing interface
 

- benefits 9-5

timing parameter units 3-5, 3-39

timing\_range group
 

- example 12-19
- faster\_factor attribute 12-19

timing\_sense attribute 4-32

timing\_type attribute
 

- defining preset and clear arcs 4-56, 4-58
- in all delay models 4-32
- no-change constraint values 4-113
- nonsequential constraint values 4-105
- recovery time constraint values 4-108
- removal constraint values 4-110
- setup and hold arc values 4-96, 4-98
- skew constraint values 4-119
- values for 4-38

total delay 3-38
 

- equation 3-3
- scaling 3-5

total\_output\_net\_capacitance 1-38

total\_output\_net\_capacitance variable, power 1-42

transition delay 3-4, 3-9, 3-24, 3-38, 4-66, 4-67, 4-68
 

- defined 4-64
- degradation 4-79
- equation 3-9
- fall\_resistance attribute 4-65
- in interconnect delay 3-15, 3-52
- in linear delay models 4-64
- in nonlinear delay models 4-68
- in piecewise linear delay models 4-66

rise\_resistance attribute 4-64  
used in slope delay calculation 3-7, 3-44  
transparent latch timing model 4-146  
tree\_type attribute, values 3-49

## U

unate, defined 4-36

## V

value attribute 1-12  
values attribute  
    definition 1-67, 1-69, 1-71  
    in em\_max\_toggle\_rate group 1-122  
variables, dc\_shell  
    target\_library 9-27  
Variation-aware timing modeling support 6-15  
    Constraint model 6-28  
    Driver model 6-16  
    Examples 6-32  
    Rceiver model 6-24  
vector group 5-5  
verification  
    constraint parameters 13-25  
    creating a test-mapping library 13-8  
    creating the test vectors 13-11  
    delay scaling 13-33  
    hold times 13-27  
    identifying function errors 13-12  
    interconnect model 13-31  
    intrinsic delay 13-21  
    mapping the netlist 13-10  
    operating\_conditions group 13-33  
    resistance parameters  
        piecewise linear model 13-22  
        standard delay model 13-21  
    simulating the netlists 13-12  
    slope 13-24  
    timing parameters 13-20  
    timing\_type 13-29

wire\_load group 13-31  
wire-length range in piecewise model 13-30  
victim net, defined 10-2  
vih voltage range 1-93  
vil voltage range 1-93  
vimax voltage range 1-94  
vimin voltage range 1-93  
voltage attribute 3-6, 3-30, 3-40, 12-16  
voltage input and output levels 1-94  
voltage variable, power 1-42  
voltage\_unit attribute 1-94, 1-95

## W

when attribute 1-11, 4-89  
    conditional timing constraint 4-121  
    defined 4-89, 4-121  
    in min\_pulse\_width group 4-125  
    in minimum\_period group 4-127  
    internal\_power group 1-65  
    state-dependent timing constraint 4-89  
    with no-change constraints 4-127  
when\_end attribute  
    defined 4-122  
    with no-change constraints 4-127  
when\_start attribute  
    defined 4-121  
    with no-change constraints 4-127  
wire  
    capacitance, in delay calculation 3-10  
    resistance, in delay calculation 3-10  
wire length in transition delay 3-10, 3-12, 3-46, 3-48  
wire\_load group 3-12, 3-46, 3-48  
    capacitance attribute 12-23  
    fanout\_length attribute 12-23, 12-24  
    group statement 12-22  
    resistance attribute 12-23  
    slope attribute 12-23  
    specifying at runtime 12-26  
wire\_load\_from\_area 12-32

wire\_load\_selection group 12-22, 12-31  
wire\_load\_table group  
fanout\_area attribute 12-30  
fanout\_capacitance attribute 12-30  
fanout\_length attribute 12-30  
fanout\_resistance attribute 12-30

worst\_case\_tree value of tree\_type 3-49

## Z

zero-load delay 3-43