

Liberty User Guide Volume 1

Version R-2020.09, September 2020

Copyright Notice

© 2004, 2006-2009, 2011-2020 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 use of Open Source Liberty users.

Destination Control Statement

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

Disclaimer

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

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at <https://www.synopsys.com/company/legal/trademarks-brands.html>.

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

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content. Synopsys, Inc.
690 E. Middlefield Road
Mountain View, CA 94043
www.synopsys.com

Contents

1. Sample Library Description	27
General Syntax	27
Statements	28
Group Statements	29
Attribute Statements	29
Simple Attributes	29
Complex Attributes	30
Define Statements	30
Reducing Library File Size	31
 2. Building a Logic Library	 33
Creating Library Groups	33
library Group	33
Using General Library Attributes	34
technology Attribute	34
delay_model Attribute	34
bus_naming_style Attribute	35
Delay and Slew Attributes	35
input_threshold_pct_fall Simple Attribute	37
input_threshold_pct_rise Simple Attribute	38
output_threshold_pct_fall Simple Attribute	38
output_threshold_pct_rise Simple Attribute	39
slew_derate_from_library Simple Attribute	39
slew_lower_threshold_pct_fall Simple Attribute	40
slew_lower_threshold_pct_rise Simple Attribute	40
slew_upper_threshold_pct_fall Simple Attribute	41
slew_upper_threshold_pct_rise Simple Attribute	41
Defining Units	42
time_unit Attribute	42
voltage_unit Attribute	43
current_unit Attribute	43

Contents

Copyright and Proprietary Information Notice

pulling_resistance_unit Attribute	43
capacitive_load_unit Attribute	44
leakage_power_unit Attribute	44
Setting Minimum Cell Requirements	45
3. Building Environments	46
Library-Level Default Attributes	46
Setting Default Cell Attributes	46
default_cell_leakage_power Simple Attribute	46
Setting Default Pin Attributes	46
Setting Wire Load Defaults	47
default_wire_load Attribute	47
default_wire_load_capacitance Attribute	48
default_wire_load_resistance Attribute	48
default_wire_load_area Attribute	48
Setting Other Environment Defaults	49
default_operating_conditions Attribute	49
default_connection_class Attribute	49
Examples of Library-Level Default Attributes	49
Defining Operating Conditions	50
operating_conditions Group	50
Defining Power Supply Cells	51
power_supply group	51
Defining Wire Load Groups	52
wire_load Group	52
wire_load_table Group	56
Specifying Delay Scaling Attributes	56
Pin and Wire Capacitance Factors	57
Scaling Factors Associated With the Nonlinear Delay Model	57
4. Defining Core Cells	59
Defining cell Groups	59
cell Group	60
area Attribute	61
cell_footprint Attribute	61
clock_gating_integrated_cell Attribute	61

Contents

Copyright and Proprietary Information Notice

Setting Pin Attributes for an Integrated Cell	63
Setting Timing for an Integrated Cell	63
contention_condition Attribute	64
is_macro_cell Attribute	64
pad_cell Attribute	64
physical_variant_cells Attribute	65
pin_equal Attribute	66
pin_opposite Attribute	66
type Group	66
cell Group Example	67
mode_definition Group	68
Group Statement	69
mode_value Group	69
Defining pin Groups	70
pin Group	70
General pin Group Attributes	71
capacitance Attribute	72
clock_gate_clock_pin Attribute	73
clock_gate_enable_pin Attribute	73
clock_gate_obs_pin Attribute	73
clock_gate_out_pin Attribute	74
clock_gate_test_pin Attribute	74
complementary_pin Simple Attribute	74
connection_class Simple Attribute	75
direction Attribute	76
driver_type Attribute	77
fall_capacitance Attribute	81
fault_model Simple Attribute	82
inverted_output Attribute	83
is_analog Attribute	83
pin_func_type Attribute	84
steady_state_resistance Attributes	84
test_output_only Attribute	85
Describing Design Rule Checks	85
fanout_load Attribute	85
max_fanout Attribute	86
min_fanout Attribute	87
max_transition Attribute	87
max_trans Group	88
min_transition Attribute	90
max_capacitance Attribute	91
max_cap Group	91

Contents

Copyright and Proprietary Information Notice

min_capacitance Attribute	93
cell_degradation Group	93
Describing Clocks	95
clock Attribute	95
min_period Attribute	95
min_pulse_width_high and min_pulse_width_low Attributes	95
Describing Clock Pin Functions	96
internal_node Attribute	100
Defining Bused Pins	100
type Group	100
bus Group	102
bus_type Attribute	102
Pin Attributes and Groups	102
Example Bus Description	103
Defining Signal Bundles	106
bundle Group	107
members Attribute	107
pin Attributes	108
Defining Layout-Related Multibit Attributes	109
Defining Multiplexers	110
Library Requirements	110
Defining Decoupling Capacitor Cells, Filler Cells, and Tap Cells	112
Syntax	112
Cell-Level Attributes	112
is_decap_cell	112
is_filler_cell	112
is_tap_cell	113
Fault Tolerant Cell Modeling	113
Library-Level Attributes and Groups	114
altitude_unit Attribute	114
soft_error_rate_template Group	114
default_soft_error_rate Group	114
soft_error_rate_confidence Attribute	114
Cell-Level Attributes and Groups	114
soft_error_rate Group	115
Fault Tolerant Library Modeling Example	115
<hr/> 5. Defining Sequential Cells	116

Contents

Copyright and Proprietary Information Notice

Using Sequential Cell Syntax	116
Describing a Flip-Flop	117
Using the ff Group	118
clocked_on and clocked_on_also Attributes	119
next_state Attribute	119
nextstate_type Attribute	119
clear Attribute	120
preset Attribute	120
clear_preset_var1 Attribute	120
clear_preset_var2 Attribute	120
power_down_function Attribute	121
ff Group Examples	122
Describing a Single-Stage Flip-Flop	122
Describing a Master-Slave Flip-Flop	124
Using the function Attribute	125
Describing a Multibit Flip-Flop	126
Describing a Latch	130
latch Group	130
enable and data_in Attributes	131
data_in_type Attribute	131
clear Attribute	131
preset Attribute	132
clear_preset_var1 Attribute	132
clear_preset_var2 Attribute	132
Determining a Latch Cell's Internal State	133
Describing a Multibit Latch	134
latch_bank Group	134
Describing Sequential Cells With the Statetable Format	138
statetable Group	141
Using Shortcuts	143
Partitioning the Cell Into a Model	145
Defining an Output pin Group	145
state_function Attribute	146
internal_node Attribute	146
input_map Attribute	146
inverted_output Attribute	148
Internal Pin Type	148
Determining a Complex Sequential Cell's Internal State	149
Flip-Flop and Latch Examples	150

Contents

Copyright and Proprietary Information Notice

Cell Description Examples	154
<hr/>	
6. Defining Test Cells	157
Describing a Scan Cell	157
test_cell Group	157
Pins in the test_cell Group	158
test_output_only Attribute	161
Describing a Multibit Scan Cell	162
Multibit Scan Cell With Parallel Scan Chain	162
Example	164
Multibit Scan Cell With Internal Scan Chain	167
Examples	170
Scan Cell Modeling Examples	175
Simple Multiplexed D Flip-Flop	175
D Flip-Flop With Gated Output	177
Multibit Cells With Multiplexed D Flip-Flop and Enable	178
LSSD Scan Cell	183
Scan-Enabled LSSD Cell	187
Functional Model of the Scan-Enabled LSSD Cell	187
Timing Model of the Scan-Enabled LSSD Cell	188
Scan-Enabled LSSD Cell Model Example	190
Scan-Enabled LSSD Cell With Asynchronous Inputs	192
Multibit Scan-Enabled LSSD Cell	194
Clocked-Scan Test Cell	196
Scan D Flip-Flop With Auxiliary Clock	198
<hr/>	
7. Timing Arcs	201
Understanding Timing Arcs	202
Combinational Timing Arcs	203
Sequential Timing Arcs	203
Modeling Method Alternatives	204
Defining the timing Group	205
Naming Timing Arcs Using the timing Group	205
Timing Arc Between a Single Pin and a Single Related Pin	206
Timing Arcs Between a Pin and Multiple Related Pins	206
Timing Arcs Between a Bundle and a Single Related Pin	207
Timing Arcs Between a Bundle and Multiple Related Pins	208

Contents

Copyright and Proprietary Information Notice

Timing Arcs Between a Bus and a Single Related Pin	209
Timing Arcs Between a Bus and Multiple Related Pins	210
Timing Arcs Between a Bus and a Related Bus Equivalent	212
Delay Model	213
delay_model Attribute	214
Defining the NLDM Template	214
Creating Lookup Tables	217
Defining the Scalable Polynomial Delay Model Template	218
timing Group Attributes	220
related_pin Simple Attribute	221
related_bus_pins Simple Attribute	222
timing_sense Simple Attribute	222
timing_type Simple Attribute	223
mode Complex Attribute	227
Describing Three-State Timing Arcs	232
Describing Three-State-Disable Timing Arcs	232
Describing Three-State-Enable Timing Arcs	233
Describing Edge-Sensitive Timing Arcs	234
Describing Clock Insertion Delay	235
Describing Intrinsic Delay	236
In the CMOS Piecewise Linear Delay Model	236
In the Scalable Polynomial Delay Model	236
Describing Transition Delay	236
Defining Delay Arcs With Lookup Tables	237
Modeling Transition Time Degradation	241
Modeling Load Dependency	244
In the CMOS Scalable Polynomial Delay Model	246
Describing Slope Sensitivity	247
Describing State-Dependent Delays	247
when Simple Attribute	248
sdf_cond Simple Attribute	250
Setting Setup and Hold Constraints	252
rise_constraint and fall_constraint Groups	254
In the Scalable Polynomial Delay Model	255
Identifying Interdependent Setup and Hold Constraints	256
Setting Nonsequential Timing Constraints	256
Setting Recovery and Removal Timing Constraints	258

Contents

Copyright and Proprietary Information Notice

Recovery Constraints	258
Removal Constraint	260
Setting No-Change Timing Constraints	262
Setting Skew Constraints	265
Setting Conditional Timing Constraints	266
when and sdf_cond Simple Attributes	267
when_start Simple Attribute	267
sdf_cond_start Simple Attribute	267
when_end Simple Attribute	268
sdf_cond_end Simple Attribute	268
sdf_edges Simple Attribute	268
min_pulse_width Group	269
min_pulse_width Example	269
constraint_high and constraint_low Simple Attributes	269
when and sdf_cond Simple Attributes	269
minimum_period Group	269
minimum_period Example	269
constraint Simple Attribute	270
when and sdf_cond Simple Attributes	270
min_pulse_width and minimum_period Example	270
Using Conditional Attributes With No-Change Constraints	272
Impossible Transitions	273
Examples of NLDM Libraries	273
CMOS Piecewise Linear Delay Model	273
Library With Timing Constraints	276
Library With Clock Insertion Delay	279
Describing a Transparent Latch Clock Model	280
Driver Waveform Support	282
Library-Level Tables, Attributes, and Variables	283
normalized_voltage Variable	283
normalized_driver_waveform Group	284
driver_waveform_name Attribute	284
Cell-Level Attributes	284
driver_waveform Attribute	284
driver_waveform_rise and driver_waveform_fall Attributes	284
Pin-Level Attributes	285
driver_waveform Attribute	285
driver_waveform_rise and driver_waveform_fall Attributes	285
Driver Waveform Example	285

Contents

Copyright and Proprietary Information Notice

Sensitization Support	287
sensitization Group	287
pin_names Attribute	288
vector Attribute	288
Cell-Level Attributes	288
sensitization_master Attribute	289
pin_name_map Attribute	289
timing Group Attributes	289
sensitization_master Attribute	289
pin_name_map Attribute	289
wave_rise and wave_fall Attributes	290
wave_rise_sampling_index and wave_fall_sampling_index Attributes ..	291
wave_rise_time_interval and wave_fall_time_interval Attributes	291
timing Group Syntax	292
NAND Cell Example	293
Complex Macro Cell Example	294
Phase-Locked Loop Support	296
Phase-Locked Loop Syntax	297
Cell-Level Attribute	297
is_pll_cell Attribute	297
Pin-Level Attributes	297
is_pll_reference_pin Attribute	298
is_pll_feedback_pin Attribute	298
is_pll_output_pin Attribute	298
Phase-Locked Loop Example	298
<hr/>	
8. Modeling Power and Electromigration	300
Modeling Power Terminology	300
Static Power	300
Dynamic Power	301
Internal Power	301
Switching Power	302
Switching Activity	303
Modeling for Leakage Power	303
Representing Leakage Power Information	304
cell_leakage_power Simple Attribute	304
Using the leakage_power Group for a Single Value	304
when Simple Attribute	305
value Simple Attribute	306

Contents

Copyright and Proprietary Information Notice

leakage_power_unit Simple Attribute	306
default_cell_leakage_power Simple Attribute	307
Example	307
Threshold Voltage Modeling	310
Modeling for Internal and Switching Power	312
Modeling Internal Power Lookup Tables	313
Representing Internal Power Information	315
Specifying the Power Model	315
Using Lookup Table Templates	316
power_lut_template Group	316
Scalar power_lut_template Group	318
Defining Internal Power Groups	318
Naming Power Relationships, Using the internal_power Group	318
Power Relationship Between a Single Pin and a Single Related Pin	319
Power Relationships Between a Single Pin and Multiple Related Pins	319
Power Relationships Between a Bundle and a Single Related Pin	320
Power Relationships Between a Bundle and Multiple Related Pins	321
Power Relationships Between a Bus and a Single Related Pin	322
Power Relationships Between a Bus and Multiple Related Pins	323
Power Relationships Between a Bus and Related Bus Pins	325
internal_power Group	326
equal_or_opposite_output Simple Attribute	326
falling_together_group Simple Attribute	327
related_pin Simple Attribute	327
rising_together_group Simple Attribute	328
switching_interval Simple Attribute	329
switching_together_group Simple Attribute	329
when Simple Attribute	330
fall_power Group	330
power Group	332
rise_power Group	332
Internal Power Examples	335
One-Dimensional Power Lookup Table	336
Two-Dimensional Power Lookup Table	337
Three-Dimensional Power Lookup Table	338
Modeling Libraries With Integrated Clock-Gating Cells	340
What Clock Gating Does	340
Looking at a Gated Clock	341
Using an Integrated Clock-Gating Cell	342
Setting Pin Attributes for an Integrated Cell	343

Contents

Copyright and Proprietary Information Notice

clock_gate_clock_pin Attribute	343
clock_gate_enable_pin Simple Attribute	344
clock_gate_test_pin Attribute	344
clock_gate_obs_pin Attribute	344
clock_gate_out_pin Attribute	345
Clock-Gating Timing Considerations	345
Timing Considerations for Integrated Cells	347
Integrated Clock-Gating Cell Example	348
Modeling Electromigration	350
Controlling Electromigration	350
em_lut_template Group	351
electromigration Group	353
related_pin Simple Attribute	354
related_bus_pins Simple Attribute	354
lifetime_profile Simple Attribute	354
em_max_toggle_rate Group	355
em_temp_degradation_factor Simple Attribute	357
Example	357
Cell Electromigration Example	357
<hr/>	
9. Advanced Low-Power Modeling	360
Power and Ground (PG) Pins	361
Example Libraries With Multiple Power Supplies	361
Partial PG Pin Cell Modeling	364
Special Partial PG Pin Cells	365
Supported Attributes	366
Partial PG Pin Cell Example	367
PG Pin Syntax	368
Library-Level Attributes	369
voltage_map Complex Attribute	369
default_operating_conditions Simple Attribute	369
Cell-Level Attributes	369
pg_pin Group	370
is_pad Simple Attribute	370
voltage_name Simple Attribute	370
pg_type Simple Attribute	370
physical_connection Simple Attribute	371
related_bias_pin Attribute	371
user_pg_type Simple Attribute	372
Pin-Level Attributes	372
direction Attribute	372

Contents

Copyright and Proprietary Information Notice

power_down_function Attribute	372
related_power_pin and related_ground_pin Attributes	373
output_signal_level_low and output_signal_level_high Attributes	373
input_signal_level_low and input_signal_level_high Attributes	373
related_pg_pin Attribute	373
Timing-Level Attributes	373
output_signal_level_low and output_signal_level_high Attributes	374
Standard Cell With One Power and Ground Pin Example	375
Inverter With Substrate-Bias Pins Example	377
Insulated Bias Modeling	379
Same PG Pin as Both Power Pin and Ground Pin	383
PG Pin Power Mode Modeling	385
Syntax	386
Library-Level Groups and Attributes	387
voltage_state_range_list Group	387
voltage_state_range Attribute	387
Cell-Level Groups and Attributes	388
pg_setting_definition Group	388
pg_setting_value Group	389
default_pg_setting Attribute	391
pg_setting_transition Group	391
illegal_transition_if_undefined Attribute	392
Specifying Power States in Timing, Power, and Noise Models	392
pg_setting Attribute in mode_value Group	393
mode Attribute in minimum_period and min_pulse_width Groups	393
Defining PG Modes in Macro Cells	393
PG Mode Examples	394
Cell With Different Power States	394
Power State Transition	395
Macro Cell With PG Modes	397
Retention Cell Leakage Power in Different Power Modes	400
Feedthrough Signal Pin Modeling	406
Multipin Feedthroughs	406
Single-Pin Feedthrough	407
Overdrive and Underdrive Voltage Modeling	408
Internally Unconnected Pin Modeling	411
Silicon-on-Insulator (SOI) Cell Modeling	412
Library-Level Attribute	414
is_soi Attribute	414
Cell-Level Attribute	414

Contents

Copyright and Proprietary Information Notice

is_soi Attribute	414
Level-Shifter Cells in a Multivoltage Design	415
Operating Voltages	415
Level Shifter Functionality	415
Basic Level-Shifter Syntax	416
Cell-Level Attributes	416
is_level_shifter Attribute	417
level_shifter_type Attribute	417
input_voltage_range Attribute	417
output_voltage_range Attribute	418
ground_input_voltage_range Attribute	418
ground_output_voltage_range Attribute	418
Pin-Level Attributes	418
std_cell_main_rail Attribute	419
level_shifter_data_pin Attribute	419
level_shifter_enable_pin Attribute	419
input_voltage_range and output_voltage_range Attributes	419
ground_input_voltage_range Attribute	419
ground_output_voltage_range Attribute	419
input_signal_level Attribute	420
power_down_function Attribute	420
alive_during_power_up Attribute	420
Enable Level-Shifter Cell	420
Differential Level-Shifter Cell	420
Clamping Enable Level-Shifter Cell Outputs	423
Pin-Level Attributes	423
Level Shifter Modeling Examples	424
Level-Shifter Cell Without Enable Pin	424
Simple Buffer Type Low-to-High Level Shifter	425
Simple Buffer Type High-to-Low Level Shifter	428
Power-and-Ground Level-Shifter Cell	430
Multibit Level-Shifter Cell	432
Overdrive Level-Shifter Cell	435
Level-Shifter Cell With Virtual Bias Pins	437
Simple Enable Level-Shifter Cell	438
Enable Level-Shifter With Isolation	440
Clamping in Latch-Based Enable Level-Shifter Cell	442
Differential Level-Shifter Cell	443
Isolation Cell Modeling	446
Cell-Level Attribute	447
is_isolation_cell Attribute	447
Pin-Level Attributes	447

Contents

Copyright and Proprietary Information Notice

isolation_cell_data_pin Attribute	447
isolation_cell_enable_pin Attribute	447
power_down_function Attribute	447
permit_power_down Attribute	447
alive_during_partial_power_down Attribute	448
alive_during_power_up Attribute	448
Attribute Dependencies	448
Isolation Cell Examples	448
Simple NAND-Type Isolation Cell	448
Simple NOR-Type Isolation Cell	450
NOR-Type Isolation Cell With Inverted Input	451
NOR-Type Isolation Cell With Inverted Output	453
Isolation Cell Without Enable Pin	454
Multibit Isolation Cell	455
Clock-Isolation Cell Modeling	458
Cell-Level Attribute	459
Pin-Level Attributes	459
Clock Isolation Cell Examples	460
Clamping Isolation Cell Output Pins	462
Pin-Level Attributes	463
Example of Clamping in Isolation Cell	463
Isolation Cells With Multiple Control Pins	464
Switch Cell Modeling	467
Coarse-Grain Switch Cells	467
Coarse-Grain Switch Cell Syntax	468
Library-Level Group	470
Cell-Level Attribute and Group	470
Pin-Level Attributes	471
pg_pin Group	472
Fine-Grained Switch Support for Macro Cells	472
Macro Cell With Fine-Grained Switch Syntax	472
Cell-Level Attributes	473
pg_pin Group	473
Switch-Cell Modeling Examples	474
Simple Coarse-Grain Header Switch Cell	474
Complex Coarse-Grain Header Switch Cell	476
Complex Coarse-Grain Switch Cell With an Internal Switch Pin	478
Complex Coarse-Grain Switch Cell With Parallel Switches	480
Retention Cell Modeling	483
Modes of Operation	485
Retention Cell Modeling Syntax	485
Cell-Level Attributes, Groups, and Variables	488

Contents

Copyright and Proprietary Information Notice

retention_cell Simple Attribute	488
ff, latch, ff_bank, and latch_bank Groups	488
retention_condition Group	488
clock_condition Group	489
preset_condition and clear_condition Groups	489
reference_pin_names Variable	490
variable1 and variable2 Variables	491
bits Variable	491
Pin-Level Attributes	491
retention_pin Complex Attribute	491
function Attribute	491
reference_input Attribute	492
save_action and restore_action Attributes	492
restore_edge_type Attribute	492
save_condition and restore_condition Attributes	493
Retention Cell Model Examples	493
Retention Cell With Balloon Latch	493
Retention Cell With Multiple latch Groups	497
Retention Cell With Multiple latch_bank Groups	501
Retention Cell With Edge-Triggered Balloon Logic	504
MUX-Scan Retention Cell	508
Always-On Cell Modeling	511
Always-On Cell Syntax	512
always_on Simple Attribute	513
Always-On Simple Buffer Example	513
Macro Cell Modeling	514
Macro Cell Isolation Modeling	515
Pin-Level Attributes	517
Modeling Macro Cells With Internal PG Pins	518
Macro Cell With Built-in Multiplexer	520
Macro Cell With Fine-Grained Internal Power Switches	524
Macro Cell With an Always-On Pin Example	525
Modeling Antenna Diodes	527
Antenna-Diode Cell Modeling	527
Cell-Level Attribute	528
Pin-Level Attributes	529
Antenna-Diode Cell Modeling Example	529
Modeling Cells With Built-In Antenna-Diode Ports	530
Pin-Level Attributes	530
Antenna-Diode Pin Modeling Example	531

10. Composite Current Source Modeling	532
Modeling Cells With Composite Current Source Information	532
Representing Composite Current Source Driver Information	532
Composite Current Source Lookup Tables	533
Defining the <code>output_current_template</code> Group	533
<code>output_current_template</code> Syntax	533
Template Variables for CCS Driver Models	533
<code>output_current_template</code> Example	533
Defining the Lookup Table Output Current Groups	534
<code>output_current_rise</code> Syntax	534
vector Group	534
<code>reference_time</code> Simple Attribute	534
Template Variables for CCS Driver Models	534
vector Group Example	535
Representing Composite Current Source Receiver Information	535
Comparison Between Two-Segment and Multisegment Receiver Models	536
Two-Segment Receiver Capacitance Model	537
Syntax	537
Defining the <code>receiver_capacitance</code> Group at the Pin Level	538
Defining the <code>lu_table_template</code> Group	539
Conditional Data Modeling	540
Examples	540
Defining the Receiver Capacitance Groups at the Timing Level	545
Conditional Data Modeling	545
Defining the <code>lu_table_template</code> Group	545
Timing-Level <code>receiver_capacitance</code> Example	546
Multisegment Receiver Capacitance Model	547
Library-Level Groups and Attributes	548
<code>lu_table_template</code> Group	548
<code>receiver_capacitance_rise_threshold_pct</code> and <code>receiver_capacitance_fall_threshold_pct</code> Attributes	548
Pin and Timing Level Groups	549
<code>receiver_capacitance</code> Group	549
Conditional Data Modeling	550
Example	550
CCS Retain Arc Support	552
CCS Retain Arc Syntax	553
<code>ccs_retain_rise</code> and <code>ccs_retain_fall</code> Groups	554
vector Group	554

Contents

Copyright and Proprietary Information Notice

reference_time Attribute	554
Compact CCS Retain Arc Syntax	554
compact_ccs_retain_rise and compact_ccs_retain_fall Groups	555
base_curves_group Attribute	555
index_1, index_2, and index_3 Attributes	556
values Attribute	556
Composite Current Source Driver and Receiver Model Example	556
11. Advanced Composite Current Source Modeling	560
Modeling Cells With Advanced Composite Current Source Information	560
Compact CCS Timing Model	560
Modeling With CCS Timing Base Curves	561
Compact CCS Timing Model Syntax	563
base_curves Group	564
compact_lut_template Group	565
compact_ccs_{rise fall} Group	566
Compact CCS Timing Library Example	566
Variation-Aware Timing Modeling Support	567
Variation-Aware Compact CCS Timing Driver Model	568
timing_based_variation Group	569
va_compact_ccs_rise and va_compact_ccs_fall Groups	570
timing_based_variation and pin_based_variation Groups	571
Variation-Aware CCS Timing Receiver Model	572
timing_based_variation and pin_based_variation Groups	574
va_parameters Complex Attribute	574
nominal_va_values Complex Attribute	574
va_receiver_capacitance1_rise, va_receiver_capacitance1_fall, va_receiver_capacitance2_rise, and va_receiver_capacitance2_fall Groups	574
va_values Attribute	575
Variation-Aware Timing Constraint Modeling	575
timing_based_variation Group	576
va_parameters Complex Attribute	576
nominal_va_values Complex Attribute	576
va_rise_constraint and va_fall_constraint Groups	576
va_values Attribute	577
Conditional Data Modeling for Variation-Aware Timing Receiver Models	577
when Attribute	578
mode Attribute	578
Variation-Aware Compact CCS Retain Arcs	583

Contents

Copyright and Proprietary Information Notice

va_compact_ccs_retain_rise and va_compact_ccs_retain_fall Groups	584
va_values Attribute	584
values Attribute	585
Variation-Aware Syntax Examples	585
va_parameters in Advanced CCS Modeling Usage	585
va_compact_ccs_rise and va_compact_ccs_fall Groups	586
va_values With Three Variation Parameters	587
peak_voltage in Values Attribute	588
pin_based_variation Group	589
Pin-based Model With nominal CCS receiver model and Variation-Aware CCS Receiver Model Groups	590
nominal_va_values in Advanced CCS Modeling Usage	591
Variational Values in Advanced CCS Modeling Usage	591
Variation-Aware CCS Driver or Receiver with Timing Constraints	592
<hr/>	
12. Nonlinear Signal Integrity Modeling	598
Modeling Noise Terminology	598
Noise Calculation	599
Noise Immunity	599
Noise Propagation	599
Modeling Cells for Noise	599
I-V Characteristics and Drive Resistance	600
Noise Immunity	602
Using the Hyperbolic Model	605
Noise Propagation	605
Representing Noise Calculation Information	606
I-V Characteristics Lookup Table Model	607
iv_lut_template Group	607
Defining the Lookup Table Steady-State Current Groups	608
I-V Characteristics Curve Polynomial Model	609
poly_template Group	609
Defining Polynomial Steady-State Current Groups	610
Using Steady-State Resistance Simple Attributes	611
Using I-V Curves and Steady-State Resistance for tied_off Cells	612
Defining tied_off Attribute Usage	612
Representing Noise Immunity Information	613
Noise Immunity Lookup Table Model	613
noise_lut_template Group	613
Defining the Noise Immunity Table Groups	614

Contents

Copyright and Proprietary Information Notice

Noise Immunity Polynomial Model	616
poly_template Group	616
Defining the Noise Immunity Polynomial Groups	617
Input Noise Width Ranges at the Pin Level	618
Defining the input_noise_width Range Limits	618
Defining the Hyperbolic Noise Groups	620
Representing Propagated Noise Information	621
Propagated Noise Lookup Table Model	621
propagation_lut_template Group	621
Defining the Propagated Noise Table Groups	623
Propagated Noise Polynomial Model	624
poly_template Group	625
Defining Propagated Noise Groups for Polynomial Representation	626
Examples of Modeling Noise	628
Scalable Polynomial Model Noise Example	628
Nonlinear Delay Model Library With Noise Information	634
13. Composite Current Source Signal Integrity Modeling	639
CCS Signal Integrity Modeling Overview	639
CCS Signal Integrity Modeling Syntax	639
Library-Level Groups and Attributes	644
lu_table_template Group	644
variable_1, variable_2, variable_3, and variable_4 Attributes	644
Pin-Level Groups and Attributes	645
ccsn_first_stage and ccsn_last_stage Groups	645
is_needed Attribute	646
is_inverting Attribute	646
stage_type Attribute	646
miller_cap_rise and miller_cap_fall Attributes	646
output_signal_level and input_signal_level Attributes	646
related_ccb_node Attribute	647
dc_current Group	647
output_voltage_rise and output_voltage_fall Groups	647
propagated_noise_high and propagated_noise_low Groups	647
when Attribute	648
CCS Noise Library Examples	648
Conditional Data Modeling in CCS Noise Models	652
when Attribute	653
mode Attribute	653

Contents

Copyright and Proprietary Information Notice

CCS Noise Modeling for Unbuffered Cells With a Pass Gate	655
Syntax for Unbuffered Output Latches	656
Pin-Level Attributes	657
is_unbuffered Attribute	657
has_pass_gate Attribute	657
ccsn_first_stage Group	657
is_pass_gate Attribute	658
CCS Noise Modeling for Multivoltage Designs	658
Referenced CCS Noise Modeling	661
Modeling Syntax	661
Cell-Level Attributes	663
driver_waveform Attribute	663
driver_waveform_rise and driver_waveform_fall Attributes	663
Pin-Level Groups	663
input_ccb and output_ccb Groups	663
Timing-Level Attributes	664
active_input_ccb Attribute	665
active_output_ccb Attribute	665
propagating_ccb Attribute	665
Examples	665
<hr/>	
14. Composite Current Source Power Modeling	669
Composite Current Source Power Modeling	669
Cell Leakage Current	670
Gate Leakage Modeling in Leakage Current	671
gate_leakage Group	671
input_low_value Attribute	672
input_high_value Attribute	672
Intrinsic Parasitic Models	672
Voltage-Dependent Intrinsic Parasitic Models	674
Library-Level Group	677
lu_table_template Group	677
Pin-Level Group	677
lut_values Group	677
Parasitics Modeling in Macro Cells	677
total_capacitance Group	677
Parasitics Modeling Syntax	678
Dynamic Power	678
Dynamic Power and Ground Current Table Syntax	679
Dynamic Power Modeling in Macro Cells	679

Contents

Copyright and Proprietary Information Notice

Examples for CCS Dynamic Power for Macro Cells	681
Conditional Data Modeling for Dynamic Current Example	682
Dynamic Current Syntax	684
Compact CCS Power Modeling	685
Compact CCS Power Syntax and Requirements	687
Library-Level Groups and Attributes	688
base_curves Group	688
compact_lut_template Group	689
Cell-Level Groups and Attributes	690
compact_ccs_power Group	690
Composite Current Source Dynamic Power Examples	691
Design Cell With a Single Output Example	691
Dense Table With Two Output Pins Example	692
Cross Type With More Than One Output Pin Example	692
Diagonal Type With More Than One Output Pin Example	693
<hr/>	
15. On-Chip Variation (OCV) Modeling	695
Advanced OCV Modeling	695
Library-Level Groups and Attributes	697
ocv_table_template Group	697
ocv_derate Group	697
default_ocv_derate_group Attribute	699
ocv_arc_depth Attribute	699
Cell-Level Attribute	699
ocv_derate_group Attribute	699
Advanced OCV Library Example	699
LVF Models For Cell Delay, Transition, and Constraint	701
Syntax	701
Library-Level Groups and Attributes	704
lu_table_template Group	704
ocv_table_template Group	706
ocv_derate Group	706
default_ocv_derate_distance_group Attribute	707
Cell-Level Attribute	707
ocv_derate_distance_group Attribute	707
Timing Arc Level Groups and Attributes	707
ocv_sigma_cell_rise Group	708
ocv_sigma_cell_fall Group	708
ocv_sigma_rise_transition Group	708

Contents

Copyright and Proprietary Information Notice

ocv_sigma_fall_transition Group	709
sigma_type Attribute	709
ocv_sigma_rise_constraint Group	709
ocv_sigma_fall_constraint Group	710
Parametric OCV Library Example	710
LVF Retain Arc Models	713
Syntax	713
Library-Level Groups	715
lu_table_template Group	715
Timing Arc Level Groups	715
ocv_sigma_retaining_rise and ocv_sigma_retaining_fall Groups	715
ocv_sigma_retain_rise_slew and ocv_sigma_retain_fall_slew Groups	716
sigma_type Attribute	716
Library Example	716
LVF Moment-Based Models For Ultra-Low Voltage Designs	718
Syntax	720
Library-Level Groups	724
lu_table_template Group	724
Timing Arc Level Groups	724
ocv_std_dev_* Groups	724
ocv_mean_shift_* Groups	725
ocv_skewness_* Groups	725
Library Example	726
<hr/>	
16. Defining I/O Pads	730
Special Characteristics of I/O Pads	730
Identifying Pad Cells	731
is_pad Attribute	731
driver_type Attribute	731
Defining Units for Pad Cells	734
Capacitance	734
Resistance	735
Voltage	735
Current	735
Describing Input Pads	735
hysteresis Attribute	735
Describing Output Pads	736

Contents

Copyright and Proprietary Information Notice

Drive Current	736
Slew-Rate Control	736
Modeling Wire Load for Pads	738
Programmable Driver Type Support in I/O Pad Cell Models	739
Syntax	739
Programmable Driver Type Functions	739
Example	740
Pad Cell Examples	742
Input Pads	743
Output Pads	746
Bidirectional Pad	747
Cell with contention_condition and x_function	748
A. Library Characterization Configuration	751
The char_config Group	751
Library Characterization Configuration Syntax	751
Common Characterization Attributes	754
driver_waveform Attribute	757
driver_waveform_rise and driver_waveform_fall Attributes	758
input_stimulus_transition Attribute	758
input_stimulus_interval Attribute	758
unrelated_output_net_capacitance Attribute	758
default_value_selection_method Attribute	759
default_value_selection_method_rise and default_value_selection_method_fall Attributes	759
merge_tolerance_abs and merge_tolerance_rel Attributes	760
merge_selection Attribute	760
NLPM Characterization Attributes	761
CCS Timing Characterization Attributes	762
ccs_timing_segment_voltage_tolerance_rel Attribute	763
ccs_timing_delay_tolerance_rel Attribute	763
ccs_timing_voltage_margin_tolerance_rel Attribute	763
CCS Receiver Capacitance Attributes	763
Input-Capacitance Characterization Attributes	764
capacitance_voltage_lower_threshold_pct_rise and capacitance_voltage_lower_threshold_pct_fall Attributes	764

Contents

Copyright and Proprietary Information Notice

capacitance_voltage_upper_threshold_pct_rise and capacitance_voltage_upper_threshold_pct_fall Attributes	764
---	-----

1

Sample Library Description

This chapter familiarizes you with the basic format and syntax of a library description. The chapter starts with an example that shows the general syntax of a library. The structure of the library description is also discussed. These topics are covered in the following sections:

This chapter includes the following sections:

- [General Syntax](#)
- [Statements](#)
- [Reducing Library File Size](#)

General Syntax

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

Example 1 General Syntax of the Library Description

```
library (name) {
    technology (name) ; /* library-level attributes */
    delay_model : generic_cmos | table_lookup | cmos2 |
                    piecewise_cmos | dcm | polynomial ;
    bus_naming_style : string;
    routing_layers (string) ;
    time_unit : unit ;
    voltage_unit : unit ;
    current_unit : unit ;
    pulling_resistance_unit : unit ;
    capacitive_load_unit (value, unit) ;
    leakage_power_unit : unit ;
    piece_type : type ;
    piece_define ("list") ;
    define_cell_area (area_name, resource_type) ;

/* default values  for environment definitions */
```

Chapter 1: Sample Library Description Statements

```
operating_conditions (name) {
    /* operating conditions */
}
timing_range (name) {
    /* timing information */
}
wire_load (name) {
    /* wire load information */
}
wire_load_selection () {
    /* area/group selections */
}
power_lut_template (name) {
    /* power lookup table template information */
}
cell (name1) { /* cell definitions */
    /* cell information */
}
cell (name2) {
    /* cell information */
}
scaled_cell (name1) {
    /* alternate scaled cell information */
}
...
type (name) {
    /* bus type name */
}
input_voltage (name) {
    /* input voltage information */
}
output_voltage (name) {
    /* output voltage information */
}
}
```

Statements

Statements are the building blocks of a library. All library information is described in one of the following types of statements:

- Group statements
- Attribute statements
- Define statements

A statement can continue across multiple lines. A continued line ends with a backslash (\).

Group Statements

A group is a named collection of statements that defines a library, a cell, a pin, a timing arc, and so forth. Braces ({}), which are used in pairs, enclose the contents of the group.

Syntax

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

name

A string that identifies the group. Check the individual group statement syntax definition to verify if *name* is required, optional, or null.

You must type the group name and the { symbol on the same line.

[Example 2](#) defines a `pin` group A.

Example 2 Group Statement Specification

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

Attribute Statements

An attribute statement defines characteristics of specific objects in the library. Attributes applying to specific objects are assigned within a group statement defining the object, such as a `cell` group or a `pin` group.

In this user guide, attribute refers to all attributes unless the manual specifically states otherwise. For clarity, this manual distinguishes different types of attribute statements according to syntax. All simple attributes use the same general syntax; complex attributes have different syntactic requirements.

Simple Attributes

This is the syntax of a simple attribute:

```
attribute_name : attribute_value ;
```

You must separate the attribute name from the attribute value with a space, followed by a colon and another space. Place attribute statements on a single line.

[Example 3](#) adds a `direction` attribute to `pin` group A shown in [Example 2](#).

Example 3 Simple Attribute Specification

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

For some simple attributes, you must enclose the attribute value in quotation marks:

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

Example 4 adds the X + Y function to the pin example.

Example 4 Defining the Function of a Pin

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

Complex Attributes

This is the syntax of a complex attribute statement. Include one or more parameters in parentheses.

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

The following example uses the line complex attribute to define a line on a schematic symbol. This line is drawn from coordinates (1,2) to coordinates (6,8):

```
line (1, 2, 6, 8);
```

Define Statements

You can create new simple attributes with the define statement.

Syntax

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

attribute_name

The name of the new attribute you are creating.

group_name

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

attribute_type

The type of attribute you are creating: Boolean, string, integer, or floating point.

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

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

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

```
bork : "nimo" ;
```

Reducing Library File Size

Large library files can compromise disk capacity and memory resources. To reduce file size and improve file management, the syntax allows you to combine multiple source files by referencing the files from within the source file containing the `library` group description. During library compilation, the referenced information is retrieved, included at the point of reference, and then the compilation continues.

Use the `include_file` attribute to reference information in another file for inclusion during library compilation. Be sure the directory of the included file is defined in your search path—the `include_file` attribute takes only the file name as its value; no path is allowed.

Syntax

```
include_file (file_name_id) ;
```

Example

```
cell( ) {  
    area : 0.1 ;  
    ...  
    include_file (memory_file) ;  
    ...  
}
```

where `memory_file` contains the `memory` group statements.

Limitations

The `include_file` attribute has these requirements:

- Recursive `include_file` statements are not allowed; that is, the source files that you include cannot also contain `include_file` statements.
- If the included file is not in the current directory, then the location of the included file must be defined in your search path.
- Multiple file names are not allowed in an `include_file` statement. However, there is no limit to the number of `include_file` statements you can have in your main source file.

Chapter 1: Sample Library Description Statements

- An included file cannot substitute for a group value statement. For example, the following is not allowed:

```
cell ( ) {  
    area : 0.1 ;  
    ...  
    pin_equal : include_file (source_file) ;  
}
```

- The `include_file` attribute cannot substitute or cross the group boundary. For example, the following is not allowed:

```
cell ( A ) include (source_file)
```

where `source_file` is the following:

```
{  
    attribute : value ;  
    attribute : value ;  
    ...  
}
```

2

Building a Logic Library

A library description identifies the characteristics of a logic library and the cells it contains. The `library` group contains the entire library description. This chapter describes the `library` group-level attributes of a CMOS logic library.

The following concepts and tasks are explained in this chapter:

- [Creating Library Groups](#)
- [Using General Library Attributes](#)
- [Delay and Slew Attributes](#)
- [Defining Units](#)
- [Setting Minimum Cell Requirements](#)

Creating Library Groups

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

library Group

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

Example

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

Using General Library Attributes

These attributes apply generally to the logic library:

- technology
 - delay_model
 - bus_naming_style
-

technology Attribute

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

Syntax

```
technology (name) ;
```

Example

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

delay_model Attribute

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

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

Syntax

```
delay_model : value ;
```

Example

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

bus_naming_style Attribute

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

Syntax

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

string

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

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

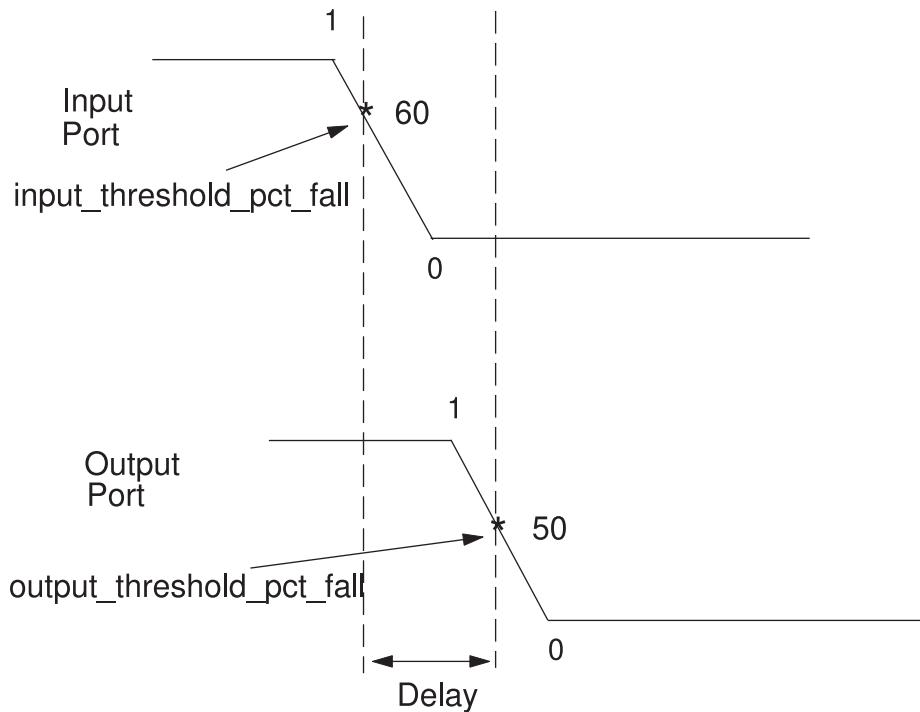
Delay and Slew Attributes

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

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

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

Figure 1 Delay Modeling for Falling Signal



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

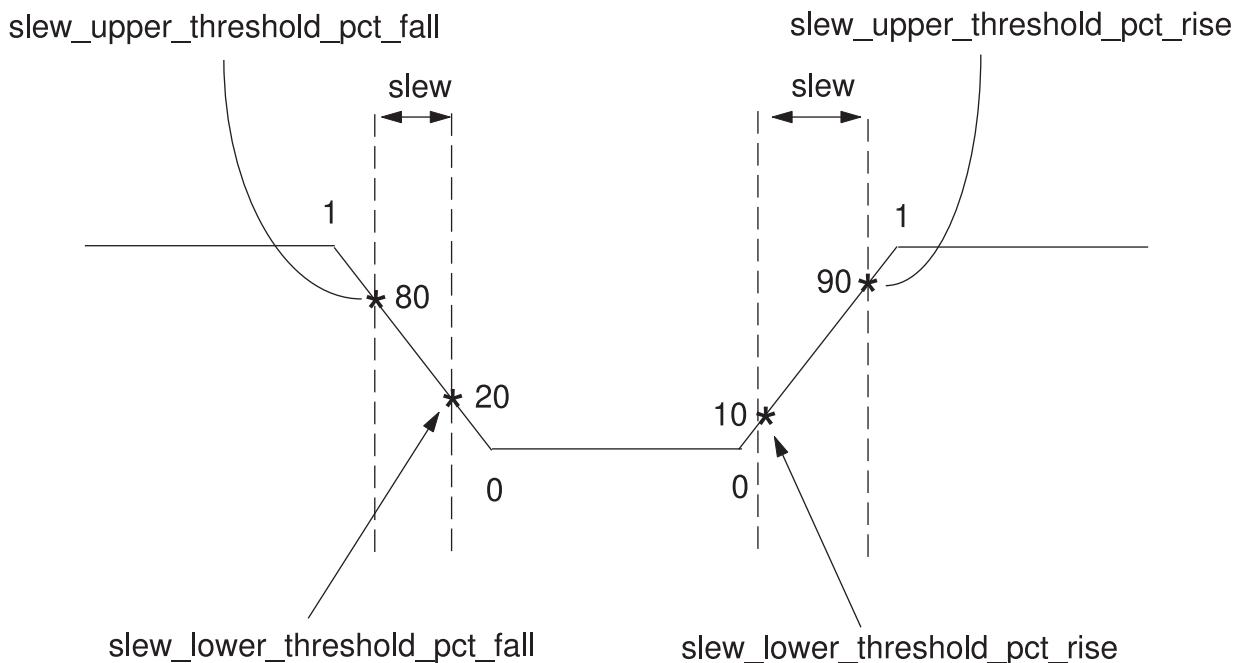
Use the following attributes to enter the two designated threshold points to model the time for voltage falling from 1 to 0:

- `slew_lower_threshold_pct_fall`
- `slew_upper_threshold_pct_fall`

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

- `slew_lower_threshold_pct_rise`
- `slew_upper_threshold_pct_rise`

Figure 2 Slew Modeling Example



Specifying Delay and Slew Attributes in Voltage Range

When partial voltage swing is defined in a pin or a timing group, the nonlinear delay model (NLDM) data in that group is for the partial swing. You must apply the following threshold and trip point attributes only in the voltage range specified using the `output_signal_level_low` and the `output_signal_level_high` attributes. For more information, see [input_signal_level_low and input_signal_level_high Attributes on page 373](#)."

input_threshold_pct_fall Simple Attribute

Note:

To model the delay of a signal going from an input pin to an output pin, you also need to set the value of the output pin. See [output_threshold_pct_fall Simple Attribute on page 38](#) and [output_threshold_pct_rise Simple Attribute on page 39](#) for details.

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

Syntax

```
input_threshold_pct_fall : trip_pointvalue ;
```

trip_point

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

Example

```
input_threshold_pct_fall : 60.0 ;
```

input_threshold_pct_rise Simple Attribute

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

Note:

To model the delay of a signal going from an input pin to an output pin, you also need to set the value of the output pin. See “[output_threshold_pct_fall Simple Attribute](#)” and [output_threshold_pct_rise Simple Attribute on page 39](#) for details.

Syntax

```
input_threshold_pct_rise : trip_pointvalue ;
```

trip_point

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

Example

```
input_threshold_pct_rise : 40.0 ;
```

output_threshold_pct_fall Simple Attribute

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

Note:

To model the delay of a signal going from an input pin to an output pin, you also need to set the value of the input pin. See “[input_threshold_pct_rise Simple](#)

[Attribute](#)” and [input_threshold_pct_fall Simple Attribute on page 37](#) for details.

Syntax

```
output_threshold_pct_fall : trip_pointvalue ;
```

trip_point

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

Example

```
output_threshold_pct_fall : 40.0 ;
```

output_threshold_pct_rise Simple Attribute

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

Note:

To model the delay of a signal going from an input pin to an output pin, you also need to set the value of the input pin. See [input_threshold_pct_rise Simple Attribute on page 38](#) and [input_threshold_pct_fall Simple Attribute on page 37](#) for details.

Syntax

```
output_threshold_pct_rise : trip_pointvalue ;
```

trip_point

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

Example

```
output_threshold_pct_rise : 40.0 ;
```

slew_derate_from_library Simple Attribute

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

Syntax

```
slew_derate_from_library : deratevalue ;
```

derate

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

Example

```
slew_derate_from_library : 0.5 ;
```

slew_lower_threshold_pct_fall Simple Attribute

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

Note:

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

Syntax

```
slew_lower_threshold_pct_fall : trip_point_value ;
```

trip_point

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

Example

```
slew_lower_threshold_pct_fall : 30.0 ;
```

slew_lower_threshold_pct_rise Simple Attribute

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

Note:

To model the delay of a pin rising from 0 to 1, you also need to set the value for the upper threshold point. See [slew_upper_threshold_pct_fall Simple Attribute](#) on page 41 for details.

Syntax

```
slew_lower_threshold_pct_rise : trip_point_value ;
```

trip_point

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

Example

```
slew_lower_threshold_pct_rise : 30.0 ;
```

slew_upper_threshold_pct_fall Simple Attribute

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

Note:

To model the delay of a pin falling from 1 to 0, you also need to set the value for the lower threshold point. See [slew_lower_threshold_pct_fall Simple Attribute on page 40](#) for details.

Syntax

```
slew_upper_threshold_pct_fall : trip_point_value ;
```

trip_point

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

Example

```
slew_upper_threshold_pct_fall : 70.0 ;
```

slew_upper_threshold_pct_rise Simple Attribute

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

Note:

To model the delay of a pin rising from 0 to 1, you also need to set the value for the lower threshold point. See [slew_lower_threshold_pct_rise Simple Attribute on page 40](#) for details.

Syntax

```
slew_upper_threshold_pct_rise : trip_pointvalue ;
```

trip_point

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

Example

```
slew_upper_threshold_pct_rise : 70.0 ;
```

Defining Units

Use these library-level attributes to define units:

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

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

time_unit Attribute

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

Syntax

```
time_unit : unit ;
```

unit

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

Example

```
time_unit : "10ps";
```

voltage_unit Attribute

Use this attribute to scale the contents of the `input_voltage` and `output_voltage` groups. Additionally, the `voltage` attribute in the `operating_conditions` group represents values in the voltage units.

Syntax

```
voltage_unit : unit ;
```

unit

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

Example

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

current_unit Attribute

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

Syntax

```
current_unit : valueenum ;
```

value

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

Example

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

pulling_resistance_unit Attribute

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

Syntax

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

unit

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

Example

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

capacitive_load_unit Attribute

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

Syntax

```
capacitive_load_unit (valuefloat,unitenum) ;
```

value

A floating-point number.

unit

Valid values are ff and pf.

Example

```
capacitive_load_unit(1,pf);
```

leakage_power_unit Attribute

This attribute indicates the units of the power values in the library. If this attribute is missing, the leakage-power values are expressed without units.

Syntax

```
leakage_power_unit : valueenum ;
```

value

Valid values are 1mW, 100mW, 10mW, 1mW, 100nW, 10nW, 1nW, 100pW, 10pW, and 1pW.

Example

```
leakage_power_unit : 100uW;
```

Setting Minimum Cell Requirements

For the Design Compiler tool to perform technology mapping and optimization on design descriptions, the logic library should contain a minimum set of cells.

This is a minimum set of cells for a CMOS logic library:

- An inverter
- A 2-input NAND gate
- A 2-input NOR gate
- A three-state buffer
- A D flip-flop with preset, clear, and complementary output values
- A D latch with preset, clear, and complementary output values

The Design Compiler tool writes out a warning message if the design contains a cell that is not included in the library or libraries associated with the design.

3

Building Environments

Variations in operating temperature, supply voltage, and manufacturing process cause performance variations in electronic networks.

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](#)
 - [Specifying Delay Scaling Attributes](#)
-

Library-Level Default Attributes

Global defaults are set at the library level. You can override many of these defaults.

Setting Default Cell Attributes

The following attributes are defaults that apply to all cells in a library.

default_cell_leakage_power Simple Attribute

Indicates the default leakage power for those cells that do not have the `cell_leakage_power` attribute. This attribute must be a nonnegative floating-point number. If it is not defined, this attribute defaults to 0.0.

Example

```
default_cell_leakage_power : 0.5;
```

Setting Default Pin Attributes

Default pin attributes apply to all pins in a library and deal with timing. How you define default timing attributes in your library depends on the timing delay model you use.

These are the defaults that apply to all pins in a library.

```
default_inout_pin_cap : valuefloat ;
```

Sets a default for capacitance for all I/O pins in the library.

```
default_input_pin_cap : valuefloat ;
```

Sets a default for capacitance for all input pins in the library.

```
default_output_pin_cap : valuefloat ;
```

Sets a default for capacitance for all output pins in the library.

```
default_max_fanout : valuefloat ;
```

Sets a default for max_fanout for all output pins in the library.

```
default_max_transition : valuefloat ;
```

Sets a default for max_transition for all output pins in the library.

```
default_fanout_load : valuefloat ;
```

Sets a default for fanout_load for all input pins in the library.

The following example shows the default pin attributes in a CMOS library:

Example 5 Default Pin Attributes for a CMOS Library

```
library (example) {  
    ...  
    /* default pin attributes */  
    default_inout_pin_cap      : 1.0 ;  
    default_input_pin_cap      : 1.0 ;  
    default_output_pin_cap     : 0.0 ;  
    default_fanout_load       : 1.0 ;  
    default_max_fanout        : 10.0 ;  
    default_max_transition    : 15.0 ;  
    ...  
}
```

Setting Wire Load Defaults

Use the following library-level attributes to set wire load defaults.

default_wire_load Attribute

Assigns the defaults to the `wire_load` group, unless you assign a different value for `wire_load` before compiling the design.

Syntax

```
default_wire_load : wire_load_name;
```

Example

```
default_wire_load : WL1;
```

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_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 50](#).

Syntax

```
default_operating_conditions : operating_condition_name;
```

Example

```
default_operating_conditions : WCCOM1;
```

default_connection_class Attribute

Sets a default for `connection_class` for all pins in a library.

Example

```
default_connection_class : name1 [name2 name3 ...];
```

Examples of Library-Level Default Attributes

In [Example 6](#), the `wire_load` and `operating_conditions` group statements illustrate the requirement that group names that are referred to by the default attributes, such as `WL1` and `OP1`, must be defined in the library.

Example 6 Setting Library-Level Default Attributes for a CMOS Library

```
library (example) {
  ...
  /* default cell attributes */

  default_cell_leakage_power : 0.2;

  /* default pin attributes */

  default_inout_pin_cap : 1.0;
  default_input_pin_cap : 1.0;
  default_output_pin_cap : 0.0;
  default_fanout_load : 1.0;
  default_max_fanout : 10.0;

  wire_load (WL1) {
    ...
}
```

```
        }
    operating_conditions (OP1) {
        ...
    }
    default_wire_load : WL1;
    default_operating_conditions : OP1;
    default_wire_load_mode : enclosed;
    ...
}
```

Defining Operating Conditions

The following section explains how to define and determine various operating conditions for a logic library.

operating_conditions Group

An `operating_conditions` group is defined in a library group.

Syntax

```
library ( lib_name ) {
    operating_conditions ( name ) {
        ...
    }
}
```

name

Identifies the set of operating conditions. Names of all `operating_conditions` groups and `wire_load` groups must be unique within a library.

The `operating_conditions` groups are useful for testing timing and other characteristics of your design in predefined simulated environments. The following attributes are defined in an `operating_conditions` group:

`process : multiplier ;`

The scaling factor accounts for variations in the outcome of the actual semiconductor manufacturing steps, typically 1.0 for most technologies. The multiplier is a floating-point number from 0 through 100.

`process_label : "name" ;`

The process name of the current process. The value is a string.

`temperature : value ;`

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

`voltage : value ;`

The operating voltage of the design, typically 5 volts for a CMOS library. The value is a floating-point number from 0 through 1,000, representing the absolute value of the actual voltage.

Note:

Define voltage units consistently.

`tree_type : model ;`

The definition for the environment interconnect model.

.

The model is one of the following three models:

- `best_case_tree`

Models the case in which the load pin is physically adjacent to the driver. In the best case, all wire capacitance is incurred but none of the wire resistance must be overcome.

- `balanced_tree`

Models the case in which all load pins are on separate, equal branches of the interconnect wire. In the balanced case, each load pin incurs an equal portion of the total wire capacitance and wire resistance.

- `worst_case_tree`

Models the case in which the load pin is at the extreme end of the wire. In the worst case, each load pin incurs both the full wire capacitance and the full wire resistance.

Defining 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 about 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);  
}
```

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.

Chapter 3: Building Environments

Defining Wire Load Groups

You can define any number of `wire_load` groups in a logic 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.

When you create a wire load manually, define only `fanout` and `length`.

You must define at least one pair of `fanout` and `length` points per wire load model. You can define as many additional pairs as necessary to characterize the fanout-length behavior you want.

```
interconnect_delay (template_name)
{valuesfloat,...float,...float,...float,... ; }
```

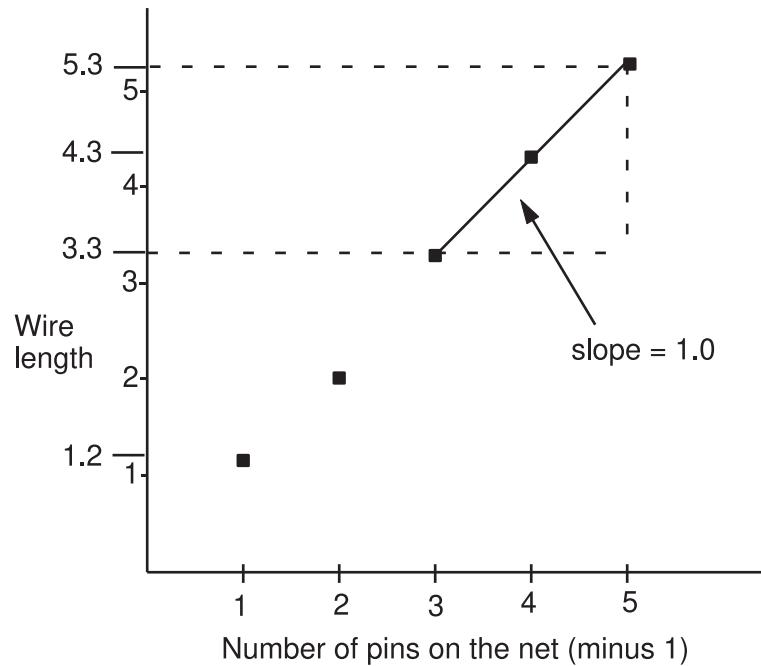
The `interconnect_delay` group specifies the lookup table template and the wire delay values.

Specify the `interconnect_delay` values.

To overwrite the default index values, specify the new index values before the `interconnect_delay` values, as shown here.

[Figure 3](#) 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 3 *Wire Load Definition Graph*



[Example 7](#) gives the library description for the model in [Figure 3](#).

Note:

In [Example 7](#) and [Example 8](#), the name 90x90 is enclosed in quotation marks because it is a string that begins with a number.

Example 7 *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 8 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. 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 7](#), enter

```
dc_shell> set_wire_load "90x90"
```

For more information, see the `set_wire_load` command in the Synopsys man pages.

If you don't define a `wire_load` group, the `default_wire_load_resistance` and `default_wire_load_capacitance` attribute values 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`.

Example 9 Report For a Wire Load Model Containing New fanout and 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

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 10 A library source file with the wire_load_table Group and 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
```

Specifying Delay Scaling Attributes

These k-factors (attributes that begin with `k_`) are multipliers that scale defined library values, taking into consideration the effects of changes in process, temperature, and voltage.

To model the effects of process, temperature, and voltage variations on circuit timing, use the following:

- k-factors that apply to the entire library and are defined at the library level.
- User-selected operating conditions that override the values in the library for an individual cell.

Using these values, downstream tools uniformly scale timing numbers for timing analysis.

Pin and Wire Capacitance Factors

The pin and wire capacitance factors scale the capacitance of a pin or wire according to process, temperature, and voltage variations. Each attribute gives the multiplier for a certain portion of the capacitance of a pin or a wire. In the following syntax, *multiplier* is a floating-point number:

`k_process_pin_cap : multiplier ;`

Scaling factor applied to pin capacitance to model process variation.

`k_process_wire_cap : multiplier ;`

Scaling factor applied to wire capacitance to model process variation.

`k_temp_pin_cap : multiplier ;`

Scaling factor applied to pin capacitance to model temperature variation.

`k_temp_wire_cap : multiplier ;`

Scaling factor applied to wire capacitance to model temperature variation.

`k_volt_pin_cap : multiplier ;`

Scaling factor applied to pin capacitance to model voltage variation.

`k_volt_wire_cap : multiplier ;`

Scaling factor applied to wire capacitance to model voltage variation.

Scaling Factors Associated With the Nonlinear Delay Model

The CMOS nonlinear delay model scaling factors scale the delay based on the variation in process, temperature, and voltage. The following scaling factors apply to the CMOS nonlinear delay model:

- `k_process_cell_rise`
- `k_temp_cell_rise`

Chapter 3: Building Environments

Specifying Delay Scaling Attributes

- 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

4

Defining Core Cells

Cell descriptions are a major part of a logic library. They provide information about the area, function, and timing of each component in an ASIC technology.

Defining core cells for CMOS logic libraries involves the following concepts and tasks described in this chapter:

- [Defining cell Groups](#)
 - [Defining pin Groups](#)
 - [Defining Bused Pins](#)
 - [Defining Signal Bundles](#)
 - [Defining Layout-Related Multibit Attributes](#)
 - [Defining Multiplexers](#)
 - [Defining Decoupling Capacitor Cells, Filler Cells, and Tap Cells](#)
 - [Fault Tolerant Cell Modeling](#)
-

Defining cell Groups

A `cell` group defines a single cell in the logic library.

This section discusses the attributes in a `cell` group.

For information about groups within a `cell`, see the following sections in this chapter:

- [Defining Bused Pins on page 100](#)
- [Defining Signal Bundles on page 106](#)

See [Chapter 6, Defining Test Cells](#),” for a test cell with a `test_cell` group. See [cell Group Example](#) for an example cell description.

cell Group

The `cell` group statement gives the name of the cell being described. It appears at the library group level, as shown here:

```
library (lib_name) {  
    ...  
    cell( name ) {  
        ... cell description ...  
    }  
    ...  
}
```

Use a *name* that corresponds to the name the ASIC vendor uses for the cell. When naming cells, remember that names are case-sensitive. For example, the cell names, AND2, and2, and And2 are all different. Cell names beginning with a number must be enclosed in quotation marks.

To create the `cell` group for the AND2 cell, use this syntax:

```
cell( AND2 ) {  
    ... cell description ...  
}
```

To describe a CMOS `cell` group, you use the `type` group and these attributes:

- `area`
- `bundle` ([See Defining Signal Bundles on page 106.](#))
- `bus` ([See Defining Bused Pins on page 100.](#))
- `cell_footprint`
- `clock_gating_integrated_cell`
- `contention_condition`
- `is_clock_gating_cell`
- `map_only`
- `pad_cell`
- `pad_type`
- `physical_variant_cells`
- `pin_equal`
- `pin_opposite`

- preferred
-

area Attribute

This attribute specifies the cell area.

Example

```
area : 2.0;
```

For unknown or undefined (black box) cells, the `area` attribute is optional. Unless a cell is a pad cell, it should have an `area` attribute. Pad cells should be given an area of 0.0, because they are not used as internal gates.

cell_footprint Attribute

This attribute assigns a footprint class to a cell.

Example

```
cell_footprint : 5MIL ;
```

Characters in the string are case-sensitive.

Use this attribute to assign the same footprint class to all cells that have the same layout boundary. Cells with the same footprint class are considered interchangeable and can be swapped during in-place optimization.

clock_gating_integrated_cell Attribute

An integrated clock-gating cell is a cell that you or your library developer creates to use especially for clock gating. The cell integrates the various combinational and sequential elements of a clock gate into a single cell that is compiled into gates and located in the logic library.

Consider using an integrated clock-gating cell if you are experiencing timing problems caused by the introduction of randomly chosen logic on your clock line.

Use the `clock_gating_integrated_cell` attribute to specify a value that determines the integrated cell functionality to be used by the clock-gating tools.

Syntax

```
clock_gating_integrated_cell : generic|valueid;
```

generic

When you specify the value generic, the actual type of clock gating integrated cell structure is determined by accessing the function specified on the library pin.

Note:

Statetables and state functions should not be used. Use latch groups with function groups instead.

value

A concatenation of up to four strings that describe the cell's functionality to the clock-gating tools:

- The first string specifies the type of sequential element you want. The options are latch-gating logic and none.
- The second string specifies whether the logic is appropriate for rising- or falling-edge-triggered registers. The options are posedge and negedge.
- The third (optional) string specifies whether you want test-control logic located before or after the latch or not at all. The options for cells set to latch are precontrol (before), postcontrol (after), or no entry. The options for cells set to no gating logic are control and no entry.
- The fourth (optional) string, which exists only if the third string does, specifies whether you want observability logic or not. The options are obs and no entry.

Example

```
clock_gating_integrated_cell : "latch_posedge_precontrol_obs" ;
```

Table 1 Values of the clock_gating_integrated_cell Attribute

When the value is	The integrated cell must contain
latch_negedge	Latch-based gating logic. Logic appropriate for falling-edge-triggered registers.
latch_posedge_postcontrol	Latch-based gating logic. Logic appropriate for rising-edge-triggered registers. Test-control logic located after the latch.
latch_negedge_precontrol	Latch-based gating logic. Logic appropriate for falling-edge-triggered registers. Test-control logic located before the latch.

When the value is	The integrated cell must contain
none_posedge_control_obs	Latch-free gating logic. Logic appropriate for rising-edge-triggered registers. Test-control logic (no latch). Observability logic.

Setting Pin Attributes for an Integrated Cell

The clock-gating tool requires that you set the pins of your integrated cells by using the attributes listed in [Table 2](#). Setting some of the pin attributes, such as those for test and observability, is optional.

Table 2 Pin Attributes for Integrated Clock-Gating Cells

Integrated cell pin name	Data direction	Required attribute
clock	in	clock_gate_clock_pin
enable	in	clock_gate_enable_pin
test_mode or scan_enable	in	clock_gate_test_pin
observability	out	clock_gate_obs_pin
enable_clock	out	clock_gate_out_pin

For details about these pin attributes, see the following sections:

- [clock_gate_clock_pin Attribute on page 73](#)
- [clock_gate_enable_pin Attribute on page 73](#)
- [clock_gate_obs_pin Attribute on page 73](#)
- [clock_gate_out_pin Attribute on page 74](#)
- [clock_gate_test_pin Attribute on page 74](#)

For more details about the `clock_gating_integrated_cell` attribute and the corresponding pin attributes, see [Chapter 8, Modeling Power and Electromigration](#), and the *Power Compiler User Guide*.

Setting Timing for an Integrated Cell

You set both the setup and hold arcs on the enable pin by setting the `clock_gate_enable_pin` attribute for the integrated cell to `true`. The setup

and hold arcs for the cell are determined by the edge values you enter for the `clock_gating_integrated_cell` attribute.

Table 3 Edge Values of the `clock_gating_integrated_cell` Attribute With Setup and Hold Arcs

Value	Setup arc	Hold arc
latch_posedge	rising	rising
latch_negedge	falling	falling
none_posedge	falling	rising
none_negedge	rising	falling

For details about setting timing for an integrated cell, see [Chapter 8, Modeling Power and Electromigration](#).

contention_condition Attribute

Contention is a clash of 0 and 1 signals. In certain cells, it can be a forbidden condition and cause circuits to short.

Example

```
contention_condition : "!ap * an" ;
```

is_macro_cell Attribute

The `is_macro_cell` attribute identifies whether a cell is a macro cell. If the attribute is set to `true`, the cell is a macro cell. If it is set to `false`, the cell is not a macro cell.

Example

```
is_macro_cell : true;
```

pad_cell Attribute

The `pad_cell` attribute in a `cell` group identifies the cell as a pad.

Example

```
pad_cell : true ;
```

If the `pad_cell` attribute is included in a cell definition, at least one pin in the cell must have an `is_pad` attribute.

If more than one pad cell is used to build a logical pad, put this attribute in the cell definitions of all the component pad cells:

```
auxiliary_pad_cell : true ;
```

If you omit the `pad_cell` or `auxiliary_pad_cell` attribute, the cell is treated as an internal core cell.

Note:

A cell with an `auxiliary_pad_cell` attribute can also be used within the core; a pull-up or pull-down cell is an example of such a cell.

physical_variant_cells Attribute

The `physical_variant_cells` attribute specifies a list of physical variant cells of the cell where it is defined.

Physical variant cells have different physical layouts but share the same logic Liberty models. For example, a design can have multiple cells that are variants of a master cell with only difference in the mask color of their pins. In smaller technology nodes, the timing of these cells are close enough for you to use the same Liberty model for the master cell and all its variants.

Syntax

```
library (library_name) {  
    cell (cell_name) {  
        physical_variant_cells : "names-list";  
        ...  
    }  
    ...  
}
```

Specify the list of cell names by separating them with a blank space or comma. The listed cell names are independent of order. For example, `physical_variant_cells : "ABC_1 ABC_2";` is identical to `physical_variant_cells : "ABC_2 ABC_1";`.

Example

The following is a .lib snippet of the master cell, `mycell`, whose color variant cells are `mycell_1`, `mycell_2`, and `mycell_3`. In the .lib file, all the four cells have the same characterization data. The physical views of these four cells might be slightly different.

```
library(mylib) {  
    ...  
    voltage_map(VDD, 1.0); /* Primary Power */  
    voltage_map(VSS, 0.0); /* Primary Ground */  
    ...  
    cell (mycell) {
```

```
physical_variant_cells : "mycell_1 mycell_2 mycell_3";
/* other cell level attributes and groups */
...
/* pg_pin definitions */
...
/* Signal pin definitions */
...
} /* end cell group */
} /* end library group */
```

pin_equal Attribute

This attribute describes a group of logically equivalent input or output pins in the cell.

pin_opposite Attribute

This attribute describes functionally opposite (logically inverse) groups of pins in a cell.
The `pin_opposite` attribute also incorporates the functionality of `pin_equal`.

Example

```
pin_opposite("Q1 Q2 Q3", "QB1 QB2") ;
```

In this example, Q1, Q2, and Q3 are equal; QB1 and QB2 are equal; and the pins of the first group are opposite to the pins of the second group.

The Design Compiler tool automatically determines the opposite pins for cells that have `function` attributes. See [function Attribute on page 96](#) for more information.

Note:

Use the `pin_opposite` attribute only in cells without function information or when you want to define required inputs.

type Group

The `type` group, when defined within a cell, is a type definition local to the cell. It cannot be used outside of the cell.

Example

```
type (bus4) {
    base_type : array;
    data_type : bit;
    bit_width : 4;
    bit_from : 0;
    bit_to : 3;
}
```

cell Group Example

[Example 11](#) shows cell definitions that include some of the CMOS cell attributes described in this section.

Example 11 cell Group Example

```
library (cell_example){  
    date : "August 14, 2015";  
    revision : 2015.03;  
    cell (inout){  
        pad_cell : true;  
        dont_use : true;  
        dont_fault : sa0;  
        dont_touch : true;  
        area : 0; /* pads do not normally consume internal  
                   core area */  
        cell_footprint : 5MIL;  
        pin (A) {  
            direction : input;  
            capacitance : 0;  
        }  
        pin (Z) {  
            direction : output;  
            function : "A";  
            timing () {  
                ...  
            }  
        }  
    }  
    cell(inverter_med){  
        area : 3;  
        preferred : true;  
        pin (A) {  
            direction : input;  
            capacitance : 1.0;  
        }  
        pin (Z) {  
            direction : output;  
            function : "A' ";  
            timing () {  
                ...  
            }  
        }  
    }  
    cell(nand){  
        area : 4;  
        pin(A) {  
            direction : input;  
            capacitance : 1;  
            fanout_load : 1.0;  
        }  
    }  
}
```

```
pin(B) {
    direction : input;
    capacitance : 1;
    fanout_load : 1.0;
}

pin (Y) {
    direction : output;
    function : "(A * B)' ";
    timing() {
        ...
    }
}
cell(buff1) {
    area : 3;
    pin (A) {
        direction : input;
        capacitance : 1.0;
    }
    pin (Y) {
        direction : output;
        function : "A ";
        timing () {
            ...
        }
    }
}
} /* End of Library */
```

mode_definition Group

A mode_definition group declares a mode group that contains several timing mode values.

Syntax

```
cell(name_string) {
    mode_definition(name_string) {
        mode_value(name1) {
            when : "Boolean expression" ;
            sdf_cond : "sdf_expression_string" ;
        }
        mode_value(name_string) {
            when : "Boolean expression" ;
            sdf_cond : "Boolean expression" ;
        }
    }
}
```

Group Statement

```
mode_value (name_string) { }
```

Specifies the condition that a timing arc depends on to activate a path.

mode_value Group

The `mode_value` group contains several mode values within a `mode` group. You can optionally put a condition on a mode value. When the condition is true, the `mode` group takes that value.

Syntax

```
mode_value (name_string) { }
```

Simple Attributes

```
when : "Boolean expression" ;  
sdf : "Boolean expression" ;
```

when Simple Attribute

The `when` attribute specifies the condition that a timing arc depends on to activate a path. The valid value is a Boolean expression.

Syntax

```
when : "Boolean expression" ;
```

Example

```
when: !R;
```

sdf_cond Simple Attribute

The `sdf_cond` attribute supports Standard Delay Format (SDF) file generation and condition matching during back-annotation.

Syntax

```
sdf_cond : "Boolean expression" ;
```

Example

```
sdf_cond: "R == 0";
```

Example 12 mode_definition Group Description

```
cell(example_cell) {  
...  
mode_definition(rw) {
```

```
mode_value(read) {
    when : "R";
    sdf_cond : "R == 1";
}
mode_value(write) {
    when : "!R";
    sdf_cond : "R == 0";
}
```

Defining pin Groups

For each pin in a cell, the `cell` group must contain a description of the pin characteristics. You define pin characteristics in a `pin` group within the `cell` group.

A `pin` group often contains a `timing` group and an `internal_power` group.

For more information about `timing` groups, see [Chapter 7, Timing Arcs](#).

For more information about the `internal_power` group, see [Chapter 8, Modeling Power and Electromigration](#).

pin Group

You can define a `pin` group within a `cell`, `test_cell`, `model`, or `bus` group.

```
library (lib_name) {
...
  cell (cell_name) {
    ...
      pin ( name | name_list ) {
        ... pin group description ...
      }
    cell (cell_name) {
      ...
        bus (bus_name) {
          ... bus group description ...
        }
      bundle (bundle_name) {
        ... bundle group description ...
      }
      pin ( name | name_list ) {
        ... pin group description ...
      }
    }
  }
}
```

See [Defining Bused Pins on page 100](#) for descriptions of `bus` groups. See [Defining Signal Bundles on page 106](#) for descriptions of `bundle` groups.

The `pin` groups are also valid within `test_cell` groups. They have different requirements from `pin` groups in `cell`, `bus`, or `bundle` groups. See [Pins in the test_cell Group on page 158](#) for specific information and restrictions on describing test pins.

All pin names within a single `cell`, `bus`, or `bundle` group must be unique. Names are case-sensitive: pins named `A` and `a` are different pins.

You can describe pins with common attributes in a single `pin` group. If a cell contains two pins with different attributes, two separate `pin` groups are required. Grouping pins with common technology attributes can significantly reduce the size of a cell description that includes many pins.

In the following example, the AND cell has two pins: A and B.

```
cell (AND) {
    area : 3 ;
    pin (A) {
        direction : input ;
        capacitance : 1 ;
    }
    pin (B) {
        direction : input ;
        capacitance : 1 ;
    }
}
```

Because pins A and B have the same attributes, the cell can also be described as

```
cell (AND) {
    area : 3 ;
    pin (A,B) {
        direction : input ;
        capacitance : 1 ;
    }
}
```

General pin Group Attributes

To define a pin, use these general `pin` group attributes:

- `capacitance`
- `clock_gate_clock_pin`
- `clock_gate_enable_pin`
- `clock_gate_obs_pin`
- `clock_gate_out_pin`
- `clock_gate_test_pin`

- complementary_pin
- connection_class
- direction
- dont_fault
- driver_type
- fall_capacitance
- fault_model
- inverted_output
- is_analog
- pin_func_type
- rise_capacitance
- steady_state_resistance
- test_output_only

For a complete list and descriptions for all the attributes and groups that you can specify in a pin group, see Chapter 3, “pin Group Description and Syntax,” in the *Synopsys Liberty Reference Manual*.

capacitance Attribute

The `capacitance` attribute defines the load of an input, output, inout, or internal pin. The load is defined with a floating-point number, in units consistent with other capacitance specifications throughout the library. Typical units of measure for capacitance include picofarads and standardized loads.

Example

The following example defines the A and B pins in an AND cell, each with a capacitance of one unit.

```
cell (AND) {  
    area : 3 ;  
    pin (A,B) {  
        direction : input ;  
        capacitance : 1 ;  
    }  
}
```

If the `timing` groups in a cell include the output-pin capacitance effect in the intrinsic-delay specification, do not specify capacitance values for the cell’s output pins.

clock_gate_clock_pin Attribute

The `clock_gate_clock_pin` attribute identifies an input pin connected to a clock signal.

Valid values for this attribute are `true` and `false`.

Example

```
clock_gate_clock_pin : true;
```

See [clock_gating_integrated_cell Attribute on page 61](#) for more information about identifying pins on integrated clock-gating cells. For additional information about integrated clock-gating, see the *Power Compiler User Guide*.

clock_gate_enable_pin Attribute

The Design Compiler tool uses the `clock_gate_enable_pin` attribute when it compiles a design containing gated clocks that were introduced by the Power Compiler tool.

The `clock_gate_enable_pin` attribute identifies an input port connected to an enable signal for nonintegrated clock-gating cells and integrated clock-gating cells.

Valid values for this attribute are `true` and `false`.

Example

```
clock_gate_enable_pin : true;
```

For nonintegrated clock-gating cells, you can set the `clock_gate_enable_pin` attribute to `true` on only one input port of a 2-input AND, NAND, OR, or NOR gate. If you do so, the other input port is the clock.

See [clock_gating_integrated_cell Attribute on page 61](#) for more information about identifying pins on integrated clock-gating cells. For additional information about integrated clock gating, see the *Power Compiler User Guide*.

clock_gate_obs_pin Attribute

The `clock_gate_obs_pin` attribute identifies an output port connected to an observability signal.

Valid values for this attribute are `true` and `false`.

Example

```
clock_gate_obs_pin : true;
```

See [clock_gating_integrated_cell Attribute on page 61](#) for more information about identifying pins on integrated clock-gating cells. For additional information about integrated clock gating, see the *Power Compiler User Guide*.

clock_gate_out_pin Attribute

The `clock_gate_out_pin` attribute identifies an output port connected to an `enable_clock` signal..

Valid values for this attribute are `true` and `false`.

Example

```
clock_gate_out_pin : true;
```

See [clock_gating_integrated_cell Attribute on page 61](#) for more information about identifying pins on integrated clock-gating cells. For additional information about integrated clock gating, see the *Power Compiler User Guide*.

clock_gate_test_pin Attribute

The `clock_gate_test_pin` attribute identifies an input port connected to a `test_mode` or `scan_enable` signal.

Valid values for this attribute are `true` and `false`.

Example

```
clock_gate_test_pin : true;
```

See [clock_gating_integrated_cell Attribute on page 61](#) for more information about identifying pins on integrated clock-gating cells. For additional information about integrated clock gating, see the *Power Compiler User Guide*.

complementary_pin Simple Attribute

The `complementary_pin` attribute supports differential I/O.

Differential I/O assumes the following:

- When the noninverting pin equals 1 and the inverting pin equals 0, the signal gets logic 1.
- When the noninverting pin equals 0 and the inverting pin equals 1, the signal gets logic 0.

The entry for this attribute identifies the differential input inverting pin with which the noninverting pin is associated and from which it inherits timing information and associated attributes.

Syntax

```
complementary_pin : "string" ;
```

string

Identifies the differential input data inverting pin whose timing information and associated attributes the noninverting pin inherits. Only one input pin is modeled at the cell level. The associated differential inverting pin is defined in the same pin group.

For details on the `fault_model` attribute used to define the value when both the complementary pin and the pin that it complements are driven to the same value, see [fault_model Simple Attribute on page 82](#).

Example

```
cell (diff_buffer) {  
    ...  
    pin (A) { /* noninverting pin /  
        direction : input ;  
        complementary_pin : "DiffA" /* inverting pin /  
    }  
}
```

connection_class Simple Attribute

The `connection_class` attribute lets you specify design rules for connections between cells.

Example

```
connection_class : "internal";
```

Only pins with the same connection class can be legally connected. For example, you can specify that clock input must be driven by clock buffer cells or that output pads can be driven only by high-drive pad driver cells between the internal logic and the pad. To do this, you assign the same connection class to the pins that must be connected. For the pad example, you attach a given connection class to the pad driver output and the pad input. This attachment makes it invalid to connect another type of cell to the pad.

•

•

In [Example 13](#), the `output_pad` cell can be driven only by the `pad_driver` cell. The pad driver's input can be connected to internal core logic, because it has the internal connection class.

In [Example 14](#), the `high_drive_buffer` cell can drive internal core cells and pad cells, whereas the `low_drive_buffer` cell can drive only internal cells.

Example 13 Connection Class Example

```
default_connection_class : "default" ;
cell (output_pad) {
    pin (IN) {
        connection_class : "external_output" ;
    ...
}
cell (pad_driver) {
    pin (OUT) {
        connection_class : "external_output" ;
    ...
}
    pin (IN) {
        connection_class : "internal" ;
    ...
}
}
```

Example 14 Multiple Connection Classes for a Pin

```
cell (high_drive_buffer) {
    pin (OUT) {
        connection_class : "internal pad" ;
    ...
}
cell (low_drive_buffer) {
    pin (OUT) {
        connection_class : "internal" ;
    ...
}
}

cell (pad_cell) {
    pin (IN) {
        connection_class : "pad" ;
    ...
}
cell (internal_cell) {
    pin (IN) {
        connection_class : "internal" ;
    ...
}
}
```

direction Attribute

Use this attribute to specify whether the pin being described is an input, output, internal, or bidirectional pin.

Example

```
direction : output;
```

For a description of the effect of pin direction on path tracing and timing analysis, see the *Design Compiler Optimization Reference Manual*.

driver_type Attribute

Use the optional `driver_type` attribute to modify the signal on a pin. This attribute specifies a signal mapping mechanism that supports the signal transitions performed by the circuit.

The `driver_type` attribute tells the application tool to use a special pin-driving configuration for the pin during simulation. A pin without this attribute has normal driving capability by default.

A driver type can be one or more of the following:

pull_up

The pin is connected to DC power through a resistor. If it is a three-state output pin and it is in the Z state, its function is evaluated as a resistive 1 (H). If it is an input or inout pin and the node to which it is connected is in the Z state, it is considered an input pin at logic 1 (H). For a pull-up cell, the pin stays constantly at logic 1 (H).

pull_down

The pin is connected to DC ground through a resistor. If it is a three-state output pin and it is in the Z state, its function is evaluated as a resistive 0 (L). If it is an input or inout pin and the node to which it is connected is in the Z state, it is considered an input pin at logic 0 (L). For a pull-down cell, the pin stays constantly at logic 0 (L).

bus_hold

The pin is a bidirectional pin on a bus holder cell. The pin holds the last logic value present at that pin when no other active drivers are on the associated net. Pins with this driver type cannot have function or `three_state` statements.

open_drain

The pin is an output pin without a pull-up transistor. Use this driver type only for off-chip output or inout pins representing pads. The pin goes to high impedance (Z) when its function is evaluated as logic 1.

open_source

The pin is an output pin without a pull-down transistor. Use this driver type only for off-chip output or inout pins representing pads. The pin goes to high impedance (Z) when its function is evaluated as logic 0.

resistive

The pin is an output pin connected to a controlled pull-up or pull-down driver with a control port (input). When the control port is disabled, the pull-up or pull-down driver is turned off and has no effect on the pin. When the control port is enabled, a functional value of 0 evaluated at the pin is turned into a weak 0 (L), a functional value of 1 is turned into a weak 1 (H), but a functional value of Z is not affected.

resistive_0

The pin is an output pin connected to DC power through a pull-up driver that has a control port (input). When the control port is disabled, the pull-up driver is turned off and has no effect on the pin. When the control port is enabled, a functional value of 1 evaluated at the pin is turned into a weak 1 (H) but the functional values of 0 and Z are not affected.

resistive_1

The pin is an output pin connected to DC ground through a pull-down driver that has a control port (input). When the control port is disabled, the pull-down driver is turned off and has no effect on the pin. When the control port is enabled, a functional value of 0 evaluated at the pin is turned into a weak 0 (L) but the functional values of 1 and Z are not affected.

Except for inout pins, each pin can have only one `driver_type` attribute.

Inout pins can have two driver types, one for input and one for output. The only valid combinations are `pull_up` or `pull_down` for input and `open_drain` for output. If you specify only one driver type and it is `bus_hold`, it is used for both input and output. If the single driver type is not `bus_hold`, it is used for output. Specify multiple driver types in one entry in this format:

```
driver_type : "driver_type1 driver_type2" ;
```

Example

This is an example of a pin connected to a controlled pull-up cell that results in a weak 1 when the control port is enabled.

```
function : 1;
driver_type : resistive;
three_state_enable : EN;
```

Interpretation of Driver Types

The driver type specifies one of the following signal modifications:

Resolve the value of Z

These driver types resolve the value of Z on an existing circuit node, implying a constant 0 or 1 signal source. They do not perform a function. Resolution driver types are pull_up, pull_down, and bus_hold.

Transform the signal

These driver types perform an actual function on an input signal, mapping the transition from 0 or 1 to L, H, or Z. Transformation driver types are open_drain, open_source, resistive, resistive_0, and resistive_1.

For output pins, the `driver_type` attribute is applied after the pin's functional evaluation. For input pins, this attribute is applied before the signal is used for functional evaluation.

Table 4 Signal Mapping and Pin Types for Different Driver Types

Driver type	Description	Signal mapping	Applicable pin types
pull_up	Resolution	01Z -> 01H	in, out
pull_down	Resolution	01Z -> 01L	in, out
bus_hold	Resolution	01Z -> 01S	inout
open_drain	Transformation	01Z -> 0ZZ	out
open_source	Transformation	01Z -> Z1Z	out
resistive	Transformation	01Z -> LHZ	out
resistive_0	Transformation	01Z -> 0HZ	out
resistive_1	Transformation	01Z -> L1Z	out

Signal Mapping:

0 and 1 represent strong logic 0 and logic 1 values

L represents a weak logic 0 value

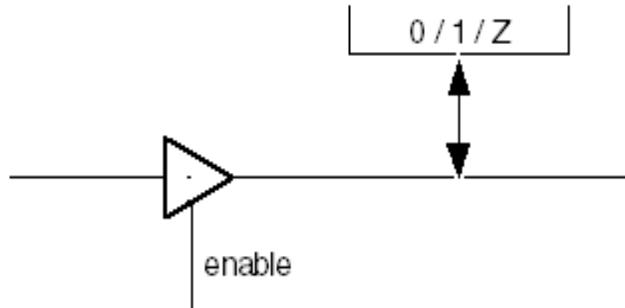
H represents a weak logic 1 value

Z represents high impedance

S represents the previous state

Interpreting Bus Holder Driver Type

Figure 4 01Z to 01S Signal Mapping for bus_hold Driver Type



For bus_hold driver types, a three-state buffer output value of 0 changes the bus value to 0. Similarly, a three-state buffer output value of 1 changes the bus value to 1. However, when the output of the three-state buffer is Z, the bus holds its previous value (S), which can be 0, 1, or Z. In other words, the buffer output value of Z is resolved to the previous value of the bus.

Modeling Pull-Up and Pull-Down Cells

Figure 5 Pull-Up Resistor of a Cell



[Example 15](#) is the description of the pull-up resistor cell in Figure 5.

Example 15 Description of a Pull-Up Cell Transistor

```
cell(pull_up_cell) {  
    area : 0;  
    auxiliary_pad_cell : true;  
    pin(Y) {  
        direction : output;  
        multicell_pad_pin : true;  
        connection_class : "inpad_network";  
        driver_type : pull_up;  
        pulling_resistance : 10000;  
    }  
}
```

[Example 16](#) describes an output pin with a pull-up resistor and the bidirectional pin on a bus holder cell.

Example 16 Pin Driver Type Specifications

```
pin(Y) {  
    direction : output ;  
    driver_type : pull_up ;  
    pulling_resistance : 10000 ;  
    function : "IO" ;  
    three_state : "OE" ;  
}  
cell (bus_hold) {  
    pin(Y) {  
        direction : inout ;  
        driver_type : bus_hold ;  
    }  
}
```

Bidirectional pads can often require one driver type for the output behavior and another associated with the input. For this case, you can define multiple driver types in one `driver_type` attribute:

```
driver_type : "open_drain pull_up" ;
```

Note:

An *n*-channel open-drain pad is flagged with `open_drain`, and a *p*-channel open-drain pad is flagged with `open_source`.

fall_capacitance Attribute

Defines the load for an input and inout pin when its signal is falling.

Setting a value for the `fall_capacitance` attribute requires that a value for the `rise_capacitance` also be set, and setting a value for the `rise_capacitance` requires that a value for the `fall_capacitance` also be set.

Syntax

```
fall_capacitance : float ;
```

float

A floating-point number in units consistent with other capacitance specifications throughout the library. Typical units of measure for `fall_capacitance` include picofarads and standardized loads.

The following example defines the A and B pins in an AND cell, each with a `fall_capacitance` of one unit, a `rise_capacitance` of two units, and a capacitance of two units.

Example

```
cell (AND) {
```

```
area : 3 ;
vhdl_name : "AND2" ;
pin (A, B) {
    direction : input ;
    fall_capacitance : 1 ;
    rise_capacitance : 2 ;
    capacitance : 2 ;
}
}
```

fault_model Simple Attribute

The differential I/O feature enables an input noninverting pin to inherit the timing information and all associated attributes of an input inverting pin in the same `pin` group designated with the `complementary_pin` attribute.

If you enter a `fault_model` attribute, you must designate the inverted pin associated with the noninverting pin, using the `complementary_pin` attribute.

For details on the `complementary_pin` attribute, see [complementary_pin Simple Attribute on page 74](#).

Syntax

```
fault_model : "two-value string" ;
```

two-value string

Two values that define the value of the differential signals when both inputs are driven to the same value. The first value represents the value when both input pins are at logic 0; the second value represents the value when both input pins are at logic 1. Valid values for the two-value string are any two-value combinations of 0, 1, and x.

If you do not enter a `fault_model` attribute value, the signal pin value goes to x when both input pins are 0 or 1.

Example

```
cell (diff_buffer) {
    ...
    pin (A) { /* noninverting pin /
        direction : input ;
        complementary_pin : ("DiffA")
        fault_model : "1x" ;
    }
}
```

[Table 5](#) shows how testing interprets the complementary pin values for this example:

Table 5 Interpretation of Pin Values

Pin A (noninverting pin)	DiffA (complementary_pin)	Resulting signal pin value
1	0	1
0	1	0
0	0	1
1	1	x

inverted_output Attribute

The `inverted_output` attribute is a Boolean attribute that you can set for any output port. It is a required attribute only for sequential cells.

Set this attribute to false for noninverting output, which is variable1 or IQ for flip-flop or latch groups. Set this attribute to true for inverting output, which is variable2 or IQN for flip-flop or latch groups.

Example

```
pin(Q) {  
    function : "IQ";  
    internal_node : "IQ";  
    inverted_output : false;  
}
```

This attribute affects the internal interpretation of the state table format used to describe a sequential cell.

is_analog Attribute

The `is_analog` attribute identifies an analog signal pin as analog so it can be recognized by tools. The valid values for `is_analog` are `true` and `false`. Set the `is_analog` attribute to `true` at the pin level to specify that the signal pin is analog.

Syntax

The syntax for the `is_analog` attribute is as follows:

```
cell (cell_name) {  
    ...  
    pin (pin_name) {  
        is_analog: true | false ;  
        ...  
    }  
}
```

}

Example

The following example identifies the pin as an analog signal pin.

```
pin(Analog) {  
    direction : input;  
    capacitance : 1.0 ;  
    is_analog : true;  
}
```

pin_func_type Attribute

This attribute describes the functions of a pin.

Example

```
pin_func_type : clock_enable;
```

With the `pin_func_type` attribute, you avoid the checking and modeling caused by incomplete timing information about the enable pin. The information in this attribute defines the clock as the clock-enabling mechanism (that is, the clock-enable pin). This attribute also specifies whether the active level of the enable pin of latches is high or low and whether the active edge of the flip-flop clock is rising or falling.

The `report_lib` command lists the pins of a cell if they have one of these `pin_func_type` values: active falling, active high, active low, active rising, or clock enable.

steady_state_resistance Attributes

When there are multiple drivers connected to an interconnect network driven by library cells and there is no direct current path between them, the driver resistances could take on different values.

Use the following attributes for more-accurate modeling of steady state driver resistances in library cells.

- `steady_state_resistance_above_high`
- `steady_state_resistance_below_low`
- `steady_state_resistance_high`
- `steady_state_resistance_low`

Example

```
steady_state_resistance_above_high : 200 ;
```

test_output_only Attribute

This attribute is an optional Boolean attribute that you can set for any output port described in statetable format.

In ff or latch format, if a port is to be used for both function and test, you provide the functional description using the `function` attribute. If a port is to be used for test only, you omit the `function` attribute.

Regardless of ff or latch statetable, the `test_output_only` attribute takes precedence over the functionality.

In statetable format, however, a port always has a functional description. Therefore, if you want to specify that a port is for test only, you set the `test_output_only` attribute to true.

Example

```
pin (my_out) {  
    direction : output ;  
    signal_type : test_scan_out ;  
    test_output_only : true ;  
}
```

Describing Design Rule Checks

To define design rule checks, use the following `pin` group attributes and group:

- `fanout_load` attribute
- `max_fanout` attribute
- `min_fanout` attribute
- `max_transition` attribute
- `max_trans` group
- `min_transition` attribute
- `max_capacitance` attribute
- `max_cap` group
- `min_capacitance` attribute
- `cell_degradation` group

fanout_load Attribute

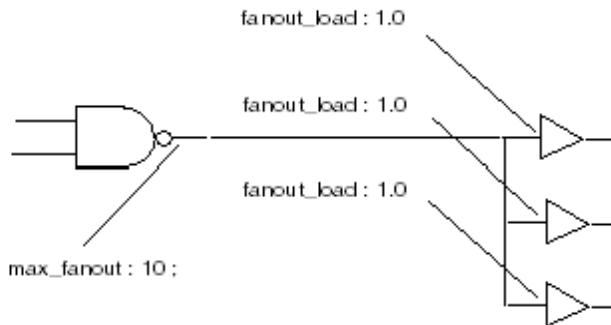
The `fanout_load` attribute gives the fanout load value for an input pin.

Example

```
fanout_load : 1.0;
```

The sum of all `fanout_load` attribute values for input pins connected to a driving (output) pin must not exceed the `max_fanout` value for that output pin.

Figure 6 Fanout Attributes of a Cell (`max_fanout` and `fanout_load`)



max_fanout Attribute

This attribute defines the maximum fanout load that an output pin can drive.

Example

```
pin(Q)
  direction : output;
  max_fanout : 10;
}
```

You can specify `max_fanout` at three levels: at the library level, at the pin level, and on the command line in `dc_shell`.

The `max_fanout` attribute is an implied design rule constraint. The Design Compiler tool attempts to resolve `max_fanout` violations, possibly at the expense of other design constraints.

In the figure in [fanout_load Attribute](#), the Design Compiler tool adds all the `fanout_load` values of the input to equal 3.0. Then the Design Compiler tool compares the sum of the `fanout_load` on all inputs with the `max_fanout` constraint of the driving output, which is 10. In this case, no design rule violation occurs. If a design rule violation occurs, the Design Compiler tool might replace the driving cell with a cell that has a higher `max_fanout` value.

When you are selecting a `max_fanout` value, the Design Compiler tool follows these rules:

- The value you assign for the pin always overrides the library default.

- When you assign a `max_fanout` value at the command line, the Design Compiler tool uses the smallest value. If the pin value is smaller, the Design Compiler tool ignores the command-line value.

To specify a global maximum fanout for all gate outputs in the design, use `max_fanout` on the command line of `dc_shell`. When defined on the command line, the value of `max_fanout` is assigned to all gate outputs that do not have a specified `max_fanout` or whose `max_fanout` value is greater than the one defined.

It differs from the capacitance pin attribute in the following ways:

- The fanout values are used only for design rule checking.
- The capacitance values are used delay calculations.

Some designs have limitations on input load that an output pin can drive regardless of any loading contributed by interconnect metal layers. To limit the number of inputs on the same net driven by an output pin, define a `max_fanout` value for each output pin and a `fanout_load` on each input pin in a cell. (See the figure in [fanout_load Attribute](#).)

To determine `max_fanout`, find the smallest loading of any input to a cell in the library and use that value as the standard load unit for the entire library. Usually the smallest buffer or inverter has the lowest input pin loading value. Use some multiple of the standard value for the fanout loads of the other cells.

Although you can use capacitance as the unit for your `max_fanout` and `fanout_load` specifications, it should be used to constrain routability requirements.

min_fanout Attribute

This attribute defines the minimum fanout load that an output or inout pin can drive. The sum of fanout cannot be less than the minimum fanout value.

Example

```
pin(Q) {  
    direction : output;  
    min_fanout : 2.0;  
}
```

The `min_fanout` attribute is an implied design rule constraint. The Design Compiler tool resolves `min_fanout` violations, possibly at the expense of other design constraints.

max_transition Attribute

This attribute defines a design rule constraint for the maximum acceptable transition time of an input or output pin.

Example

```
pin(A) {
```

```
    direction : input;
    max_transition : 4.2;
}
```

You can specify `max_transition` at three levels: at the library level, at the pin level, and on the command line.

With an output pin, `max_transition` is used only to drive a net for which the cell can provide a transition time at least as fast as the defined limit.

With an input pin, `max_transition` indicates that the pin cannot be connected to a net that has a transition time greater than the defined limit.

In the following example, the cell that contains pin Q cannot be used to drive a net for which the cell cannot provide a transition time faster than 5.2:

```
pin(Q) {
    direction : output ;
    max_transition : 5.2 ;
}
```

The `max_transition` value you define is checked by the synthesis tool, the transition delay that the Design Compiler tool calculates is the rise and fall resistance multiplied by the sum of the pin and wire capacitances.

If the calculated delay is greater than the value you specify with the `max_transition` attribute, a design rule violation is reported, and the Design Compiler tool tries to correct the violation, possibly at the expense of other design constraints.

max_trans Group

The `max_trans` group specifies the maximum transition time of an input, output, or inout pin as a function of operating frequency of the cell, input transition time, and output load. Use the `max_trans` group instead of the `max_transition` attribute to include these effects on the maximum transition time. When both the `max_trans` group and `max_transition` attribute are present, the `max_trans` group overrides the `max_transition` attribute.

To model the effect of operating frequency, input transition time, and output load on the maximum transition, define the lookup table template for the `max_trans` group at the library level.

To define the lookup table template, use the `maxtrans_lut_template` group at the library level. The `maxtrans_lut_template` group can have the variables, `variable_1`, `variable_2`, and `variable_3`. The valid values of the `variable_1`, `variable_2`, and `variable_3` variables are `frequency`, `input_transition_time`, and `total_output_net_capacitance` respectively.

The one-dimensional lookup table consists of the maximum transition values for different values of `frequency`. Similarly, the two-dimensional lookup table consists of the maximum

Chapter 4: Defining Core Cells

Defining pin Groups

transition values for different values of frequency and input_transition_time and so on.

The following syntax shows the maximum transition model:

```
library (library_name) {
    delay_model : table_lookup;
    ...
    maxtrans_lut_template (template_name1) { /*1-D LUT template*/
        variable_1 : frequency ;
        index_1 ("float, ... float");
    }

    maxtrans_lut_template (template_name2) { /*2-D LUT template*/
        variable_1 : frequency ;
        variable_2 : input_transition_time ;
        index_1 ("float, ... float");
        index_2 ("float, ... float");
    }

    maxtrans_lut_template (template_name3) { /*3-D LUT template*/
        variable_1 : frequency ;
        variable_2 : input_transition_time ;
        variable_3 : total_output_net_capacitance ;
        index_1 ("float, ... float");
        index_2 ("float, ... float");
        index_3 ("float, ... float");
    }
}

cell (cell_name) {
    ...
    pin (pin_name) { /* input pin */
        ...
        max_trans (template_name1) {
            index_1 ("float, ... float");
            values ("float, ... float");
        }
        ...
    } /* pin */
    ...
    pin (pin_name) { /* input pin */
        ...
        max_trans (template_name2) { /* output or inout pin */
            index_1 ("float, ... float");
            index_2 ("float, ... float");
            values ("float, ... float");
            values ("float, ... float");
        }
        ...
    } /* input pin */
    ...
    max_trans (template_name3) { /* output or inout pin */
        index_1 ("float, ... float");
        index_2 ("float, ... float");
    }
}
```

```
    index_3 ("float, ... float");
    values ("float, ... float");
    values ("float, ... float");
    values ("float, ... float");
}
} /* pin */
}/* cell */
...
}/* library */
```

Maximum Transition Modeling Example

Example 17 Frequency-Dependent Maximum Transition Model With One-Dimensional Lookup Table

```
library(my_lib) {
  technology (cmos);
  delay_model : table_lookup;
  ...
  maxtrans_lut_template ( mt ) {
    variable_1 : frequency;
    index_1 ( "100.0000, 200.0000" );
  }
  ...
  cell ( test ) {
    .....
    pin(Z) {
      direction : output ;
      function: "A";
      ...
      max_trans(mt) {
        index_1 ( "100.0000, 200.0000, 300.0000");
        values ( "925.0000, 835.0000, 745.0000");
      } /* end of max_trans */
      timing () {
        related_pin : "A" ;
        ...
      } /* end of arc */
    } /* end of pin Z */
    pin(A) {
      direction : input ;
      max_transition : 5000.000000 ;
      capacitance : 1.0 ;
      } /* end of pin A */
    } /* end of cell */
  }/* end of library */
```

min_transition Attribute

This attribute defines a design rule constraint for the minimum acceptable transition time of an input, output, or an inout pin.

Example

```
pin(A) {  
    direction : input;  
    min_transition : 1.2;  
}
```

max_capacitance Attribute

This attribute defines the maximum total capacitive load that an output pin can drive. This attribute can be specified only for an output or inout pin.

Example

```
pin(Q) {  
    direction : output;  
    max_capacitance : 5.0;  
}
```

You can specify `max_capacitance` at three levels: at the library level, at the pin level, and on the command line.

The Design Compiler tool uses an output pin only when the pin has a `max_capacitance` attribute value greater than or equal to the total wire and pin capacitive load it drives.

The total wire and pin capacitance is derated before it is checked by this tool. The tool attempts to resolve `max_capacitance` design rule violations, possibly at the expense of other design constraints.

max_cap Group

The `max_cap` group specifies the maximum capacitive load that an output or inout pin can drive as a function of operating frequency of the cell or both operating frequency and input transition time. Use the `max_cap` group instead of the `max_capacitance` attribute to include these effects on the maximum capacitance. When both the `max_cap` group and `max_capacitance` attribute are present, the `max_cap` group overrides the `max_capacitance` attribute.

To model the effect of operating frequency and input transition time on the maximum capacitance, define the lookup table template for the `max_cap` group at the library level.

To define the lookup table template, use the `maxcap_lut_template` group at the library level. The `maxcap_lut_template` group can have the `variables`, `variable_1` and `variable_2`. The valid values of the `variable_1` and `variable_2` variables are `frequency` and `input_transition_time`, respectively.

The one-dimensional lookup table consists of the maximum capacitance values for different values of `frequency`. The `max_cap` group takes the value of the maximum capacitance from the lookup table by using the `variable_1` variable.

The two-dimensional lookup table consists of the maximum capacitance values for different values of frequency and input_transition_time. The max_cap group takes the value of the maximum capacitance from the lookup table by using the variable_1 and variable_2 variables.

The following shows the maximum capacitance model syntax:

```
library (library_name) {
    delay_model : table_lookup;
    ...
    maxcap_lut_template (template_name1) { /*1-D lookup table template*/
        variable_1 : frequency ;
        index_1 ( "float, ... float" );
    }
    maxcap_lut_template (template_name2) { /*2-D LUT template */
        variable_1 : frequency ;
        variable_2 : input_transition_time ;
        index_1 ( "float, ... float" );
        index_2 ( "float, ... float" );
    }
    cell (cell_name) {
        ...
        pin (pin_name) {
            ...
            max_cap (template_name1) {
                index_1 ("float, ... float");
                values ("float, ... float");
            }
            ...
        } /* pin */
        ...
    }/* cell */
    ...
}/* library */
```

Maximum Capacitance Modeling Example

Example 18 Frequency-Dependent Maximum Capacitance Model With One-Dimensional Lookup Table

```
library(example_library) {
    technology (cmos);
    delay_model : table_lookup;
    ...
    maxcap_lut_template ( mc ) {
        variable_1 : frequency;
        index_1 ( "100.0000, 200.0000" );
    }
    ...
    cell ( test ) {
        .....
        pin(Z) {
```

```
direction : output ;
function: "A";
max_transition : 5000.000000 ;
min_transition : 0.000000 ;
min_capacitance : 0.000000 ;
max_cap(mc) {
    index_1 ( "100.0000, 200.0000, 300.0000");
    values ( "925.0000, 835.0000, 745.0000");
} /* end of max_cap */
timing () {
    related_pin : "A" ;
    ...
} /* end of arc */
} /* end of pin Z */
pin(A) {
    direction : input ;
    max_transition : 5000.000000 ;
    capacitance : 1.0 ;
    } /* end of pin A */
} /* end of cell */
} /* end of library */
```

min_capacitance Attribute

This attribute defines the minimum total capacitive load that an output pin can drive. The capacitance load cannot be less than the minimum capacitance value. This attribute can be specified only for an output or inout pin.

Example

```
pin(Q) {
    direction : output;
    min_capacitance : 1.0;
}
```

The Design Compiler tool uses an output pin only when the pin has a `min_capacitance` attribute value less than or equal to the total wire and pin capacitive load it drives. The total wire and pin capacitance is derated before it is checked by the tool. The tool attempts to resolve `min_capacitance` design rule violations, possibly at the expense of other design constraints.

cell_degradation Group

Use the `cell_degradation` group to describe a cell performance degradation design rule when compiling a design. A cell degradation design rule specifies the maximum capacitive load a cell can drive without causing cell performance degradation during the fall transition.

This description is restricted to functionally related input and output pairs. You can determine the degradation value by switching some inputs while keeping other inputs

constant. This causes output discharge. The degradation value for a specified input transition rate is the maximum output loading that does not cause cell degradation.

You can model cell degradation only in libraries using the CMOS nonlinear delay model. Cell degradation modeling uses the same format of templates and lookup tables used to model delay with the nonlinear delay model.

There are two ways to model cell degradation,

1. Create a one-dimensional lookup table template that is indexed by input transition time.

The following shows the syntax of the cell degradation template.

```
lu_table_template(template_name) {  
    variable_1 : input_net_transition;  
    index_1 ("float, ..., float");  
}
```

The valid value for `variable_1` is `input_net_transition`.

The `index_1` values must be greater than or equal to 0.0 and follow the same rules for the lookup table template `index_1` attribute described in [Defining pin Groups on page 70](#). The number of floating-point numbers in `index_1` determines the size of the table dimension.

This is an example of a cell degradation template.

```
lu_table_template(deg_constraint) {  
    variable_1 : input_net_transition;  
    index_1 ("0.0, 1.0, 2.0");  
}
```

2. Use the `cell_degradation` group and the cell degradation template to create a one-dimensional lookup table for each timing arc in the cell. You receive warning messages if you define a `cell_degradation` construct for some, but not all, timing arcs in the cell.

The following example shows the `cell_degradation` group:

```
pin(output) {  
    timing() {  
        cell_degradation(deg_constraint) {  
            index_1 ("0.5, 1.5, 2.5");  
            values ("0.0, 2.0, 4.0");  
        }  
    }  
}
```

You can describe cell degradation groups only in the following types of `timing` groups:

- combinational

- three_state_enable
 - rising_edge
 - falling_edge
 - preset
 - clear
-

Describing Clocks

To define clocks and clocking, use these `pin` group attributes:

- `clock`
- `min_period`
- `min_pulse_width_high`
- `min_pulse_width_low`

clock Attribute

This attribute indicates whether or not an input pin is a clock pin.

A true value labels a pin as a clock pin. A false value labels a pin as not a clock pin, even though it might otherwise have such characteristics.

Example

```
clock : true ;
```

min_period Attribute

Place the `min_period` attribute on the clock pin of a flip-flop or a latch to specify the minimum clock period required for the input pin. The minimum period is the sum of the data arrival time and setup time. This time must be consistent with the `max_transition` time.

Example

```
min_period : 26.0;
```

min_pulse_width_high and min_pulse_width_low Attributes

Use these optional attributes to specify the minimum length of time a pin must remain at logic 1 (`min_pulse_width_high`) or logic 0 (`min_pulse_width_low`). These attributes can be placed on a clock input pin or an asynchronous clear/preset pin of a flip-flop or latch.

Note:

The Design Compiler tool does not support minimum pulse width as a constraint during synthesis.

Example

The following example shows both attributes on a clock pin, indicating the minimum pulse width for a clock pin.

```
pin(CLK) {  
    direction : input ;  
    capacitance : 1 ;  
    min_pulse_width_high : 3 ;  
    min_pulse_width_low : 3 ;  
}
```

Describing Clock Pin Functions

To define the function of a clock pin, use these `pin` group attributes:

- `function`
- `three_state`
- `x_function`
- `state_function`
- `internal_node`

function Attribute

The `function` attribute defines the value of an output or inout pin in terms of the cell's input or inout pins.

Syntax

```
function : "Boolean expression" ;
```

The precedence of the operators is left to right, with inversion performed first, then XOR, then AND, then OR.

The `function` attribute statement provides information for the Design Compiler and DFT Compiler tools. The Design Compiler tool uses this information when choosing components during synthesis. A cell without a `function` attribute is treated as a black box and is not synthesized or optimized.

If you want to prevent a cell from being used or replaced during optimization, use the `dont_use` attribute.

Note:

The DFT Compiler tool cannot test cells that do not include a `function` attribute or cells that cannot have a `function` attribute, such as shift registers and counters.

Table 6 Valid Boolean Operators in the function Attribute Statement

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

Grouped Pins in function Statements

Grouped pins can be used as variables in a `function` statement; see [Defining Bused Pins on page 100](#) and [Defining Signal Bundles on page 106](#). In `function` statements that use bus or bundle names, all the variables in the statements must be either a single pin or buses or bundles of the same width.

Ranges of buses or bundles are valid if the range you define contains the same number of members as the other buses or bundles in the same expression. You can reverse the bus order by listing the member numbers in reverse (high: low) order. Two buses, bundles, or bused-pin ranges with different widths should not appear in the same `function` statement.

When the `function` attribute of a cell with group input pins is a combinational-logic function of grouped variables only, the logic function is expanded to apply to each set of output grouped pins independently. For example, if A, B, and Z are defined as buses of the same width and the `function` statement for output Z is

```
function : "(A & B)" ;
```

the function for Z[0] is interpreted as

```
function : "(A[0] & B[0])" ;
```

Likewise, the function for Z[1] is interpreted as

```
function : "(A[1] & B[1])" ;
```

If a bus and a single pin are in the same `function` attribute, the single pin is distributed across all members of the bus. For example, if A and Z are buses of the same width, B is a single pin, and the function statement for the Z output is

```
function : "(A & B)" ;
```

The function for Z[0] is interpreted as

```
function : "(A[0] & B)" ;
```

Likewise, the function for Z[1] is interpreted as

```
function : "(A[1] & B)" ;
```

three_state Attribute

Use this attribute to define a three-state output pin in a cell.

Only the high impedance to logic 0 and high impedance to logic 1 timing arcs are used during three-state component synthesis and optimization.

In [Example 19](#), output pin Z is a three-state output pin. When input pin E (enable) goes low, pin Z goes to a high-impedance state. The value of the `three_state` attribute is, therefore, E'.

Example 19 Three-State Cell Description

```
library(example){  
    technology (cmos) ;  
    date : "May 14, 2002" ;  
    revision : 2002.05;  
    ...  
    cell(TRI_INV2) {  
        area : 3 ;  
        pin(A) {  
            direction : input ;  
            capacitance : 2 ;  
        }  
        pin(E) {  
            direction : input ;  
            capacitance : 2 ;  
        }  
        pin(Z) {  
            direction : output ;  
            function : "A'" ;  
        }  
    }  
}
```

```
    three_state : "E'" ;
    timing() {
        ...
    }
}
}
```

x_function Attribute

Use the `x_function` attribute to describe the X behavior of a pin, where X is a state other than 0, 1, or Z.

Note:

Only the Formality tool uses the `x_function` attribute.

The `three_state`, `function`, and `x_function` attributes are defined for output and inout pins and can have shared input. You can assign `three_state`, `function`, and `x_function` to be the function of the same input pins. When these functions have shared input, however, the cell must be inserted manually.

Also, when the values of more than one function equal 1, the three functions are evaluated in this order:

1. `x_function`
2. `three_state`
3. `function`

Example

```
pin (y) {
    direction: output;
    function : "!ap * !an" ;
    x_function : "!ap * an" ;
    three_state : "ap * !an" ;
}
```

state_function Attribute

Use this attribute to define output logic. Ports in the `state_function` Boolean expression must be either input, three-state inout, or ports with an `internal_node` attribute. If the output logic is a function of only the inputs (IN), the output is purely combinational (for example, feed-through output). A port in the `state_function` expression refers only to the non-three-state functional behavior of that port. An inout port in the `state_function` expression is treated only as an input port.

Example

```
state_function : Q*X;
```

internal_node Attribute

Use this attribute to resolve node names to real port names. The `internal_node` attribute describes the sequential behavior of an output pin. It provides the relationship between the `statetable` group and a pin of a cell. Each output with the `internal_node` attribute might also have the optional `input_map` attribute.

Example

```
internal_node : "Q";
```

Defining Bused Pins

To define bused pins, use these groups:

- `type` group
- `bus` group

You can use a defined bus or bus member in Boolean expressions in the `function` attribute. An output pin does not need to be defined in a cell before it is referenced.

type Group

If your library contains bused pins, you must define `type` groups and define the structural constraints of each bus type in the library.

The `type` group is defined at the `library` group level, as follows:

```
library (lib_name) {
    type ( name ) {
        ... type description ...
    }
}
```

name

Identifies the bus type.

A `type` group can one of the following:

base_type

Only the array base type is supported.

data_type

Only the bit data type is supported.

bit_width

An integer that designates the number of bus members. The default is 1.

bit_from

An integer indicating the member number assigned to the most significant bit (MSB) of successive array members. The default is 0.

bit_to

An integer indicating the member number assigned to the least significant bit (LSB) of successive array members. The default is 0.

downto

A value of true indicates that member number assignment is from high to low instead of low to high. The default is false (low to high).

Example 20 type Group Statement

```
type ( BUS4 ) {  
    base_type : array ;  
    data_type : bit ;  
    bit_width : 4 ;  
    bit_from : 0 ;  
    bit_to : 3 ;  
    downto :false ;  
}
```

It is not necessary to use all the `type` group attributes. For example, both the `type` group statements in [Example 21](#) are valid descriptions of `BUS4` of [Example 20](#).

Example 21 Alternative type Group Statements

```
type ( BUS4 ) {  
    base_type : array ;  
    data_type : bit ;  
    bit_width : 4 ;  
    bit_from : 0 ;  
    bit_to : 3 ;  
}  
type ( BUS4 ) {  
    base_type : array ;  
    data_type : bit ;  
    bit_width : 4 ;  
    bit_from : 3 ;  
    downto : true ;  
}
```

After you define a `type` group, you can use the `type` group in a `bus` group to describe bused pins.

bus Group

A `bus` group describes the characteristics of a bus. You define it in a `cell` group, as shown here:

```
library (lib_name) {  
    cell (cell_name) {  
        area : float ;  
        bus ( name ) {  
            ... bus description ...  
        }  
    }  
}
```

A `bus` group contains the following elements:

- `bus_type` attribute
- pin groups

In a `bus` group, use the number of bus members (pins) defined by the `bit_width` attribute in the applicable `type` group. You must declare the `bus_type` attribute first in the `bus` group.

bus_type Attribute

The `bus_type` attribute specifies the bus type. It is a required element of all `bus` groups. Always declare the `bus_type` as the first attribute in a `bus` group.

Note:

The bus type name must exist in a `type` group.

Syntax

```
bus_type : name ;
```

Pin Attributes and Groups

Pin attributes in a `bus` or `bundle` group specify default attribute values for all pins in that bus or bundle. Pin attributes can also appear in pin groups inside the `bus` or `bundle` group to define attribute values for specific bus or bundle pins or groups of pins. Values used in pin groups override the default attribute values defined for the bus or bundle.

All pin attributes are valid inside `bus` and `pin` groups. See [General pin Group Attributes on page 71](#) for a description of pin attributes. The `direction` attribute value of all bus members must be the same.

Use the full name of a pin for the names of pins in a `pin` group contained in a `bus` group.

The following example shows a `bus` group that defines bus A, with defaults for direction and capacitance assigned:

```
bus (A) {  
    bus_type : bus1 ;  
    direction : input ;  
    capacitance : 3 ;  
    ...  
}
```

The following example illustrates a `pin` group that defines a new `capacitance` attribute value for pin 0 in bus A:

```
pin (A[0]) {  
    capacitance : 4 ;  
}
```

You can also define pin groups for a range of bus members. A range of bus members is defined by a beginning value and an ending value, separated by a colon. No spaces can appear between the colon and the member numbers.

The following example illustrates a `pin` group that defines a new `capacitance` attribute value for bus members 0, 1, 2, and 3 in bus A:

```
pin (A[0:3]) {  
    capacitance : 4 ;  
}
```

For nonbused pins, you can identify member numbers as single numbers or as a range of numbers separated by a colon. Do not define member numbers in a list.

Table 7 Comparison of Bused and Single-Pin Formats

Pin type	Logic library
Bused Pin	pin x[3:0]
Single Pin	pin x

Example Bus Description

[Example 22](#) includes `type` and `bus` groups. It also shows the use of bus variables in `function`, `related_pin`, `pin_opposite`, and `pin_equal` attributes.

Example 22 A Complete Bus Description

```
library (ExamBus) {  
    date : "May 14, 2002";
```

Chapter 4: Defining Core Cells

Defining Bused Pins

```
revision : 2002.05;
bus_naming_style :"%s[%d]";/* Optional; this is the
                           default */
type (bus4) {
    base_type : array; /* Required */
    data_type : bit; /* Required if base_type is array */
    bit_width : 4; /* Optional; default is 1 */
    bit_from : 0; /* Optional MSB; defaults to 0 */
    bit_to : 3; /* Optional LSB; defaults to 0 */
    downto : false; /* Optional; defaults to false */
}
cell (bused_cell) {
    area : 10;
    bus (A) {
        bus_type : bus4;
        direction : input;
        capacitance : 3;
        pin (A[0:2]) {
            capacitance : 2;
        }
        pin (A[3]) {
            capacitance : 2.5;
        }
    }
    bus (B) {
        bus_type : bus4;
        direction : input;
        capacitance : 2;
    }
    pin (E) {
        direction : input ;
        capacitance 2 ;
    }
    bus (X) {
        bus_type : bus4;
        direction : output;
        capacitance : 1;
        pin (X[0:3]) {
            function : "A & B'";
            timing() {
                related_pin : "A B";
                /* A[0] and B[0] are related to X[0],
                   A[1] and B[1] are related to X[1], etc. */
            }
        }
    }
    bus (Y) {
        bus_type : bus4;
        direction : output;
        capacitance : 1;
        pin (Y[0:3]) {
            function : "B";
            three_state : "!E";
        }
    }
}
```

Chapter 4: Defining Core Cells

Defining Bused Pins

```
timing () {
    related_pin : "A[0:3] B E";
}
internal_power() {
    when: "E" ;
    related_pin : B ;
    power() {
        ...
    }
}
internal_power() {
    related_pin : B ;
    power() {
        ...
    }
}
bus (Z) {
    bus_type : bus4;
    direction : output;
    pin (Z[0:1]) {
        function : "!A[0:1]";
        timing () {
            related_pin : "A[0:1]";
        }
        internal_power() {
            related_pin : "A[0:1]";
            power() {
                ...
            }
        }
    }
    pin (Z[2]) {
        function "A[2]";
        timing () {
            related_pin : "A[2]";
        }
        internal_power() {
            related_pin : "A[0:1]";
            power() {
                ...
            }
        }
    }
    pin (Z[3]) {
        function : "!A[3]";
        timing () {
            related_pin : "A[3]";
        }
        internal_power() {
            related_pin : "A[0:1]";
            power() {
```

```
    ...
}
}
}
}
pin_opposite("Y[0:1]", "Z[0:1]");
/* Y[0] is opposite to Z[0], etc. */
pin_equal("Y[2:3] Z[2:3]");
/* Y[2], Y[3], Z[2], and Z[3] are equal */
cell (bused_cell2) {
    area : 20;
    bus (A) {
        bus_type : bus41;
        direction : input;
        capacitance : 1;
        pin (A[0:3]) {
            capacitance : 2;
        }
        pin (A[3]) {
            capacitance : 2.5;
        }
    }
    bus (B) {
        bus_type : bus4;
        direction : input;
        capacitance : 2;
    }
    pin (E) {
        direction : input ;
        capacitance 2 ;
    }
    bus(X) {
        bus_type : bus4;
        direction : output;
        capacitance : 1;
        pin (X[0:3]) {
            function : "A & B'";
            timing() {
                related_pin : "A B";
                /* A[0] and B[0] are related to X[0],
                A[1] and B[1] are related to X[1], etc. */
            }
        }
    }
}
```

Defining Signal Bundles

You need certain attributes to define a bundle. A bundle groups several pins that have similar timing or functionality. Bundles are used for multibit cells such as multibit latch, multibit flip-flop, and multibit AND gate.

bundle Group

Define a `bundle group` in a `cell group`, as shown:

```
library (lib_name) {  
    cell (cell_name) {  
        area : float ;  
        bundle ( name ) {  
            ... bundle description ...  
        }  
    }  
}
```

A `bundle group` contains the following elements:

`members` attribute

The `members` attribute must be declared first in a `bundle group`.

`pin` attributes

These include `direction`, `function`, and `three-state`.

members Attribute

The `members` attribute is used in a `bundle group` to list the pin names of the signals in a bundle. The `members` attribute must be included as the first attribute in the `bundle group`. It provides the bundle element names and groups a set of pins that have similar properties. The number of members defines the width of the bundle.

If a bundle has a `function` attribute defined for it, that function is copied to all bundle members. For example,

```
pin (C) {  
    direction : input ;  
    ...  
}  
bundle(A) {  
    members(A0, A1, A2, A3);  
    direction : output ;  
    function : "B' + C";  
    ...  
}  
bundle(B) {  
    members(B0, B1, B2, B3);  
    direction : input;  
    ...  
}
```

means that the members of the A bundle have these values:

```
A0 = B0' + C;  
A1 = B1' + C;  
A2 = B2' + C;  
A3 = B3' + C;
```

Each bundle operand (B) must have the same width as the function parent bundle (A).

pin Attributes

For information about pin attributes, see [General pin Group Attributes on page 71](#).

Example 23 Multibit Latch With Signal Bundles (bundle Group Syntax)

```
cell (latch4) {  
    area: 16;  
    pin (G) { /* active-high gate enable signal */  
        direction : input;  
        :  
    }  
    bundle (D) { /* data input with four member pins */  
        members(D1, D2, D3, D4); /*must be first attribute */  
        direction : input;  
    }  
    bundle (Q) {  
        members(Q1, Q2, Q3, Q4);  
        direction : output;  
        function : "IQ" ;  
    }  
    bundle (QN) {  
        members (Q1N, Q2N, Q3N, Q4N);  
        direction : output;  
        function : "IQN";  
    }  
    latch_bank(IQ, IQN, 4) {  
        enable : "G" ;  
        data_in : "D" ;  
    }  
}  
  
cell (latch5) {  
    area: 32;  
    pin (G) { /* active-high gate enable signal */  
        direction : input;  
        :  
    }  
    bundle (D) { /* data input with four member pins */  
        members(D1, D2, D3, D4); /*must be first attribute */  
        direction : input;  
    }  
    bundle (Q) {  
        members(Q1, Q2, Q3, Q4);  
        direction : output;  
        function : "IQ" ;  
    }  
    bundle (QN) {
```

```
members (Q1N, Q2N, Q3N, Q4N);
direction : output;
function : "IQN";
}
latch_bank(IQ, IQN, 4) {
    enable : "G";
    data_in : "D";
}
}
```

Defining Layout-Related Multibit Attributes

The `single_bit_degenerate` attribute is a layout-related attribute for multibit cells. The attribute also applies to sequential and combinational cells.

The `single_bit_degenerate` attribute is for use on multibit bundle or bus cells that are black boxes. The value of this attribute is the name of a single-bit library cell.

Example 24 Multibit Cells With `single_bit_degenerate` Attribute

```
cell (FDX2) {
    area : 18 ;
    single_bit_degenerate : FDB ;
    bundle (D) {
        members (D0, D1) ;
        direction : input ;
        ...
        timing () {
            ...
            ...
        }
    }
}

cell (FDX4)
area : 18 ;
single_bit_degenerate : FDB ;
bus (D) {
    bus_type : bus4 ;
    direction : input ;
    ...
    timing () {
        ...
        ...
    }
}
```

The library description does not include information such as cell height; this must be provided by the library developer.

Defining Multiplexers

A one-hot MUX is a library cell that behaves functionally as a regular MUX logic gate. However, in the case of a one-hot MUX, some inputs are considered dedicated control inputs and others are considered dedicated data inputs. There are as many control inputs as data inputs, and the function of the cell is the logic AND of the i_{th} control input with the i_{th} data input. For example, a 4-to-1 one-hot MUX has the following function:

$$Z = (D_0 \& C_0) | (D_1 \& C_1) | (D_2 \& C_2) | (D_3 \& C_3)$$

One-hot MUXs are generally implemented using pass gates, which makes them very fast and allows their speed to be largely independent of the number of data bits being multiplexed. However, this implementation requires that exactly one control input be active at a time. If no control inputs are active, the output is left floating. If more than one control input is active, there could be an internal drive fight.

Library Requirements

One-hot MUX library cells must meet the following requirements:

- A one-hot MUX cell in the target library should be a single-output cell.
- Its inputs can be divided into two disjoint sets of the same size as follows:

$$C=\{C_1, C_2, \dots, C_n\} \text{ and } D=\{D_1, D_2, \dots, D_n\}$$

where n is greater than 1 and is the size of the set. Actual names of the inputs can vary.

- The `contention_condition` attribute must be set on the cell. The value of the attribute is a combinational function, $f\{C\}$, of inputs in set C that defines prohibited combinations of inputs as shown in the following examples (where size n of the set is 3):

$$FC = C_0' \& C_1' \& C_2' | C_0 \& C_1 | C_0 \& C_2 | C_1 \& C_2$$

or

$$FC = (C_0 \& C_1' \& C_2' | C_0' \& C_1 \& C_2' | C_0' \& C_1' \& C_2)'$$

- The cell must have a combinational function FO defined on the output with respect to all its inputs. This function FO must logically define, together with the contention condition, a base function F^* that is a sum of n product terms, where the i_{th} term contains all the inputs in C , with C_i high and all others low and exclusively one input in D .

Chapter 4: Defining Core Cells

Defining Multiplexers

Examples of the defined function are as follows (for n = 3):

$F^* = C_0 \cdot C_1' \cdot C_2' \cdot D_0 + C_0' \cdot C_1 \cdot C_2' \cdot D_1 + C_0' \cdot C_1' \cdot$
 \cdot

or

$F^* = C_0 \cdot C_1' \cdot C_2' \cdot D_0' + C_0' \cdot C_1 \cdot C_2' \cdot D_1' + C_0' \cdot$
 $C_1' \cdot$
 \cdot
 $C_2 \cdot D_2'$

The function FO can take many forms, if it satisfies the following condition:

$FO \cdot FC' == F^*$

when FO is restricted by FC', it should be equivalent to F^* . The term $FO = F^*$ is acceptable; other examples are as follows (for n = 3):

$FO = (D_0 \cdot C_0) + (D_1 \cdot C_1) + (D_2 \cdot C_2)$

or

$FO = (D_0' \cdot C_0) + (D_1' \cdot C_1) + (D_2' \cdot C_2)$

Note:

When FO is restricted by FC, inverting all inputs in D is equivalent to inverting the output. Inverting only a subset of D yields an incompatible function. It is recommended that you use the simple form described earlier, or F^* .

The following example shows a cell that is properly specified.

Example

```
cell(one_hot_mux_example) {  
    ... ...  
    contention_condition : "(C0 C1 + C0' C1')";  
    ... ...  
    pin(D0) {  
        direction : input;  
        ... ...  
    }  
    pin(D1) {  
        direction : input;  
        ... ...  
    }  
    pin(C0) {  
        direction : input;  
        ... ...  
    }  
    pin(C1) {
```

```
    direction : input;
    ...
}
pin(Z) {
    direction : output;
    function : "(C0 D0 + C1 D1)";
    ...
}
}
```

Defining Decoupling Capacitor Cells, Filler Cells, and Tap Cells

Decoupling capacitor cells, or *decap cells*, are cells that have a capacitor placed between the power rail and the ground rail to overcome dynamic voltage drop; filler cells are used to connect the gaps between the cells after placement; and tap cells are physical-only cells that have power and ground pins and do not have signal pins. Cell-level attributes that identify decoupling capacitor cells, filler cells, and tap cells in libraries are supported.

Syntax

The following syntax shows a cell with the `is_decap_cell`, `is_filler_cell` and `is_tap_cell` attributes, which identify decoupling capacitor cells, filler cells, and tap cells, respectively. However, only one attribute can be set to true in a given cell.

```
cell (cell_name) {
    ...
    is_decap_cell : true | false;
    is_filler_cell : true | false;
    is_tap_cell : true | false;
}
/* End cell group */
```

Cell-Level Attributes

The following attributes can be set at the cell level to identify decoupling capacitor cells, filler cells, and tap cells.

is_decap_cell

The `is_decap_cell` attribute identifies a cell as a decoupling cell. Valid values are true and false.

is_filler_cell

The `is_filler_cell` attribute identifies a cell as a filler cell. Valid values are true and false.

is_tap_cell

The `is_tap_cell` attribute identifies a cell as a tap cell. Tap cells are physical-only cells, which means they have power and ground pins only and not signal pins. Tap cells are well-tied cells that bias the silicon infrastructure of n-wells or p-wells. Valid values for the `is_tap_cell` attribute are true and false.

Fault Tolerant Cell Modeling

Fault tolerance is a requirement of safe automotive systems. Fault-tolerant library cell models ensure that fault-tolerant cell designs are honored by downstream tools.

In IC designs, registers contribute to most of the soft errors. Soft errors affect the data state of sequential elements and are caused by random environmental radiation events.

Soft error rate (SER) is the rate at which a library cell is predicted to encounter soft errors. SER is expressed as the number of failures-in-time (FIT) per million library cells, where FIT is equivalent to one error per billion hours of device operation. SER is affected by changes in process, voltage, temperature, altitude, frequency, and confidence.

Fault Tolerant Modeling Syntax

```
library (library_name) {  
    ...  
    altitude_unit : enum_value ;  
  
    soft_error_rate_template (temp_name) {  
        variable_1 : [ altitude | frequency ] ;  
        variable_2 : [ altitude | frequency ] ;  
        index_1 ( ... ) ;  
        index_2 ( ... ) ;  
    }  
    default_soft_error_rate (temp_name) {  
        index_1 ( ... ) ;  
        index_2 ( ... ) ;  
        values ( ... ) ;  
    }  
    soft_error_rate_confidence : float ;  
  
    cell (cell_name) {  
        soft_error_rate (temp_name) {  
            index_1 ( ... ) ;  
            index_2 ( ... ) ;  
            values ( ... ) ;  
        }  
        ...  
    } /* end cell group */  
    ...  
} /* end library group */
```

Library-Level Attributes and Groups

Fault tolerant cell modeling contains the following library-level attributes and groups.

altitude_unit Attribute

The `altitude_unit` attribute specifies the unit of altitude in the library. Valid values are 1 m (default) or 1 km.

soft_error_rate_template Group

The `soft_error_rate_template` group defines the table template for the `default_soft_error_rate` and the `soft_error_rate` group. The template has two indexes, altitude and frequency. The template table can be one-dimensional or two-dimensional.

The template definition includes two one-dimensional variables, `variable_1` and `variable_2`, with values, altitude and frequency, respectively.

The `index_1` attribute specifies the `altitude` values at which the cell SER is sampled in the library. The unit is defined by the library-level `altitude_unit` attribute.

The `index_2` attribute specifies the `frequency` values at which the cell SER is sampled in the library. The values must be greater than or equal to zero. The unit is defined by the library-level `time_attribute` attribute, as the frequency unit is reciprocal of the time unit.

default_soft_error_rate Group

The `default_soft_error_rate` group is a lookup table that specifies the default cell SER in the library. The table values must be greater than or equal to zero.

The default table applies to cells in libraries where the cell-level `soft_error_rate` table is not defined. The table can be one-dimensional or two-dimensional. It is recommended to specify the library-level `default_soft_error_rate` table even for libraries that contain a cell-level `soft_error_rate` table.

soft_error_rate_confidence Attribute

The `soft_error_rate_confidence` attribute specifies the confidence level at which the cell SER is sampled in the library. The value range is from 0.0 to 1.0.

Cell-Level Attributes and Groups

Fault tolerant modeling contains the following cell-level group.

soft_error_rate Group

The `soft_error_rate` group is a lookup table that specifies the SER for the cell. The table values must be greater than or equal to zero. You can specify this table for all cell types. The cell-level values override the library-level `default_soft_error_rate` values. The table can be one-dimensional or two-dimensional.

Fault Tolerant Library Modeling Example

```
library(mylib) {  
    ...  
    time_unit : 1ns ;  
    altitude_unit : 1km ;  
  
    soft_error_rate_template ( ser_temp ) {  
        variable_1 : altitude ;  
        variable_2 : frequency ;  
        index_1 ( "1.6, 1.8" ) ; /* altitude 1.6 km, 1.8 km */  
        index_2 ( "0.1, 0.2" ) ; /* frequency 100 MHz, 200 MHz */  
    }  
  
    default_soft_error_rate ( ser_temp ) {  
        index_1 ( "1.6, 1.8" ) ; /* altitude 1.6 km, 1.8 km */  
        index_2 ( "0.1, 0.2" ) ; /* frequency 100 MHz, 200 MHz */  
  
        /* 3.5 soft errors per 1 billion hours of operation */  
        values ( "3.5, 3.6, 4.5, 4.6" ) ;  
    }  
  
    soft_error_rate_confidence : 0.5 ;/* confidence level 50% */  
  
    cell(mycell) {  
        ...  
        soft_error_rate ( ser_temp ) {  
            index_1 ( "1.6, 1.8" ) ; /* altitude 1.6 km, 1.8 km */  
            index_2 ( "0.1, 0.2" ) ; /* frequency 100 MHz, 200 MHz */  
  
            /* 3.55 soft error per 1 billion hours of operation */  
            values ( "3.55, 3.65, 4.55, 4.65" ) ;  
        }  
    } /* end cell group */  
} /* end library group */
```

5

Defining Sequential Cells

This chapter describes the peculiarities of defining flip-flops and latches, building upon the cell description syntax given in [Chapter 4, Defining Core Cells](#).” It describes group statements that apply only to sequential cells and also describes a variation of the `function` attribute that makes use of state variables.

To design flip-flops and latches, you must understand the following concepts and tasks:

- [Using Sequential Cell Syntax](#)
- [Describing a Flip-Flop](#)
- [Using the function Attribute](#)
- [Describing a Multibit Flip-Flop](#)
- [Describing a Latch](#)
- [Describing a Multibit Latch](#)
- [Describing Sequential Cells With the Statetable Format](#)
- [Flip-Flop and Latch Examples](#)
- [Cell Description Examples](#)

Using Sequential Cell Syntax

You can describe sequential cells with the following cell definition formats:

- ff or latch format

Cells using the ff or latch format are identified by the `ff` group and `latch` group.

- statetable format

Cells using the statetable format are identified by the `statetable` group. The statetable format supports all the sequential cells supported by the ff or latch format. In addition, the statetable format supports complex sequential cells, such as the following:

- Sequential cells with multiple clock ports, such as a cell with a system clock and a test scan clock

- internal state sequential cells, such as master-slave cells
- Multistate sequential cells, such as counters and shift registers
- Sequential cells with combinational outputs
- Sequential cells with complex clocking and complex asynchronous behavior
- Sequential cells with multiple simultaneous input transitions
- Sequential cells with illegal input conditions

The statetable format contains a complete, expanded set of table rules for which all L and H permutations of table input are explicitly specified.

Some cells cannot be modeled with the statetable format. For example, you cannot use the statetable format to model a cell whose function depends on differential clocks when the inputs change.

The format you use depends on the tool for which you design the library.

- Design synthesis, test synthesis, and fault simulation tools support only the ff or latch format.

Note:

The Design Compiler tool supports the ff or latch format, and ASIC vendors should continue to use ff or latch format for all sequential cells that are already supported. However, the statetable syntax is more intuitive and easier to use than the current sequential modeling format. Therefore, ASIC vendors should add the statetable format to all their sequential cells, to ensure automatic functional verification and compatibility.

- DFT Compiler supports the statetable format design rule checking (DRC) but requires ff or latch format for scan substitution. See [Chapter 6, Defining Test Cells](#), for information about modeling cells for test.

Describing a Flip-Flop

To describe an edge-triggered storage device, include a `ff` group or a `statetable` group in a cell definition. This section describes how to define a flip-flop by using the ff or latch format. See [Describing Sequential Cells With the Statetable Format on page 138](#) for the way to define cells using the `statetable` group.

Using the ff Group

A `ff` group describes either a single-stage or a master-slave flip-flop. The `ff_bank` group represents multibit registers, such as a bank of flip-flops. See [Describing a Multibit Flip-Flop on page 126](#) for more information about the `ff_bank` group.

Syntax

```
library (lib_name) {
  cell (cell_name) {
    ...
    ff ( variable1, variable2 ) {

      clocked_on : "Boolean_expression" ;
      next_state : "Boolean_expression" ;

      clear : "Boolean_expression" ;
      preset : "Boolean_expression" ;

      clear_preset_var1 : value ;
      clear_preset_var2 : value ;
      clocked_on_also : "Boolean_expression" ;
      power_down_function : "Boolean_expression" ;
    }
  }
}
```

variable1

The state of the noninverting output of the flip-flop. It is considered the 1-bit storage of the flip-flop.

variable2

The state of the inverting output

You can name *variable1* and *variable2* anything except the name of a pin in the cell being described. Both variables are required, even if one of them is not connected to a primary output pin.

The `clocked_on` and `next_state` attributes are required in the `ff` group; all other attributes are optional.

clocked_on and clocked_on_also Attributes

The `clocked_on` and `clocked_on_also` attributes identify the active edge of the clock signals.

Single-state flip-flops use only the `clocked_on` attribute. When you describe flip-flops that require both a master and a slave clock, use the `clocked_on` attribute for the master clock and the `clocked_on_also` attribute for the slave clock.

Examples

A rising-edge-triggered device is:

```
clocked_on : "CP";
```

A falling-edge-triggered device is:

```
clocked_on : "CP'";
```

next_state Attribute

The `next_state` attribute is a logic equation written in terms of the cell's input pins or the first state variable, *variable1*. For single-stage storage elements, the `next_state` attribute equation determines the value of *variable1* at the next active transition of the `clocked_on` attribute.

For devices such as a master-slave flip-flop, the `next_state` attribute equation determines the value of the master stage's output signals at the next active transition of the `clocked_on` attribute.

Example

```
next_state : "D";
```

nextstate_type Attribute

The `nextstate_type` attribute is a pin group attribute that defines the type of `next_state` attribute used in the `ff` or `ff_bank` group.

Any pin with the `nextstate_type` attribute must be included in the value of the `next_state` attribute.

Note:

Specify a `nextstate_type` attribute to ensure that the sync set (or sync reset) pin and the D pin of sequential cells are not swapped when instantiated.

Example

```
nextstate_type : data;
```

The example in [Using the function Attribute](#) and [Describing a Multibit Flip-Flop](#) show the use of the `nextstate_type` attribute.

clear Attribute

The `clear` attribute gives the active value for the clear input.

The example defines an active-low clear signal.

Example

```
clear : "CD'" ;
```

For more information about the `clear` attribute, see [Describing a Single-Stage Flip-Flop on page 122](#).

preset Attribute

The `preset` attribute gives the active value for the preset input.

The example defines an active-high preset signal.

Example

```
preset : "PD'" ;
```

For more information about the `preset` attribute, see [Describing a Single-Stage Flip-Flop on page 122](#).

clear_preset_var1 Attribute

The `clear_preset_var1` attribute gives the value that `variable1` has when `clear` and `preset` are both active at the same time.

Example

```
clear_preset_var1 : L;
```

For more information about the `clear_preset_var1` attribute, including its function and values, see [Describing a Single-Stage Flip-Flop on page 122](#).

clear_preset_var2 Attribute

The `clear_preset_var2` attribute gives the value that `variable2` has when `clear` and `preset` are both active at the same time.

Example

```
clear_preset_var2 : L ;
```

For more information about the `clear_preset_var2` attribute, including its function and values, see [Describing a Single-Stage Flip-Flop on page 122](#).

power_down_function Attribute

The `power_down_function` attribute specifies the Boolean condition when the cell's output pin is switched off by the power and ground pins (when the cell is in off mode due to the external power pin states).

You specify the `power_down_function` attribute for combinational and sequential cells. For simple or complex sequential cells, the `power_down_function` attribute also determines the condition of the cell's internal state.

Syntax

```
library (name) {
    cell (name) {
        ff (variable1,variable2) {
            //...flip-flop description...
            clear : "Boolean expression" ;
            clear_preset_var1 : L | H | N | T | X ;
            clear_preset_var2 : L | H | N | T | X ;
            clocked_on : "Boolean expression" ;
            clocked_on_also : "Boolean expression" ;
            next_state : "Boolean expression" ;
            preset : "Boolean expression" ;
            power_down_function : "Boolean expression" ;
        }
        ...
    }
}
```

Example

```
library ("low_power_cells") {
    cell ("retention_dff") {
        pg_pin(VDD) {
            voltage_name : VDD;
            pg_type : primary_power;
        }
        pg_pin(VSS) {
            voltage_name : VSS;
            pg_type : primary_ground;
        }
        pin ("D") {
            direction : "input";
        }
        pin ("CP") {
            direction : "input";
        }
        ff(IQ,IQN) {
            next_state : "D" ;
            clocked_on : "CP" ;
            power_down_function : "!VDD + VSS" ;
        }
    }
}
```

```

        }
        pin ("Q") {
            function : " IQ ";
            direction : "output";
            power_down_function : "!VDD + VSS";
        }
        ...
    }
...
}
```

ff Group Examples

The following is an example of the `ff` group for a single-stage D flip-flop.

```
ff(IQ, IQN) {
    next_state : "D" ;
    clocked_on : "CP" ;
}
```

The example defines two variables, IQ and IQN. The `next_state` attribute determines the value of IQ after the next active transition of the `clocked_on` attribute. In the example, IQ is assigned the value of the D input.

For some flip-flops, the next state depends on the current state. In this case, the first state variable, `variable1` (IQ in the example), is used in the `next_state` statement; and the second state variable, IQN, is not used.

For the example, the `ff` attribute group for a JK flip-flop is,

```
ff(IQ, IQN) {
    next_state : "(J K IQ') + (J K') + (J' K' IQ)" ;
    clocked_on : "CP";
}
```

The `next_state` and `clocked_on` attributes define the synchronous behavior of the flip-flop.

Describing a Single-Stage Flip-Flop

A single-stage flip-flop does not use the optional `clocked_on_also` attribute.

Table 8 Functions of the Attributes in the ff Group for a Single-Stage Flip-Flop

active_edge	clear	preset	variable1	variable2
clocked_on	inactive	inactive	next_state	!next_state
--	active	inactive	0	1

Chapter 5: Defining Sequential Cells
Describing a Flip-Flop

active_edge	clear	preset	variable1	variable2
--	inactive	active	1	0
--	active	active	clear_preset_var1	clear_preset_var2

The `clear` attribute gives the active value for the clear input. The `preset` attribute gives the active value for the preset input. For example, the following statement defines an active-low clear signal.

```
clear : "CD'" ;
```

The `clear_preset_var1` and `clear_preset_var2` attributes specify the value for `variable1` and `variable2` when `clear` and `preset` are both active at the same time.

Table 9 Valid Values for the clear_preset_var1 and clear_preset_var2 Attributes

Variable values	Equivalence
L	0
H	1
N	No change
T	Toggle the current value from 1 to 0, 0 to 1, or X to X
X	Unknown

If you use both `clear` and `preset`, you must use either `clear_preset_var1`, `clear_preset_var2`, or both. Conversely, if you include `clear_preset_var1`, `clear_preset_var2`, or both, you must use both `clear` and `preset`.

The flip-flop cell is activated whenever `clear`, `preset`, `clocked_on`, or `clocked_on_also` change.

[Example 25](#) is a `ff` group for a single-stage D flip-flop with a rising edge, negative clear and preset, and the output pins set to 0 when both `clear` and `preset` are active (low).

Example 25 Single-Stage D Flip-Flop

```
ff(IQ, IQN) {
    next_state : "D" ;
    clocked_on : "CP" ;
    clear : "CD'" ;
    preset : "PD'" ;
```

Chapter 5: Defining Sequential Cells

Describing a Flip-Flop

```

    clear_preset_var1 : L ;
    clear_preset_var2 : L ;
}

```

Example 26 is a `ff` group for a single-stage, rising edge-triggered JK flip-flop with scan input, negative clear and preset, and the output pins set to 0 when `clear` and `preset` are both active.

Example 26 Single-Stage JK Flip-Flop

```

ff(IQ, IQN) {
    next_state :"(TE*TI)+(TE'*J*K')+(TE'*J'*K'*IQ)+(TE'*J*K*IQ' ) " ;
    clocked_on : "CP" ;
    clear : "CD'" ;
    preset : "PD'" ;
    clear_preset_var1 : L ;
    clear_preset_var2 : L ;
}

```

Example 27 is a `ff` group for a D flip-flop with synchronous negative clear.

Example 27 D Flip-Flop With Synchronous Negative Clear

```

ff(IQ, IQN) {
    next_state : "D * CLR" ;
    clocked_on : "CP" ;
}

```

Describing a Master-Slave Flip-Flop

The specification for a master-slave flip-flop is the same as for a single-stage device, except that it includes the `clocked_on_also` attribute.

Table 10 Functions of Attributes of ff Group for a Master-Slave Flip-Flop

active_edge	clear	preset	internal1	internal2	variable1	variable2
clocked_on	inactive	inactive	next_state	!next_state		
clocked_on_also	inactive	inactive			internal1	internal2
	active	active	clear_preset_var1	clear_preset_var2	clear_preset_var1	clear_preset_var2
	active	inactive	0	1	0	1
	inactive	active	1	0	1	0

The `internal1` and `internal2` variables represent the output values of the master stage, and `variable1` and `variable2` represent the output values of the slave stage.

The `internal1` and `internal2` variables have the same value as `variable1` and `variable2`, respectively, when the `clear` and `preset` attributes are both active at the same time.

Note:

You do not need to specify the `internal1` and `internal2` variables, which represent internal stages in the flip-flop.

Example 28 shows the `ff` group for a master-slave D flip-flop with a rising-edge sampling, falling-edge data transfer, negative clear and preset, and output values set to high when the `clear` and `preset` attributes are both active.

Example 28 Master-Slave D Flip-Flop

```
ff(IQ, IQN) {
    next_state : "D" ;
    clocked_on : "CLK" ;
    clocked_on_also : "CLKN'" ;
    clear : "CDN'" ;
    preset : "PDN'" ;
    clear_preset_var1 : H ;
    clear_preset_var2 : H ;
}
```

Using the function Attribute

Each storage device output pin needs a `function` attribute statement. Only the two state variables, `variable1` and `variable2`, can be used in the `function` attribute statement for sequentially modeled elements.

Example 29 Complete Functional Description of a Rising-Edge-Triggered D Flip-Flop With Active-Low Clear and Preset

```
cell (dff) {
    area : 1 ;
    pin (CLK) {
        direction : input ;
        capacitance : 0 ;
    }
    pin (D) {
        nextstate_type : data;
        direction : input ;
        capacitance : 0 ;
    }
    pin (CLR) {
        direction : input ;
    }
}
```

```
    capacitance : 0 ;
}
pin (PRE) {
    direction : input ;
    capacitance : 0 ;
}
ff (IQ, IQN) {
    next_state : "D" ;
    clocked_on : "CLK" ;
    clear : "CLR'" ;
    preset : "PRE'" ;
    clear_preset_var1 : L ;
    clear_preset_var2 : L ;
}
pin (Q) {
    direction : output ;
    function : "IQ" ;
}
pin (QN) {
    direction : output ;
    function : "IQN" ;
}
} /* end of cell dff */
```

Flip-flops that have the `function` attribute statements can be

- Inferred from a state machine description or a VHDL or Verilog description
- Translated to flip-flops in a different logic library
- Sized during timing optimization
- Changed to flip-flops of a different type, such as D to JK, or D with preset to D with clear
- Converted to scan cells by DFT Compiler with the `insert_dft` command

Describing a Multibit Flip-Flop

The `ff_bank` group describes a cell that is a collection of parallel, single-bit sequential parts. Each part shares control signals with the other parts and performs an identical function. The `ff_bank` group is typically used to represent multibit registers. It can be used in `cell` and `test_cell` groups.

The syntax is similar to that of the `ff` group; see [Describing a Flip-Flop on page 117](#).

Syntax

```
library (lib_name)
{
    cell (cell_name) {
```

Chapter 5: Defining Sequential Cells

Describing a Multibit Flip-Flop

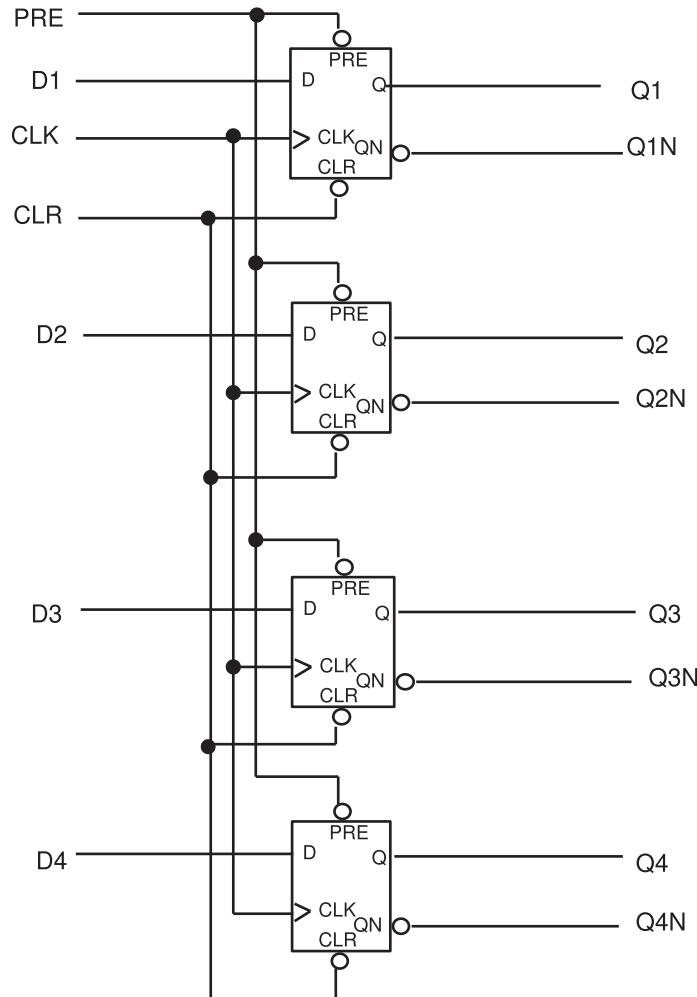
```
...
pin (pin_name) {
...
}
bundle (bundle_name) {
...
}
ff_bank ( variable1, variable2, bits ) {
    clocked_on : "Boolean_expression" ;
    next_state : "Boolean_expression" ;
    clear : "Boolean_expression" ;
    preset : "Boolean_expression" ;
    clear_preset_var1 : value ;
    clear_preset_var2 : value ;
    clocked_on_also : "Boolean_expression" ;
}
}
```

An input that is described in a `pin` group, such as the `clk` input, is fanned out to each flip-flop in the bank. Each primary output must be described in a `bus` or `bundle` group, and its function statement must include either `variable1` or `variable2`.

Three-state output pins are allowed; you can add a `three_state` attribute to an output bus or bundle to specify this function.

The `bits` value in the `ff_bank` definition is the number of bits in this multibit cell.

Figure 7 4-Bit Register With Rising-Edge-Triggered D Flip-Flops Having Clear and Preset



[Example 30](#) is the description of the multibit register shown in [Figure 7](#).

[Example 30 4-Bit Register Modeled Using ff_bank Group for Rising-Edge-Triggered D Flip-Flops](#)

```
cell (dff4) {
    area : 1 ;
    pin (CLK) {
        direction : input ;
        capacitance : 0 ;
        min_pulse_width_low : 3 ;
        min_pulse_width_high : 3 ;
    }
}
```

Chapter 5: Defining Sequential Cells

Describing a Multibit Flip-Flop

```
bundle (D) {
    members(D1, D2, D3, D4);
    nextstate_type : data;
    direction : input ;
    capacitance : 0 ;
    timing() {
        related_pin      : "CLK" ;
        timing_type      : setup_rising ;
    }
    timing() {
        related_pin      : "CLK" ;
        timing_type      : hold_rising ;
    }
}
pin (CLR) {
    direction : input ;
    capacitance : 0 ;
    timing() {
        related_pin      : "CLK" ;
        timing_type      : recovery_rising ;
    }
}
pin (PRE) {
    direction : input ;
    capacitance : 0 ;
    timing() {
        related_pin      : "CLK" ;
        timing_type      : recovery_rising ;
    }
}
ff_bank (IQ, IQN, 4) {
    next_state : "D" ;
    clocked_on : "CLK" ;
    clear : "CLR'" ;
    preset : "PRE'" ;
    clear_preset_var1 : L ;
    clear_preset_var2 : L ;
}
bundle (Q) {
    members(Q1, Q2, Q3, Q4);
    direction : output ;
    function : "(IQ)" ;
    timing() {
        related_pin      : "CLK" ;
        timing_type      : rising_edge ;
    }
    timing() {
        related_pin      : "PRE" ;
        timing_type      : preset ;
        timing_sense     : negative_unate ;
    }
    timing() {
        related_pin      : "CLR" ;
    }
}
```

```
        timing_type      : clear ;
        timing_sense    : positive_unate ;
    }
}
bundle (QN) {
    members(Q1N, Q2N, Q3N, Q4N);
    direction : output ;
    function : "IQN" ;
    timing() {
        related_pin      : "CLK" ;
        timing_type      : rising_edge ;
    }
    timing() {
        related_pin      : "PRE" ;
        timing_type      : clear ;
        timing_sense    : positive_unate ;
    }
    timing() {
        related_pin      : "CLR" ;
        timing_type      : preset ;
        timing_sense    : negative_unate ;
    }
}
} /* end of cell dff4 */
```

Describing a Latch

To describe a level-sensitive storage device, you include a `latch` group or `statetable` group in the cell definition. This section describes how to define a latch by using the `ff` or `latch` format. See [Describing Sequential Cells With the Statetable Format on page 138](#) for information about defining cells using the `statetable` group.

latch Group

This section describes a level-sensitive storage device found within a `cell` group.

Syntax

```
library (lib_name) {
    cell (cell_name) {
        ...
        latch (variable1, variable2) {
            enable : "Boolean_expression" ;
            data_in : "Boolean_expression" ;
            clear : "Boolean_expression" ;
            preset : "Boolean_expression" ;
            clear_preset_var1 : value ;
            clear_preset_var2 : value ;
        }
        pin (pin_name) {
```

```
    ...
    data_in_type : data | preset | clear | load ;
}
}
}
```

variable1

The state of the noninverting output of the latch. It is considered the 1-bit storage of the latch.

variable2

The state of the inverting output.

You can name **variable1** and **variable2** anything except the name of a pin in the cell being described. Both variables are required, even if one of them is not connected to a primary output pin.

If you include both `clear` and `preset`, you must use either `clear_preset_var1`, `clear_preset_var2`, or both. Conversely, if you include `clear_preset_var1`, `clear_preset_var2`, or both, you must use both `clear` and `preset`.

enable and data_in Attributes

The `enable` and `data_in` attributes are optional, but if you use one of them, you must include the other. The `enable` attribute gives the state of the enable input, and the `data_in` attribute gives the state of the data input.

Example

```
enable : "G" ;
data_in : "D";
```

data_in_type Attribute

In a pin group, the `data_in_type` attribute specifies the type of input data defined by the `data_in` attribute. The valid values are `data`, `preset`, `clear`, or `load`. The default is `data`.

Note:

The Boolean expression of the `data_in` attribute must include the pin with the `data_in_type` attribute.

Example

```
data_in_type : data ;
```

clear Attribute

The `clear` attribute gives the active value for the clear input.

Example

This example defines an active-low clear signal.

```
clear : "CD'" ;
```

preset Attribute

The `preset` attribute gives the active value for the preset input.

Example

This example defines an active-low preset signal.

```
preset : "R'" ;
```

clear_preset_var1 Attribute

The `clear_preset_var1` attribute gives the value that *variable1* has when `clear` and `preset` are both active at the same time. Valid values are shown in [Table 11](#).

Example

```
clear_preset_var1 : L;
```

Table 11 Valid Values for the `clear_preset_var1` and `clear_preset_var2` Attributes

Variable values	Equivalence
L	0
H	1
N	No change
T	Toggle the current value from 1 to 0, 0 to 1, or X to X
X	Unknown

clear_preset_var2 Attribute

The `clear_preset_var2` attribute gives the value that *variable2* has when `clear` and `preset` are both active at the same time. Valid values are shown in [Table 11](#).

Example

```
clear_preset_var2 : L ;
```

Table 12 Functions of the Attributes in the latch Group

enable	clear	preset	variable1	variable2
active	inactive	inactive	data_in	!data_in
--	active	inactive	0	1
--	inactive	active	1	0
--	active	active	clear_preset_var1	clear_preset_var2

The latch cell is activated whenever `clear`, `preset`, `enable`, or `data_in` changes.

Example 31 D Latch With Active-High enable and Negative clear

```
latch(IQ, IQN) {
    enable : "G" ;
    data_in : "D" ;
    clear : "CD'" ;
}
```

The `enable` and `data_in` attributes are not required to model an SR latch, as shown in [Example 32](#).

Example 32 SR Latch

```
latch(IQ, IQN) {
    clear : "S'" ;
    preset : "R'" ;
    clear_preset_var1 : L ;
    clear_preset_var2 : L ;
}
```

Determining a Latch Cell's Internal State

You can use the `power_down_function` attribute to specify the Boolean condition under which the cell's output pin is switched off by the state of the power and ground pins (when the cell is in off mode due to the external power pin states). The attribute should be set under the `latch` group. The `power_down_function` attribute also determines the condition of the cell's internal state.

For more information about `power_down_function`, see [power_down_function Attribute on page 121](#).

power_down_function Syntax For Latch Cells

```
library (name) {
    cell (name) {
        latch (variable1,variable2) {
```

```
//...latch description...
clear : "Boolean expression" ;
clear_preset_var1 : L | H | N | T | X ;
clear_preset_var2 : L | H | N | T | X ;
data_in : "Boolean expression" ;
enable : "Boolean expression" ;
preset : "Boolean expression" ;
power_down_function : "Boolean expression" ;
}
...
}
...
}
```

Describing a Multibit Latch

The `latch_bank` group describes a cell that is a collection of parallel, single-bit sequential parts. Each part shares control signals with the other parts and performs an identical function. The `latch_bank` group is typically used in `cell` and `test_cell` groups to represent multibit registers.

latch_bank Group

The syntax is similar to that of the `latch` group. See [Describing a Latch on page 130](#).

Syntax

```
library (lib_name) {
  cell (cell_name) {
    ...
    pin (pin_name) {
      ...
    }
    bundle (bus_name) {
      ...
    }
    latch_bank (variable1, variable2, bits) {
      enable : "Boolean_expression" ;
      data_in : "Boolean_expression" ;
      clear : "Boolean_expression" ;
      preset : "Boolean_expression" ;
      clear_preset_var1 : value ;
      clear_preset_var2 : value ;
    }
  }
}
```

Chapter 5: Defining Sequential Cells

Describing a Multibit Latch

An input that is described in a `pin` group, such as the `clk` input, is fanned out to each latch in the bank. Each primary output must be described in a `bus` or `bundle` group, and its function statement must include either `variable1` or `variable2`.

Three-state output pins are allowed; you can add a `three_state` attribute to an output bus or bundle to define this function.

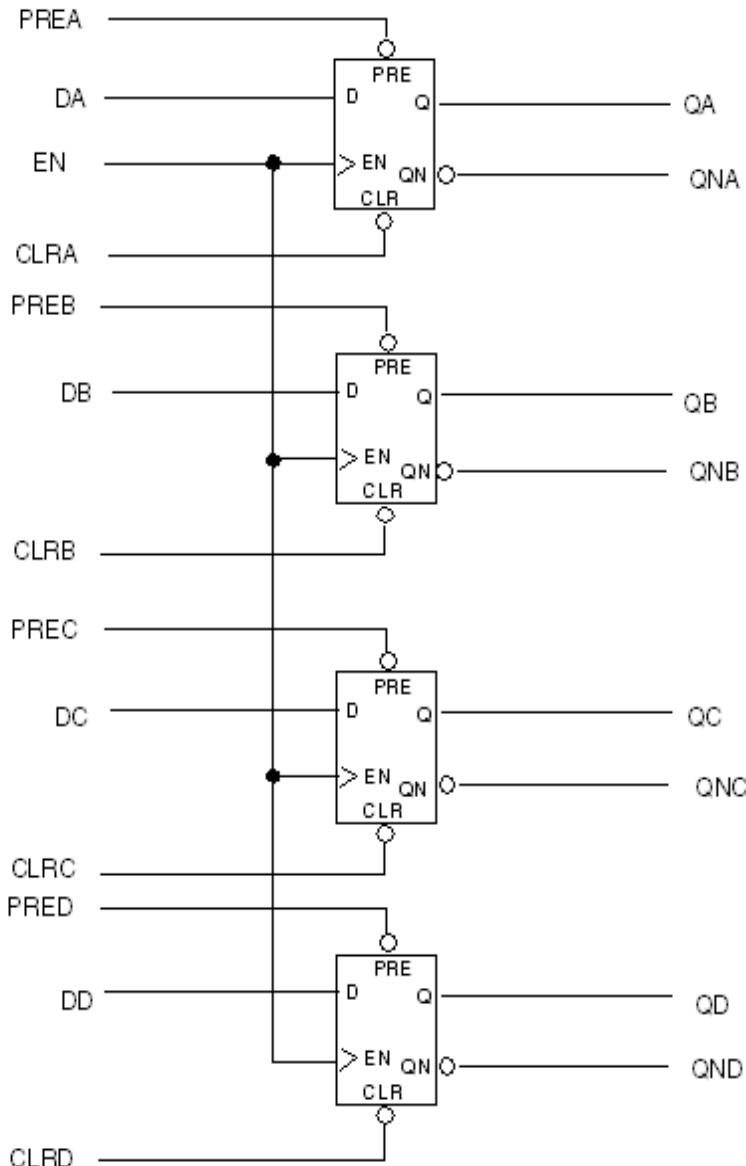
The `bits` value in the `latch_bank` definition is the number of bits in the multibit cell.

Example 33 4-Bit Register Modeled With `latch_bank` Group for Rising-Edge-Triggered D Latches

```
cell (latch4) {
    area: 16;
    pin (G) { /* gate enable signal, active-high */
        direction : input;
        ...
    }
    bundle (D) { /* data input with four member pins */

        members(D1, D2, D3, D4);
        /*must be first bundle attribute*/
        direction : input;
        ...
    }
    bundle (Q) {
        members(Q1, Q2, Q3, Q4);
        direction : output;
        function : "IQ" ;
        ...
    }
    bundle (QN) {
        members (Q1N, Q2N, Q3N, Q4N);
        direction : output;
        function : "IQN";
        ...
    }
    latch_bank(IQ, IQN, 4) {
        enable : "G" ;
        data_in : "D" ;
    }
    ...
}
```

Figure 8 4-Bit Register Containing Enable-High D Latches With Clear



[Example 34](#) is the cell description of the multibit register shown in [Figure 8](#).

Example 34 4-Bit Register Modeled Using Enable-High D Latches With Clear

```

cell (DLT2) {
    /* note: 0 hold time */

    area : 1 ;
    pin (EN) {

```

Chapter 5: Defining Sequential Cells

Describing a Multibit Latch

```
direction : input ;
capacitance : 0 ;
min_pulse_width_low : 3 ;
min_pulse_width_high : 3 ;
}

bundle (D) {
    members(DA, DB, DC, DD);
    direction : input ;
    capacitance : 0 ;
    timing() {
        related_pin      : "EN" ;
        timing_type      : setup_falling ;
    }
    timing() {
        related_pin      : "EN" ;
        timing_type      : hold_falling ;
    }
}

bundle (CLR) {
    members(CLRA, CLRB, CLRC, CLRD);
    direction : input ;
    capacitance : 0 ;
    timing() {
        related_pin      : "EN" ;
        timing_type      : recovery_falling ;
    }
}

bundle (PRE) {
    members(PREA, PREB, PREC, PRED);
    direction : input ;
    capacitance : 0 ;
    timing() {
        related_pin      : "EN" ;
        timing_type      : recovery_falling ;
    }
}

latch_bank(IQ, IQN, 4) {
    data_in : "D" ;
    enable  : "EN" ;
    clear   : "CLR ' " ;
    preset  : "PRE ' " ;
    clear_preset_var1 : H ;
    clear_preset_var2 : H ;
}

bundle (Q) {
    members(QA, QB, QC, QD);
    direction : output ;
    function : "IQ" ;
```

Chapter 5: Defining Sequential Cells
Describing Sequential Cells With the Statetable Format

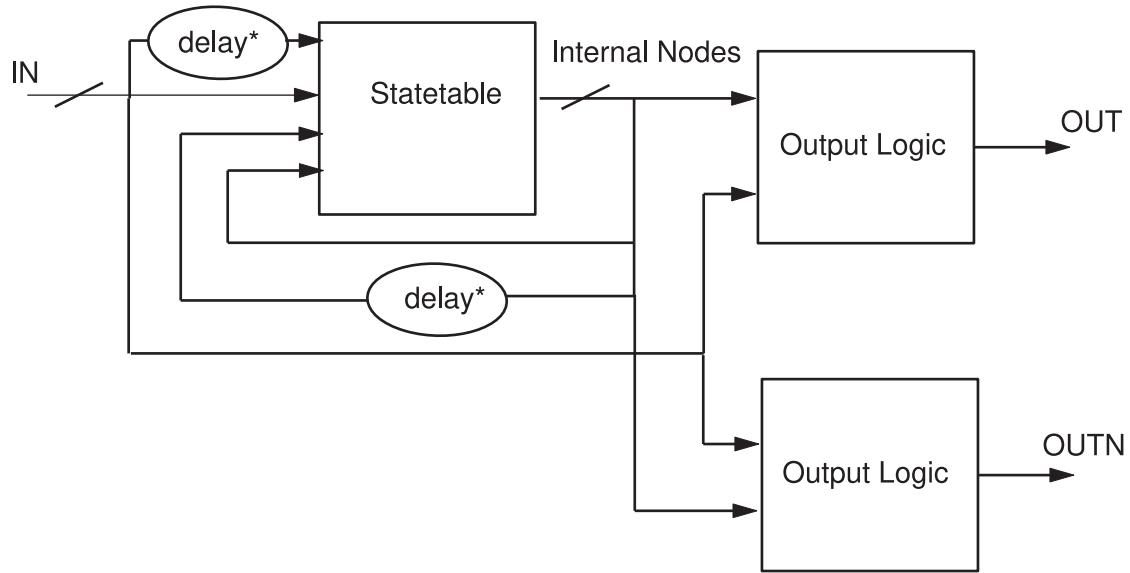
```
timing() {
    related_pin      : "D" ;
}
timing() {
    related_pin      : "EN" ;
    timing_type      : rising_edge ;
}
timing() {
    related_pin      : "CLR" ;
    timing_type      : clear ;
    timing_sense     : positive_unate ;
}
timing() {
    related_pin      : "PRE" ;
    timing_type      : preset ;
    timing_sense     : negative_unate ;
}
}

bundle (QN) {
    members(QNA, QNB, QNC, QND);
    direction : output ;
    function : "IQN" ;
    timing() {
        related_pin      : "D" ;
    }
    timing() {
        related_pin      : "EN" ;
        timing_type      : rising_edge ;
    }
    timing() {
        related_pin      : "CLR" ;
        timing_type      : preset ;
        timing_sense     : negative_unate ;
    }
    timing() {
        related_pin      : "PRE" ;
        timing_type      : clear ;
        timing_sense     : positive_unate ;
    }
}
} /* end of cell DLT2 */
```

Describing Sequential Cells With the Statetable Format

The statetable format provides an intuitive way to describe the function of complex sequential cells. Using this format, the library developer can translate a state table in a databook to a cell description.

Figure 9 Generic Sequential Library Cell Model



OUT and OUTN

Sequential output ports of the sequential cell.

IN

The set of all primary input ports in the sequential cell functionally related to OUT and OUTN.

delay*

A small time delay. An asterisk suffix indicates a time delay.

Statetable

A sequential lookup table. The state table takes a number of inputs and their delayed values and a number of internal nodes and their delayed values to form an index to new internal node values. A sequential library cell can have only one state table.

Internal Nodes

As storage elements, internal nodes store the output values of the state table. There can be any number of internal nodes.

Output Logic

A combinational lookup table. For the sequential cells supported in ff or latch format, there are at most two internal nodes and the output logic must be a buffer, an inverter, a three-state buffer, or a three-state inverter.

To capture the function of complex sequential cells, use the `statetable` group in the `cell` group to define the statetable in [Figure 9](#). The `statetable` group syntax maps to the truth tables in databooks as shown in the example of [Figure 10](#). For table input token values, see [Statetable Group on page 141](#).

Figure 10 Mapping Databook Truth Table to Statetable

Databook	Meaning	Statetable input	Statetable output
f	Fall	F	N/A
nc	No event	N/A	N ←
r	Rise	R	N/A
tg	Toggle	N/A	(1) ←
u	Undefined	N/A	X ←
x	Don't Care	-	X
-	Not used	-	X

Rising edge
(from low to high)

Don't care

Falling edge
(from high to low)

No event from current value

Toggle flag tg (1)

Unknown

To map a databook truth table to a statetable, do the following:

1. When the databook truth table includes the name of an input port, replace that port name with the tokens for low/high (L/H).
2. When the databook truth table includes the name of an output port, use L/H for the current value of the output port and the next value of the output port.
3. When the databook truth table has the toggle flag tg (1), use L/H for the current value of the output port and H/L for the next value of the output port.

In the truth table, an output port preceded with a tilde symbol (~) is inverted.
Sometimes you must map f to ~R and r to ~F.

statetable Group

The `statetable` group contains a table consisting of a single string.

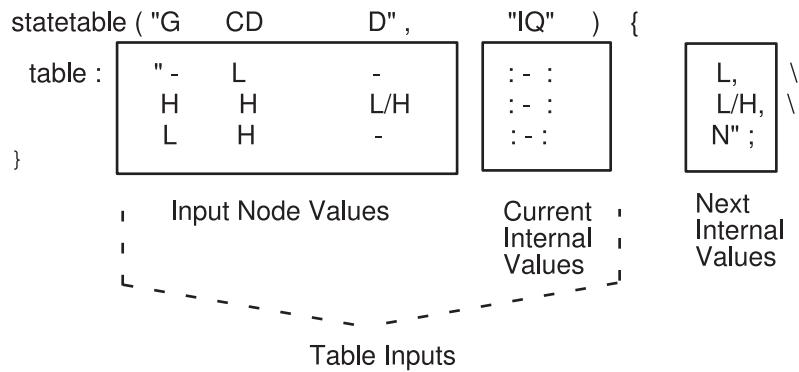
Syntax

```
statetable("input node names", "internal node names")
{
    table : "input node values : current internal values \
              : next internal values,\n
    input node values : current internal values : next internal values";
}
```

You need to follow these conventions when using a `statetable` group:

- Give nodes unique names.
- Separate node names with white space.
- Place each rule consisting of a set of input node values, current internal values, and next internal values on a separate line, followed by a comma and the line continuation character (\). To prevent syntax errors, the line continuation character must be followed immediately by the next line character.
- Separate node values and the colon delimiter (:) with white space.
- Insert comments only where a character space is allowed. For example, you cannot insert a comment within an H/L token or after the line continuation character (\).

Figure 11 *statetable Group Example*



Chapter 5: Defining Sequential Cells
 Describing Sequential Cells With the Statetable Format

Table 13 Legitimate Values for Table Inputs (Input and Current Internal Nodes)

Input node values	Current internal node values	State represented
L	L	Low
H	H	High
-	-	Don't care
L/H	L/H	Expands to both L and H
H/L	H/L	Expands to both H and L
R		Rising edge (from low to high)
F		Falling edge (from high to low)
~R		Not rising edge
~F		Not falling edge

Table 14 Legitimate Values for Next Internal Node

Next internal node values	State represented
L	Low
H	High
-	Output is not specified
L/H	Expands to both L and H
H/L	Expands to both H and L
X	Unknown
N	No event from current value. Hold. Use only when all asynchronous inputs and clocks are inactive

Note:

It is important to use the N token value appropriately. The output is never N when any input (asynchronous or synchronous) is active. The output should be N only when all the inputs are inactive.

Using Shortcuts

To represent a statetable explicitly, list the next internal node one time for every permutation of L and H for the current inputs, previous inputs, and current states.

Example 35 Fully Expanded statetable Group for a Data Latch With Active-Low clear

```
statetable(" G CD D", "IQ") {
    table :" L L L : L : L,\ \
              L L L : H : L,\ \
              L L H : L : L,\ \
              L L H : H : L,\ \
              L H L : L : N,\ \
              L H L : H : N,\ \
              L H H : L : N,\ \
              L H H : H : N,\ \
              H L L : L : L,\ \
              H L L : H : L,\ \
              H L H : L : L,\ \
              H L H : H : L,\ \
              H H L : L : L,\ \
              H H L : H : L,\ \
              H H H : L : H,\ \
              H H H : H : H";
```

You can use the following shortcuts when you represent your table.

don't care symbol (-)

For the input and current internal node, the don't care symbol represents all permutations of L and H. For the next internal node, the don't care symbol means the rule does not define a value for the next internal node.

For example, a master-slave flip-flop can be written as follows:

```
statetable(" D CP CPN", "MQ SQ") {
    table : " H/L R ~F : - - : H/L N,\ \
              - ~R F : H/L - : N H/L,\ \
              H/LR F : L - : H/L L,\ \
              H/LR F : H - : H/L H,\ \
              - ~R ~F : - - : N N";
```

Or it can be written more concisely as follows

```
statetable(" D CP CPN", "MQ SQ") {
    table : " H/L R - : - - : H/L -, \ \
              - ~R - : - - : N -, \ \
              - - F : H/L - : - H/L,\ \
              - - ~F : - - - : - N";
```

Chapter 5: Defining Sequential Cells

Describing Sequential Cells With the Statetable Format

L/H and H/L

Both L/H and H/L represent two individual lines: one with L and the other with H. For example, the following line

H H H/L : - : L/H,

is a concise version of the following lines.

H H H : - : L,
H H L : - : H,

R, ~R, F, and ~F (input edge-sensitive symbols)

The input edge-sensitive symbols represent permutations of L and H for the delayed input and the current input. Every edge-sensitive input, one that has at least one input edge symbol, is expanded into two level-sensitive inputs: the current input value and the delayed input value. For example, the input edge symbol R expands to an L for the delayed input value and to an H for the current input value. In the following statetable of a D flip-flop, clock C can be represented by the input pair C* and C. C* is the delayed input value, and C is the current input value.

```
statetable ( "C      D", "IQ") {  
    table :  "R L/H  : - :  L/H,  \  
             ~R -  : - :  N";
```

Table 15 Internal Representation of the Cell

C*	C	D	IQ
L	H	L/H	L/H
H	H	-	N
H	L	-	N
L	L	-	N

Priority ordering of outputs

The outputs follow a prioritized order. Two rules can cover the same input combinations, but the rule defined first has priority over subsequent rules.

Note:

Use shortcuts with care. When in doubt, be explicit.

Partitioning the Cell Into a Model

You can partition a structural netlist to match the general sequential output model described in [Example 36](#) by performing these tasks:

1. Represent every storage element by an internal node.
2. Separate the output logic from the internal node. The internal node must be a function of only the sequential cell data input when the sequential cell is triggered.

Note:

There are two ways to specify that an output does not change logic values. One way is to use the inactive N value as the next internal value, and the other way is to have the next internal value remain the same as the current internal value (see the italic lines in [Example 36](#)).

Example 36 JK Flip-Flop With Active-Low, Direct-Clear, and Negative-Edge Clock

```
statetable ("J K CN CD" , "IQ") {  
    table : " - - - L : - : L,\\"  
             - - ~F H : - : N,\\"  
             L L F H : L/H : L/H,\\"  
             H L F H : - : H,\\"  
             L H F H : - : L,\\"  
             H H F H : L/H : H/L" ;  
}
```

In [Example 36](#), the value of the next internal node of the second rule is N, because both the clear CD and clock CN are inactive. The value of the next internal node of the third rule is L/H, because the clock is active.

Defining an Output pin Group

Every output pin in the cell has either an `internal_node` attribute, which is the name of an internal node, or a `state_function` attribute, which is a Boolean expression of ports and internal pins.

Every output pin can also have an optional `three_state` expression.

An output pin can have one of the following combinations:

- A `state_function` attribute and an optional `three_state` attribute
- An `internal_node` attribute, an optional `input_map` attribute, and an optional `three_state` attribute

state_function Attribute

The `state_function` attribute defines output logic. Ports in the `state_function` Boolean expression must be either input, inout that can be made three-state, or ports with an `internal_node` attribute.

The `state_function` attribute specifies the purely combinational function of input and internal pins. A port in the `state_function` expression refers only to the non-three-state functional behavior of that port. For example, if E is a port of the cell that enables the three-state, the `state_function` attribute cannot have a Boolean expression that uses pin E.

An inout port in the `state_function` expression is treated only as an input port.

Note:

Use the `state_function` attribute to define a cell with a state table. Use this attribute when the functional expression does not reference any state table signals, and is a function of only the input pins.

Example

```
pin(Q)
  direction : output;
  state_function : "A"; /*combinational feedthrough*/
```

internal_node Attribute

The `internal_node` attribute is used to resolve node names to real port names.

The statetable is in an independent, isolated name space. The term *node* refers to the identifiers. Input node and internal node names can contain any characters except white space and comments. These node names are resolved to the real port names by each port with an `internal_node` attribute.

Each output defined by the `internal_node` attribute might have an optional `input_map` attribute.

Example

```
internal_node :
  "IQ";
```

input_map Attribute

The `input_map` attribute maps real port and internal pin names to each table input and internal node specified in the state table. An input map is a listing of port names, separated by white space, that correspond to internal nodes.

Example

```
input_map : "Gx1 CDx1 Dx1 QN"; /*QN is internal node*/
```

Mapping port and internal pin names to table input and internal nodes occurs by using a combination of the `input_map` attribute and the `internal_node` attribute. If a node name is not mapped explicitly to a real port name in the input map, it automatically inherits the node name as the real port name. The internal node name specified by the `internal_node` attribute maps implicitly to the output being defined. For example, to map two input nodes, A and B, to real ports D and CP, and to map one internal node, Z, to port Q, set the `input_map` attribute as follows:

```
input_map : "D  CP  Q"
```

You can use a *don't care* symbol in place of a name to signify that the input is not used for this output. Comments are allowed in the `input_map` attribute.

The delayed nature of outputs specified with the `input_map` attribute is implicit:

Internal node

When an output port is specified for this type of node, the internal node is forced to map to the delayed value of the output port.

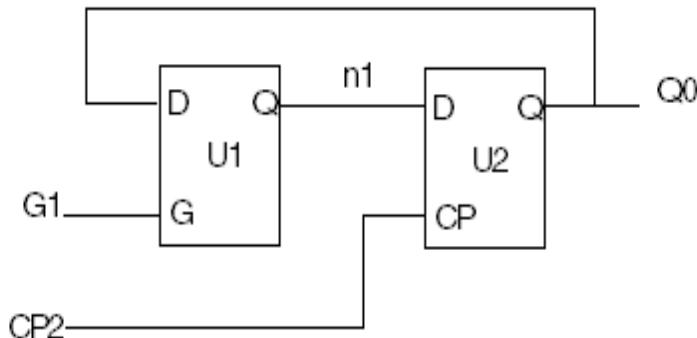
Synchronous data input node

When an output port is specified for this type of node, the input is forced to map to the delayed value of the output port.

Asynchronous or clock input node

When an output port is specified for this type of node, the input is forced to map to the undelayed value of the output port.

Figure 12 Circuit With Latch U1 and Flip-Flop U2



For example, in Figure 12, internal pin n1 should map to ports "Q0 G1 n1*" and output Q0 should map to "n1* CP2 Q0*". The subtle significance is relevant during simultaneous input transitions.

With this `input_map` attribute, the functional syntax for ganged cells is identical to that of nonganged cells. You can use the input map to represent the interconnection of any network of sequential cells (for example, shift registers and counters).

The `input_map` attribute can be completely unspecified, completely specified, or can specify only the beginning ports. The default for an incompletely defined input map is the assumption that missing trailing primary input nodes and internal nodes are equal to the node names specified in the statetable. The current state of the internal node should be equal to the output port name.

inverted_output Attribute

You can define output as inverted or noninverted by using an `inverted_output` attribute on the pin. The `inverted_output` attribute is a required Boolean attribute for the statetable format that you can set for any output port. Set this attribute to false for noninverting output. This is variable1 or IQ for flip-flop or latch groups. Set this attribute to true for inverting output. This is variable2 or IQN for flip-flop or latch groups.

Example

```
inverted_output : true;
```

In the statetable format, an output is considered inverted if it has any data input with a negative sense. This algorithm might be inconsistent with a user's intentions. For example, whether a toggle output is considered inverted or noninverted is arbitrary.

Internal Pin Type

In the flip-flop or latch format, the `pin`, `bus`, or `bundle` groups can have a `direction` attribute with input, output, or inout values. In the statetable format, the `direction` attribute of a pin of a library cell (combinational or sequential) can have the internal value. A pin with the internal value to the `direction` attribute is treated like an output port, except that it is hidden within the library cell. It can take any of the attributes of an output pin, but it cannot have the `three_state` attribute.

An internal pin can have timing arcs and can have timing arcs related to it, allowing a distributed delay model for multistate sequential cells.

Because a cell with a statetable model is considered a black box for synthesis, no checks are performed to ensure consistency between timing and function.

Example

```
pin (n1) {
    direction : internal;
    internal_node :"IQ"; /* use default input_map */
    ...
}
pin (QN) {
```

```
direction : output;
internal_node :"IQN";
input_map : "Gx1 CDx1 Dx1 QN"; /* QN is internal node */

three_state : "EN2";
...
}
pin (QNZ) {
    direction : output;
    state_function :"QN"; /* ignores QN's three state */

    three_state : "EN";
    ...
}
pin (Y) {
    direction : output;
    state_function : "A"; /* combinational feedthrough */

    ...
}
```

Determining a Complex Sequential Cell's Internal State

You can use the `power_down_function` string attribute to specify the Boolean condition under which the cell's output pin is switched off by the state of the power and ground pins (when the cell is in off mode due to the external power pin states). The attribute should be set under the `statetable` group. The `power_down_function` attribute also determines the condition of the cell's internal state.

For more information about the `power_down_function` attribute, see [power_down_function Attribute on page 121](#).

power_down_function Syntax for State Tables

The `power_down_function` attribute is specified after `table`, in the `statetable` group, as shown in the following syntax:

```
statetable( "input node names", "internal node names" ){
    table : " input node values : current internal values :\n        next internal values,\n        input node values : current internal values:\n        next internal values" ;
    power_down_function : "Boolean expression" ;
}
```

Flip-Flop and Latch Examples

Example 37 D Flip-Flop

```
/* ff/latch format */

ff (IQ,IQN) {
    next_state : "D" ;
    clocked_on : "CP" ;
}
/* statetable format */

statetable("      D CP", "IQ") {
    table : "      H/L   R : - : H/L,\n"
             "- ~R : - : N";
}
```

Example 38 D Flip-Flops With Master-Slave Clock Input Pins

```
/* ff/latch format */

ff (IQ,IQN) {
    next_state : "D" ;
    clocked_on : "CK" ;
    clocked_on_also : "CKN'" ;
}

/* statetable format */
statetable(" D   CK      CKN", "MQSQ") {
    table : " H/L   R   ~F : - - : H/L   N,\n"
             "- ~R   F : H/L - : N   H/L,\n"
             " H/L   R   F : L   - : H/L   L,\n"
             " H/L   R   F : H   - : H/L   H,\n"
             "-   ~R   ~F : - - : N   N";
}
```

Example 39 D Flip-Flop With Gated Clock

```
/* ff/latch format */

ff (IQ,IQN) {
    next_state : "D" ;
    clocked_on : "C1 * C2" ;
}

/* statetable format */

statetable("      D  C1 C2", "IQ") {
    table : "      H/L   H   R : - : H/L,\n"
             " H/L   R   H : - : H/L,\n"
             " H/L   R   R : - : H/L,\n"
             "-   - - : - : N";
}
```

Chapter 5: Defining Sequential Cells
 Flip-Flop and Latch Examples

}

Example 40 D Flip-Flop With Active-Low Direct-Clear and Active-Low Direct-Set

```
/* ff/latch format */

ff (IQ,IQN) {
    next_state : "D" ;
    clocked_on : "CP" ;
    clear : " CD' " ;
    preset : " SD' " ;
    clear_preset_var1 : L ;
    clear_preset_var2 : L ;
}

/* statetable format */

statetable("D      CP      CD      SD", "IQ IQN") {
    table : "H/L  R      H      H : -      - : H/L      L/H, \
              - ~R      H      H : -      - : N          N, \
              -      L      H : -      - : L          H, \
              -      -      H      L : -      - : H          L, \
              -      -      L      L : -      - : L          L";
}
```

Example 41 D Flip-Flop With Active-Low Direct-Clear, Active-Low Direct-Set, and One Output

```
/* ff/latch format */

ff (IQ,IQN) {
    next_state : "D" ;
    clocked_on : "CP" ;
    clear : " CD' " ;
    preset : " SD' " ;
    clear_preset_var1 : L ;
    clear_preset_var2 : L ;
}

/* statetable format */

statetable(" D      CP      CD      SD", "IQ") {
    table : " H/L  R      H      H : -      - : H/L, \
              - ~R      H      H : -      - : N, \
              -      L      H/L   : -      - : L, \
              -      -      H      L : -      - : H";
}
```

Example 42 JK Flip-Flop With Active-Low Direct-Clear and Negative-Edge Clock

```
/* ff/latch format */

ff (IQ, IQN) {
    next_state : " (J' K' IQ) + (J K') + (J K IQ' )"
```

Chapter 5: Defining Sequential Cells
 Flip-Flop and Latch Examples

```
;
    clocked_on : " CN'" ;
    clear : " CD'" ;
}

/* statetable format */

statetable(" J   K   CD   CD", "IQ") {
    table : " -   -   -   L   : -   : L,\n
              -   -   ~F  H   : -   : N,\n
              L   L   F   H   : L/H : L/H \
              H   L   F   H   : -   : H,\n
              L   H   F   H   : -   : L,\n
              H   H   F   H   : L/H : H/L";
```

Example 43 D Flip-Flop With Scan Input Pins

```
/* ff/latch format */

ff (IQ, IQN) {
    next_state : " (D TE') + (TI TE)" ;
    clocked_on : "CP" ;
}

/* statetable format */

statetable(" D   TE   TI   CP", "IQ") {
    table : " H/L  L   -   R   : -   : H/L,\n
              -     H   H/L  R   : -   : H/L,\n
              -     -   -   ~R  : -   : N";
}
```

Example 44 D Flip-Flop With Synchronous Clear

```
/* ff/latch format */

ff (IQ, IQN) {
    next_state : "D CR" ;
    clocked_on : "CP" ;
}

/* statetable format */

statetable(" D   CR   CP", "IQ") {
    table : " H/L  H   R   : -   : H/L,\n
              -     L   R   : -   : L,\n
              -     -   ~R  : -   : N";
}
```

Example 45 D Latch

```
/* ff/latch format */

latch (IQ, IQN) {
```

Chapter 5: Defining Sequential Cells Flip-Flop and Latch Examples

```
    data_in : "D" ;
    enable : "G" ;
}

/* statetable format */

statetable(" D G", "IQ") {
    table : " H/L H : - : H/L,\n            - L : - : N";
}
```

Example 46 SR Latch

```
/* ff/latch format */

latch (IQ,IQN) {
    clear : "R" ;
    preset : "S" ;
    clear_preset_var1 : L ;
    clear_preset_var2 : L ;
}

/* statetable format */

statetable(" R      S",      "IQ IQN") {
    table : " H   L   : - - : L   H,\n              L   H   : - - : H   L,\n              H   H   : - - : L   L,\n              L   L   : - - : N   N";
}
```

Example 47 D Latch With Active-Low Direct-Clear

```
/* ff/latch format */

latch (IQ,IQN) {
    data_in : "D" ;
    enable : "G" ;
    clear : " CD' " ;
}

/* statetable format */

statetable(" D   G   CD",      "IQ") {
    table : " H/L H   H   : - : H/L,\n              -   L   H   : - : N,\n              -   -   L   : - : L";
}
```

Example 48 Multibit D Latch With Active-Low Direct-Clear

```
/* ff/latch format */
```

Chapter 5: Defining Sequential Cells

Cell Description Examples

```

latch_bank(IQ, IQN, 4) {
    data_in : "D" ;
    enable : "EN" ;
    clear : "CLR'" ;
    preset : "PRE'" ;
    clear_preset_var1 : H ;
    clear_preset_var2 : H ;
}

/* statetable format */

statetable(" D   EN   CL   PRE", "IQ  IQN") {
    table : "H/L  H   H   H : -   - : H/L  L/H,\n
              -   L   H   H : -   - : N   N,\n
              -   -   L   H : -   - : L   H,\n
              -   -   H   L : -   - : H   L,\n
              -   -   L   L : -   - : H   H";
}

```

Example 49 D Flip-Flop With Scan Clock

```

statetable(" D   S   CD   SC   CP", "IQ") {
    table : " H/L  -   ~R   R   : - : H/L,\n
              -   H/L  R   ~R   : - : H/L,\n
              -   -   ~R   ~R   : - : N,\n
              -   -   R   R   : - : X";
}

```

Cell Description Examples

Example 50 D Latches With Master-Slave Enable Input Pins

```

cell(ms_latch) {
    area : 16;
    pin(D) {
        direction : input;
        capacitance : 1;
    }
    pin(G1) {
        direction : input;
        capacitance : 2;
    }
    pin(G2) {
        direction : input;
        capacitance : 2;
    }
    pin(mq) {
        internal_node : "Q";
        direction : internal;
        input_map : "D G1";
        timing() {

```

Chapter 5: Defining Sequential Cells Cell Description Examples

```
        related_pin : "G1";
    }
}
pin(Q) {
    direction : output;
    function : "IQ";
    internal_node : "Q";
    input_map : "mq G2";
    timing() {
        related_pin : "G2";
    }
}
pin(QN) {
    direction : output;
    function : "IQN";
    internal_node : "QN";
    input_map : "mq G2";
    timing() {
        related_pin : "G2";
    }
}
ff(IQ, ION) {
    clocked_on : "G1";
    clocked_on_also : "G2";
    next_state : "D";
}
statetable ( "D G", "Q QN" ) {
    table : "L/H H : - - : L/H H/L,\n          - L : - - : N N";
}
}
```

Example 51 FF Shift Register With Timing Removed

```
cell(shift_reg_ff) {
    area : 16;
    pin(D) {
        direction : input;
        capacitance : 1;
    }
    pin(CP) {
        direction : input;
        capacitance : 2;
    }
    pin (Q0) {
        direction : output;
        internal_node : "Q";
        input_map : "D CP";
    }
    pin (Q1) {
        direction : output;
        internal_node : "Q";
        input_map : "Q0 CP";
    }
}
```

Chapter 5: Defining Sequential Cells

Cell Description Examples

```
        }
    pin (Q2) {
        direction : output;
        internal_node : "Q";
        input_map : "Q1 CP";
    }
    pin (Q3) {
        direction : output;
        internal_node : "Q";
        input_map : "Q2 CP";
    }
    statetable( "D CP", "Q QN" ) {
        table : "- ~R : - - : N   N,\n          H/L R : - - : H/L L/H";
    }
}
```

Example 52 FF Counter With Timing Removed

```
cell(counter_ff) {
    area : 16;
    pin(reset) {
        direction : input;
        capacitance : 1;
    }
    pin(CP) {
        direction : input;
        capacitance : 2;
    }
    pin (Q0) {
        direction : output;
        internal_node : "Q0";
        input_map : "CP reset Q0 Q1";
    }
    pin (Q1) {
        direction : output;
        internal_node : "Q1";
        input_map : "CP reset Q0 Q1";
    }
    statetable( "CP reset", "Q0 Q1" ) {
        table : "- L : - - : L H,\n          ~R H : - - : N N,\n          R H : L L : H L,\n          R H : H L : L H,\n          R H : L H : H H,\n          R H : H H : L L";
    }
}
```

6

Defining Test Cells

A test cell contains information that supports a full-scan or a partial-scan methodology.

You must add test-specific details of scannable cells to your logic libraries. For example, you must identify scannable flip-flops and latches and select the types of unscannable cells they replace for a given scan methodology.

To do this, you must understand the following concepts described in this chapter:

- [Describing a Scan Cell](#)
- [Describing a Multibit Scan Cell](#)
- [Scan Cell Modeling Examples](#)

Describing a Scan Cell

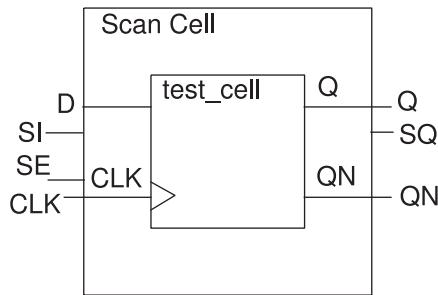
To specify a cell as a scan cell, add the `test_cell` group to the cell description.

Only the nontest mode function of a scan cell is modeled in the `test_cell` group. The nontest operation is described by its `ff`, `ff_bank`, `latch`, or `latch_bank` declaration and `pin` function attributes.

test_cell Group

The `test_cell` group defines only the nontest mode function of a scan cell.

Figure 13 test_cell Group Defined in a Scan Cell



Ensure the following when you define a `test_cell` group in a scan cell:

- Pin names in the scan cell and the test cell must match.
- The scan cell and the test cell must contain the same functional outputs.

Syntax

```
library (lib_name) { cell (cell_name) {
    test_cell () {
        ... test cell
    description ...
        pin ( name ) {
            ... pin description ...
        }
        pin ( name ) {
            ...
        pin description ...
        }
    }
}
```

You do not need to give the `test_cell` group a name, because the test cell takes the cell name of the cell being defined. The `test_cell` group can contain `ff`, `ff_bank`, `latch`, or `latch_bank` group statements and `pin` groups.

Pins in the `test_cell` Group

Both test pins and nontest pins can appear in `pin` groups within a `test_cell` group. These groups are like `pin` groups in a `cell` group but must adhere to the following rules:

- Each pin defined in the `cell` group must have a corresponding pin defined at the `test_cell` group level with the same name.

- The pin group can contain only direction, function, test_output_only, and signal_type attribute statements. The pin group cannot contain timing, capacitance, fanout, or load information.
- The function attributes can reflect only the nontest behavior of the cell.
- Input pins must be referenced in an ff, ff_bank, latch, or latch_bank statement or have a signal_type attribute assigned to them.
- An output pin must have either a function attribute, a signal_type attribute, or both.

Example 53 Multiplexed D Flip-Flop Scan Cell Modeled With Multiple test_cell Groups

```
cell(SDFF1) {
    area : 9;
    pin(D) {
        direction : input;
        capacitance : 1;
        timing() {
            timing_type : setup_rising;
            related_pin : "CLK";
        }
        timing() {
            timing_type : hold_rising;
            related_pin : "CLK";
        }
    }
    pin(CLK) {
        direction : input;
        capacitance : 1;
    }
    pin(SI) {
        direction : input;
        capacitance : 1;
        prefer_tied : "0";
        timing() {
            timing_type : setup_rising;
            related_pin : "CLK";
        }
        timing() {
            timing_type : hold_rising;
            related_pin : "CLK";
        }
    }
    pin(SE) {
        direction : input;
        capacitance : 1;
        timing() {
            timing_type : setup_rising;
            related_pin : "CLK";
        }
        timing() {
```

Chapter 6: Defining Test Cells

Describing a Scan Cell

```
        timing_type : hold_rising;
        related_pin : "CLK";
    }
}
ff("IQ","IQN") {
    next_state : "(D & !SE) | (SI & SE)";
    clocked_on : "CLK";
}
pin(Q) {
    direction : output;
    function : "IQ"
    timing() {
        timing_type : rising_edge;
        related_pin : "CLK";
    }
}
pin(QN) {
    direction : output;
    function : "IQN"
    timing() {
        timing_type : rising_edge;
        related_pin : "CLK";
    }
}
/* first test_cell group defines nontest
   behavior with both Q and QN */
test_cell() {
    pin(D,CLK) {
        direction : input;
    }
    pin(SI) {
        direction : input;
        signal_type : "test_scan_in";
    }
    pin(SE) {
        direction : input;
        signal_type : "test_scan_enable";
    }
    ff ("IQ","IQN") {
        next_state : "D";
        clocked_on : "CLK";
    }
    pin(Q) {
        direction : output;
        function : "IQ";
        signal_type : "test_scan_out";
    }
    pin(QN) {
        direction : output;
        function : "IQN";
        signal_type : "test_scan_out_inverted";
    }
}
```

```
/* second test_cell group defines nontest
   behavior with only Q */
test_cell() {
    pin(D,CLK) {
        direction : input;
    }
    pin(SI) {
        direction : input;
        signal_type : "test_scan_in";
    }
    pin(SE) {
        direction : input;
        signal_type : "test_scan_enable";
    }
    ff ("IQ","IQN") {
        next_state : "D";
        clocked_on : "CLK";
    }
    pin(Q) {
        direction : output;
        function : "IQ";
        signal_type : "test_scan_out";
    }
    pin(QN) {
        direction : output;
        /* notice no function attribute for QN pin */
        signal_type : "test_scan_out_inverted";
    }
}
```

test_output_only Attribute

This attribute indicates that an output port is set for test only (as opposed to function only, or function and test).

Syntax

```
test_output_only : true | false ;
```

When you use statetable format to describe the functionality of a scan cell, you must declare the output pin as set for test only by setting the `test_output_only` attribute to `true`.

Example

```
test_output_only : true ;
```

Describing a Multibit Scan Cell

Multibit scan cells can have parallel or serial scan chains. The scan output of a multibit scan cell can reuse the data output pins or have a dedicated scan output (SO) pin in addition to the bus, or the bundle output pins.

Multibit scan cells with the following structures are supported:

- Parallel scan chain without dedicated SO bus
 - Parallel scan chain with dedicated SO bus
 - Serial scan chain without a dedicated SO pin
 - Serial scan chain with a dedicated SO pin
-

Multibit Scan Cell With Parallel Scan Chain

A multibit scan cell with parallel scan chain has parallel data and scan inputs, and parallel functional and scan outputs.

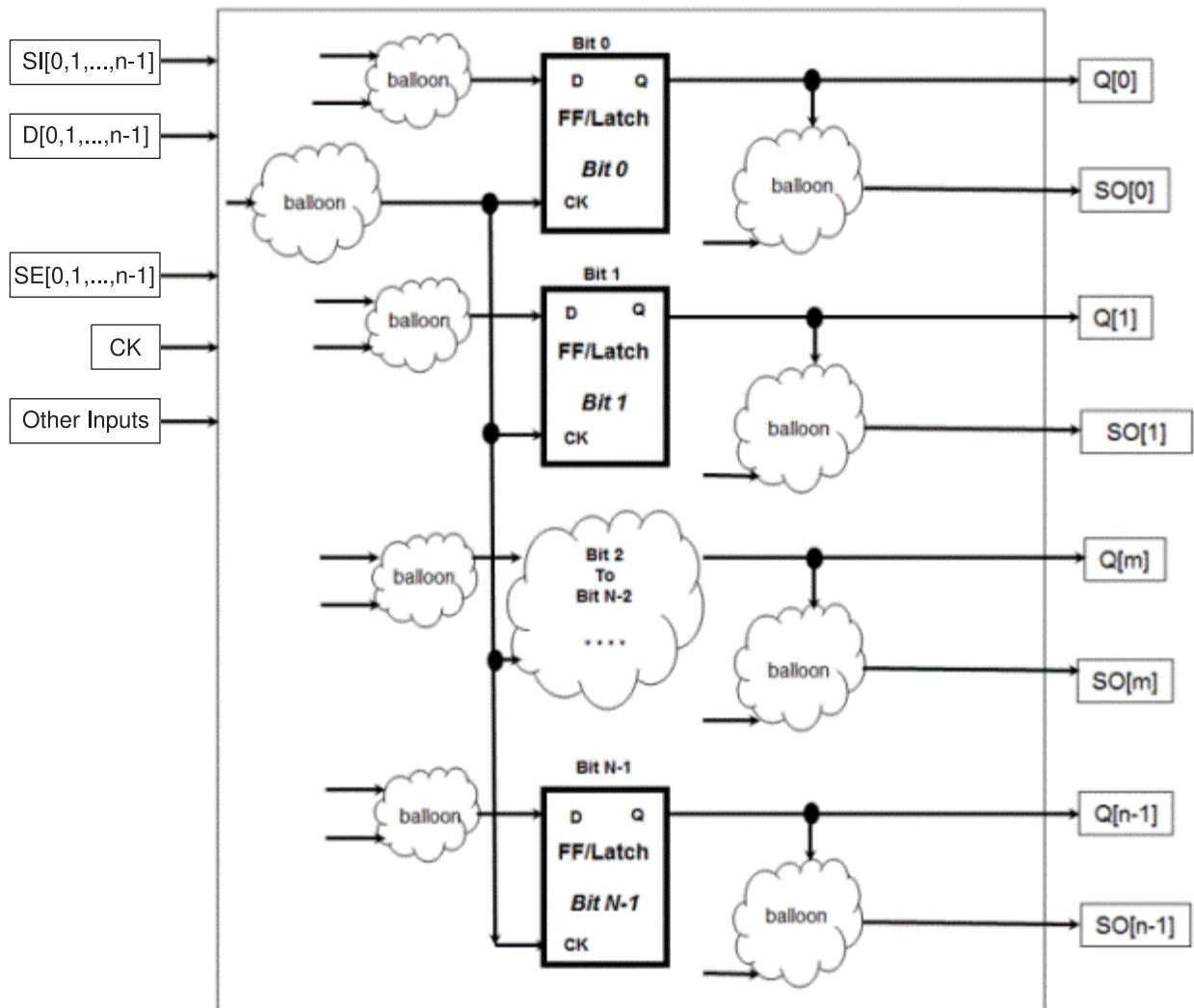
[Figure 14](#) shows the schematic diagram of a generic multibit scan cell with parallel input and output buses for the normal and scan modes.

In the normal mode, the cell uses the input and output buses, D[0:n-1] and Q[0:n-1], respectively.

The balloons represent combinational logic. The combinational logic at the input of each sequential element of the cell is a multiplexer with data and scan inputs. At the input of clock, CK, to each sequential element, the combinational logic is a clock pin or a clock-gating logic.

In the scan mode, the cell functions as a parallel-in parallel-out shift register with the scan input bus, SI[0:n-1], and either reuses the bus, Q[0:n-1], or uses a dedicated bus, SO[0:n-1], to output the scan data. The scan enable signal can be a bus, SE[0:n-1], where each bit of the bus enables a corresponding sequential element of the cell. The scan enable signal can also be a single-bit pin to enable each sequential element of the cell.

Figure 14 Generic Schematic Diagram of a Multibit Scan Cell With Parallel Scan Chain



Use the following syntax to model the multibit scan cell shown in Figure 14:

```
library(library_name) {
...
cell(cell_name) {
...
bus(scan_in_pin_name) {
    /* cell scan in with signal_type "test_scan_in" from test_cell */
...
}
bus(scan_out_pin_name) {
    /* cell scan out with signal_type "test_scan_out" from test_cell */
...
}
```

```
        }
    bus | bundle (bus_bundle_name) {
        direction : input | output;
    }
    test_cell() {
        pin(scan_in_pin_name) {
            signal_type : test_scan_in;
            ...
        }
        pin(scan_out_pin_name) {
            signal_type : test_scan_out |
                test_scan_out_inverted;
            ...
        }
        ...
    }
}
```

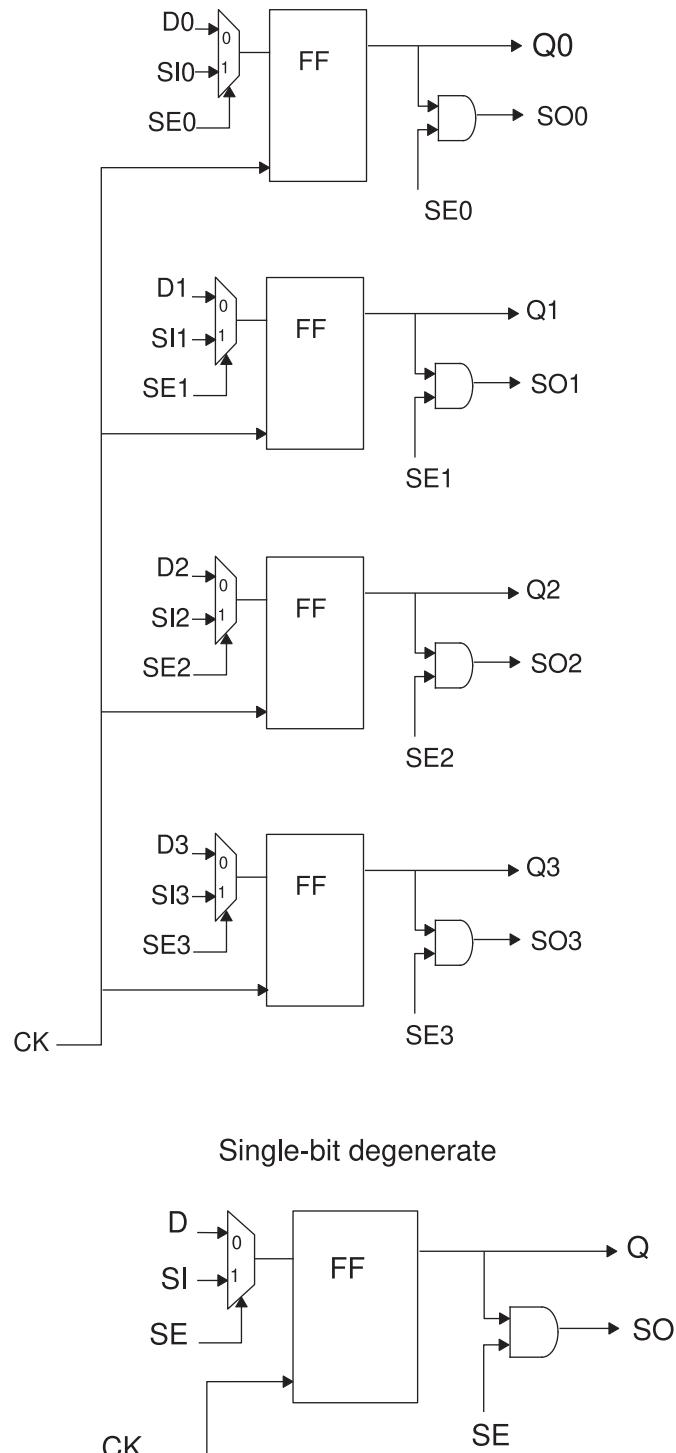
Example

4-bit Scan Cell With Parallel Scan Chains and Dedicated Scan Out Bus

[Figure 15](#) shows the schematic of a 4-bit scan cell with parallel scan chains and a dedicated scan output bus, SO[0:3]. The figure also shows an equivalent single-bit cell.

Chapter 6: Defining Test Cells
Describing a Multibit Scan Cell

Figure 15 4-Bit Scan Cell With Parallel Scan Chain and Single-Bit Equivalent Cell



Each pin of the scan output bus, SO[0:3], has AND logic. The cell is defined using:

- The `statetable` group
- The `state_function` attribute on the bus, SO

In the normal mode, the cell is a parallel shift register that uses the data input bus, D[0:3], and the data output bus, Q[0:3].

In the scan mode, the cell functions as a shift register with parallel scan input, scan enable, and scan output buses, SI[0:3], SE[0:3], and SO[0:3]. To use the cell in the scan mode, set the `signal_type` attribute as `test_scan_out` on the bus, SO, in the `test_cell` group. Do not define the `state_function` attribute for this bus in the `test_cell` group.

[Example 54](#) describes a model of the multibit scan cell shown in [Figure 15](#). The example specifically models the multibit scan cell, and not the single-bit degenerate cell.

Example 54 Model of 4-Bit Scan Cell With Parallel Scan Chain and Dedicated Scan Output

```
cell (4-bit_Parallel_Scan_Cell) {
    ...
    statetable ("D      CK     SI     SE", "IQ, IQN") {
        table :   "-      ~R     -      -      :-      -: N      N,      \
                   H/L    R      L      -      :-      -: H/L   H/L, \
                   -      R      H      H/L    :-      -: H/L   L/H";
    }
    /* functional output bus */
    bus(Q) {
        bus_type : bus4;
        direction : output;
        function : IQ;
        ...
    }
    /* dedicated scan output bus */
    bus(SO) {
        bus_type : bus4;
        direction : output;
        state_function : "Q * SE" ;
        ...
    }
    pin(CK) {
        direction : input;
        ...
    }
    /* scan enable bus */
    bus(SE) {
        bus_type : bus4;
        direction : input;
        ...
    }
    /* scan input bus */
```

Chapter 6: Defining Test Cells

Describing a Multibit Scan Cell

```
bus(SI) {
    bus_type : bus4;
    direction : input;
    ...
}
/* data input bus pins */
bus(D) {
    bus_type : bus4;
    direction : input;
    ...
}
...
test_cell () {
    pin(CK) {
        direction : input;
    }
    bus(D) {
        bus_type : bus4;
        direction : input;
    }
    bus(SI) {
        bus_type : bus4;
        direction : input;
        signal_type : "test_scan_in";
    }
    bus(SE) {
        bus_type : bus4;
        direction : input;
        signal_type : "test_scan_enable";
    }
    ff_bank (IQ,IQN,4) {
        next_state : "D";
        clocked_on : "CK";
    }
    bus(Q) {
        bus_type : bus4 ;
        direction : output;
        function : "IQ";
    }
    bus(SO) {
        bus_type : bus4;
        direction : output;
        signal_type : "test_scan_out";
    }
} /* end test_cell group */
} /* end cell group */
```

Multibit Scan Cell With Internal Scan Chain

A multibit scan cell with an internal scan chain has a serial scan chain and a dedicated pin to output the scan signal.

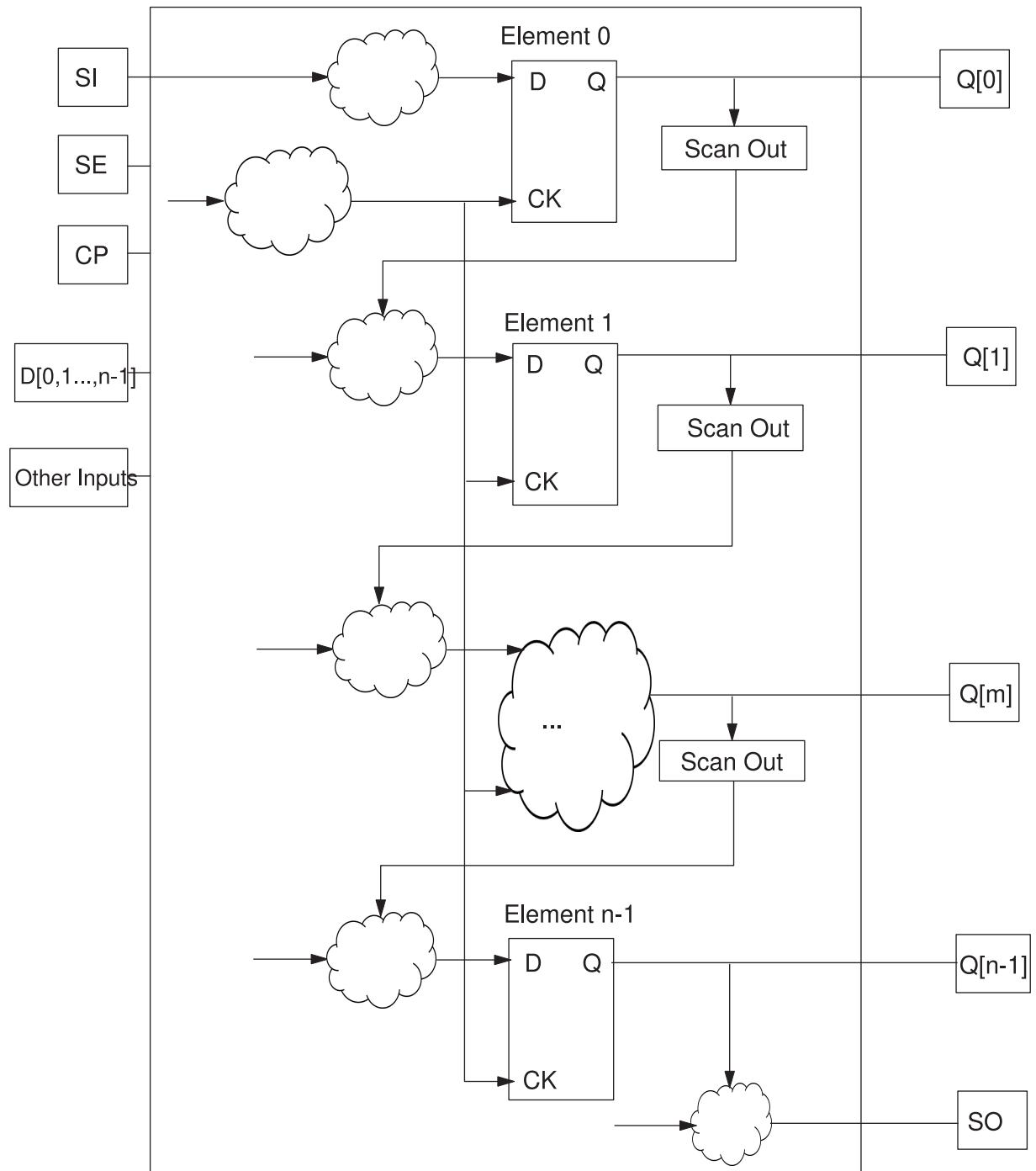
[Figure 16](#) shows the schematic diagram of a generic multibit scan cell with an internal or serial scan chain and the pin, SO, for the scan chain output.

In the normal mode, the cell is a shift register that uses the parallel input and output buses, D[0:n-1] and Q[0:n-1], respectively.

The balloons represent combinational logic. The combinational logic at the input of each sequential element of the cell is a multiplexer with data and scan inputs. At the input of clock, CK, to each sequential element, the combinational logic is clock-gating logic.

In the scan mode, the cell functions as a shift register with the single-bit output pin, SO. The serial scan chain is from the scan input (SI) pin to the scan output (SO) pin. The scan chain is stitched using the data output, Q, of each sequential element of the cell.

Figure 16 Generic Schematic Diagram of a Multibit Scan Cell With Serial Scan Chain



Use the following syntax to model the multibit scan cell shown in [Figure 16](#):

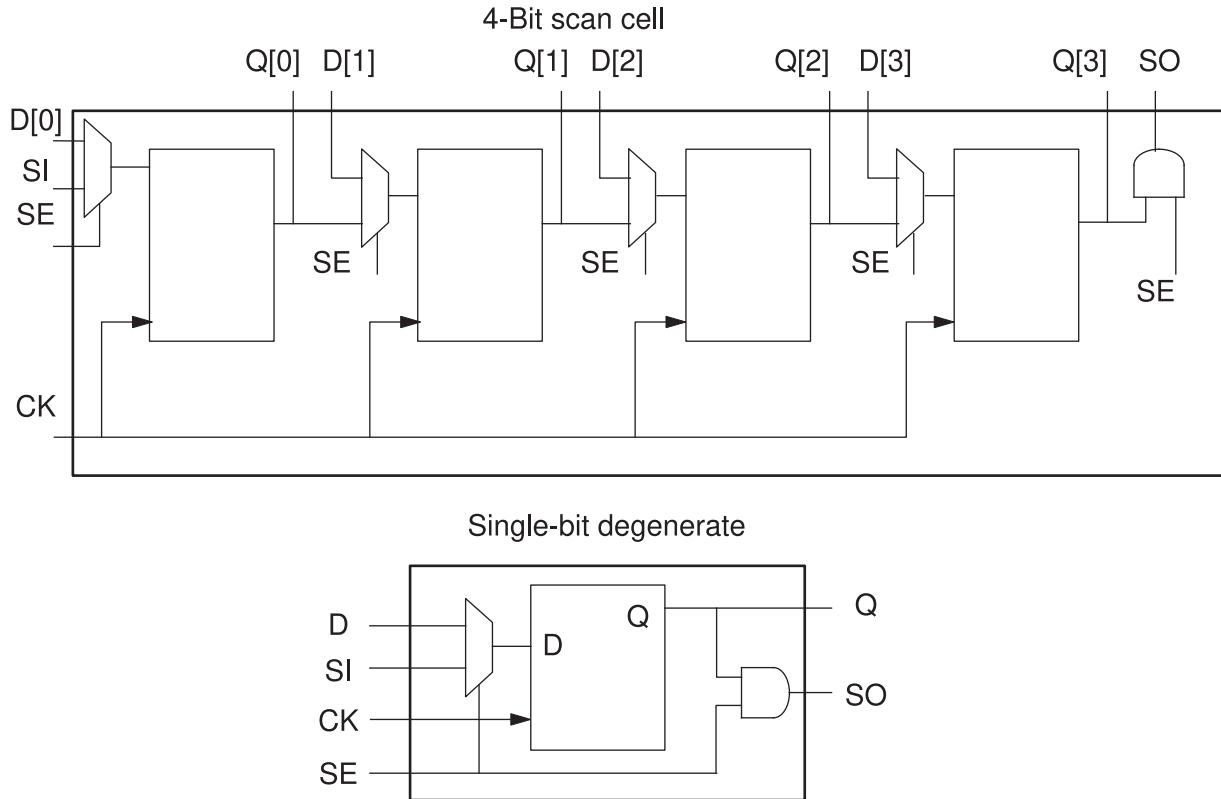
```
library(library_name) {
    ...
    cell(cell_name) {
        pin(scan_in_pin_name) {
            /* cell scan in with signal_type "test_scan_in" from test_cell */
            ...
        }
        pin(scan_out_pin_name) {
            /* cell scan out with signal_type "test_scan_out" from test_cell */
            ...
        }
        bus | bundle (bus_bundle_name) {
            direction : inout | output;
        }
        test_cell() {
            pin(scan_in_pin_name) {
                signal_type : test_scan_in;
                ...
            }
            pin(scan_out_pin_name) {
                signal_type : test_scan_out | test_scan_out_inverted;
                ...
            }
            ...
        }
    }
}
```

Examples

The following sections show Liberty examples of multibit scan cells with internal scan chains.

4-Bit Scan Cell With Internal Scan Chain and Dedicated Scan Out Pin

Figure 17 4-Bit Scan Cell With Internal Scan Chain and Dedicated Scan Out Pin, SO



The 4-bit cell has the output bus Q[0:3], and a single-bit output pin (SO) with combinational logic. The cell is defined using:

- The `statetable` group
- The `state_function` attribute on the pin, SO

In the normal mode, the cell is a shift register that uses the output bus Q[0:3].

In scan mode, the cell functions as a shift register with the single-bit output pin, SO. To use the cell in scan mode, set the `signal_type` attribute as `test_scan_out` on the pin, SO, in the `test_cell` group. Do not define the `function` attribute of this pin.

[Example 55](#) describes a model of the multibit scan cell shown in Figure 17. The example specifically models the multibit scan cell, and not the single-bit degenerate cell, SB.

Example 55 Model of 4-Bit Multibit Scan Cell With Serial Scan Chain

```
cell (4-bit_Serial_Scan_Chain) {
```

Chapter 6: Defining Test Cells

Describing a Multibit Scan Cell

```
...
statetable ( " D      CK    SE     SI " ,      "Q" ) {
    table :   " - ~R   - - : - - : N, \
                H/L    R    L   - : - : H/L, \
                -     R    H   H/L : - : H/L" ;
}
bus(Q) {
    bus_type : bus4;
    direction : output;
    internal_node: Q;

    pin (Q[0]) { input_map : " D[0]  CK SE  SI " ; }
    pin (Q[1]) { input_map : " D[1]  CK SE  Q[0]  " ; }
    pin (Q[2]) { input_map : " D[2]  CK SE  Q[1]  " ; }
    pin (Q[3]) { input_map : " D[3]  CK SE  Q[2]  " ; }
    ...
}

/* dedicated scan output pin */
pin(SO) {
    direction : output;
    inverted_output : false;
    state_function : "Q[3] * SE" ;
    ...
}
pin(CK) {
    direction : input;
    ...
}
pin(SE) {
    direction : input;
    ...
}
/* scan input pin */
pin(SI) {
    direction : input;
    ...
}
/* data input bus pins */
bus(D) {
    direction : input;
    bus_type : bus4;
    ...
}
...
test_cell () {
    pin(CK) {
        direction : input;
    }
    bus(D) {
        bus_type : bus4;
        direction : input;
    }
    pin(SI) {
        direction : input;
    }
}
```

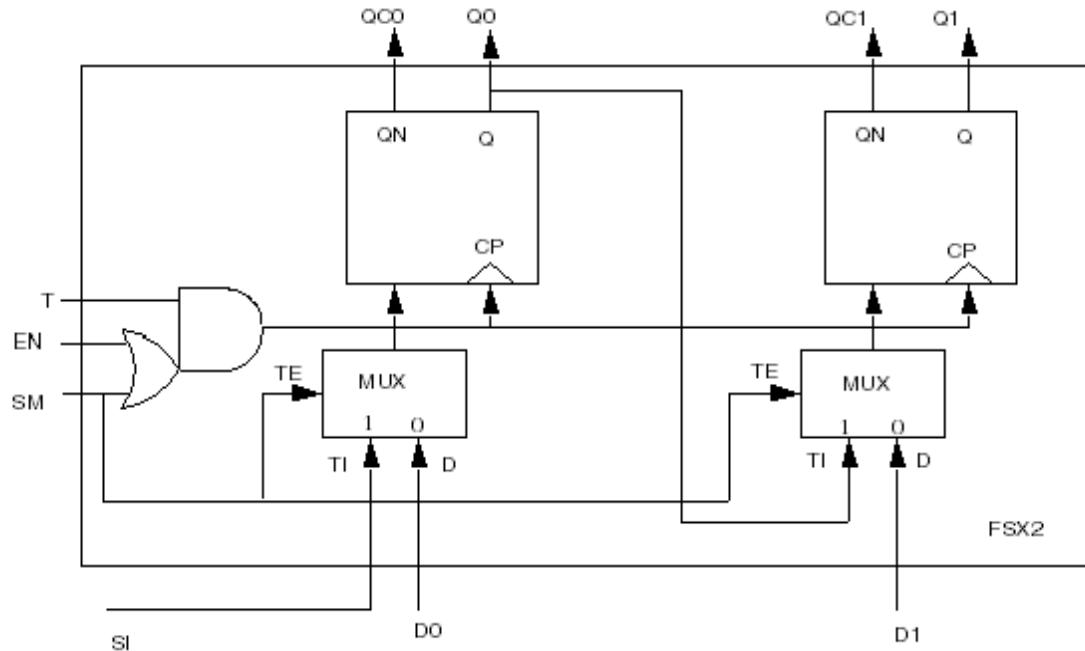
```
        signal_type : "test_scan_in";
    }
    pin(SE) {
        direction : input;
        signal_type : "test_scan_enable";
    }
    ff_bank (IQ,IQN,4) {
        next_state : "D";
        clocked_on : "CK";
    }
    bus(Q) {
        bus_type : bus4 ;
        direction : output;
        function : "IQ";
    }
    pin(SO) {
        direction : output;
        signal_type : "test_scan_out";
        test_output_only : "true";
    }
}
}
```

2-Bit Scan Cell Without Dedicated SO Pin

In [Figure 18](#), the scan cell has scan and enable inputs and no dedicated scan output pin. The output pin, Q1, is reused to output the scan signal.

Chapter 6: Defining Test Cells
 Describing a Multibit Scan Cell

Figure 18 2-Bit Scan Cell With Internal Scan Chain and Without SO Pin



The following example shows the relevant syntax to model the cell of [Figure 18](#). In this example, the `statetable` group describes the behavior of a single sequential element of the cell. The `input_map` statements on pin Q0 and pin Q1 indicate the interconnections between the sequential elements—for example, Q of bit 0 goes to TI of bit 1.

```
cell (FSX2) { ...
    bundle (D) {
        members (D0, D1);
        ...
    }
    bundle (Q) {

        members (Q0, Q1);
        ...
        pin (Q0) {
            input_map : "D0 T SI SM";
            ...
        }
        pin (Q1) {
            input_map : "D1 T Q0 SM";
            ...
        }
    }
    pin (SI) {
        ...
    }
    pin (SM) {
        ...
    }
}
```

```
pin (T) {
    ...
}
pin (EN) {
    ...
}
statetable ( "D CP TI TE EN", "Q QN") {
    /*D   CP   TI   TE   EN   Q   QN   Q+   QN+ */
    table : "- ~R - - - : - - : N N, \
              - - - L L: - - : N N, \
              - R H/L H - - : - - : H/L L/H, \
              H/L R - L H: - - : H/L L/H "
}
}
```

Scan Cell Modeling Examples

This section contains modeling examples for these test cells:

- [Simple Multiplexed D Flip-Flop](#)
 - [D Flip-Flop With Gated Output](#)
 - [Multibit Cells With Multiplexed D Flip-Flop and Enable](#)
 - [LSSD Scan Cell](#)
 - [Scan-Enabled LSSD Cell](#)
 - [Clocked-Scan Test Cell](#)
 - [Scan D Flip-Flop With Auxiliary Clock](#)
-

Simple Multiplexed D Flip-Flop

Example 56 Simple Multiplexed D Flip-Flop Scan Cell

```
cell(Sdff1) {
    area : 9;
    pin(D) {
        direction : input;
        capacitance : 1;
        timing() {...}
    }
    pin(CP) {
        direction : input;
        capacitance : 1;
        timing() {...}
    }
    pin(TI) {
        direction : input;
        capacitance : 1;
```

Chapter 6: Defining Test Cells

Scan Cell Modeling Examples

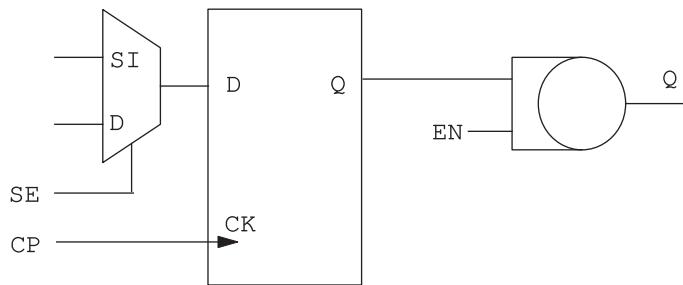
```
    timing() {...}
}
pin(TE) {
    direction : input;
    capacitance : 2;
    timing() {...}
}
ff(IQ,IQN) {
    /* model full behavior (if possible): */
    next_state : "D TE' + TI TE ";
    clocked_on : "CP";
}
pin(Q) {
    direction : output;
    function : "IQ";
    timing() {...}
}
pin(QN) {
    direction : output;
    function : "IQN";
    timing() {...}
}
test_cell() {
    pin(D) {
        direction : input
    }
    pin(CP) {
        direction : input
    }
    pin(TI) {
        direction : input;
        signal_type : test_scan_in;
    }
    pin(TE) {
        direction : input;
        signal_type : test_scan_enable;
    }
    ff(IQ,IQN) {
        /* just model nontest operation behavior */
        next_state : "D";
        clocked_on : "CP";
    }
    pin(Q) {
        direction : output;
        function : "IQ";
        signal_type : test_scan_out;
    }
    pin(QN) {
        direction : output;
        function : "IQN";
        signal_type : test_scan_out_inverted;
    }
}
```

}

D Flip-Flop With Gated Output

[Example 57](#) shows the modeling syntax for the D flip-flop with gated output shown in [Figure 19](#).

Figure 19 D Flip-Flop With Gated Output



Example 57 D Flip-Flop With Gated Output

```
cell(DFF1) {
    ...
    ff(IQ,IQN) {
        clocked_on : "CP";
        next_state : "((SE SI) + (!SE D))";
    }
    pin(Q) {
        direction : output;
        power_down_function : "!VDD + !VPP + VSS + VBB";
        function : "(IQ EN)";
        related_ground_pin : VSS;
    }
    ...
    test_cell() {
        pin(Q) {
            direction : output;
            function : "(IQ EN)";
            signal_type : test_scan_out;
        }
        pin(SI) {
            direction : input;
            signal_type : test_scan_in;
        }
        pin(D) {
            direction : input
        }
        pin(EN) {
```

```
    direction : input
}
pin(SE) {
    direction : input;
    signal_type : test_scan_enable;
}
pin(CP) {
    direction : input
}
ff(IQ,IQN) {
    clocked_on : "CP";
    next_state : "D";
}
}
```

Multibit Cells With Multiplexed D Flip-Flop and Enable

Example 58 Library Containing Multibit Scan Cells With Multiplexed D Flip-Flops and Enable

```
library(banktest) {
...
default inout pin_cap      : 1.0;
default input pin_cap      : 1.0;
default output pin_cap     : 0.0;
default fanout load        : 1.0;
time unit : "1ns";
voltage unit : "1V";
current unit : "1uA";
pulling resistance unit : "1kohm";
capacitive load unit (0.1,ff);
type (bus4) {
    base type : array;
    data type : bit;
    bit width : 4;
    bit from : 0;
    bit to   : 3;
}
cell(FDSX4) {
    area : 36;
    bus(D) {
        bus type : bus4;
        direction : input;
        capacitance : 1;
        timing() {
            timing type : setup_rising;
            related pin : "CP";
        }
        timing() {
            timing type : hold_rising;
            related pin : "CP";
        }
    }
}
```

Chapter 6: Defining Test Cells Scan Cell Modeling Examples

```
        }
    }
pin(CP) {
    direction : input;
    capacitance : 1;
}
pin(TI) {
    direction : input;
    capacitance : 1;
    timing() {
        timing_type : setup_rising;
        related_pin : "CP";
    }
    timing() {
        timing_type : hold_rising;
        related_pin : "CP";
    }
}
pin(TE) {
    direction : input;
    capacitance : 2;
    timing() {
        timing_type : setup_rising;
        related_pin : "CP";
    }
    timing() {
        timing_type : hold_rising;
        related_pin : "CP";
    }
}
statetable ( " D   CP   TI   TE   ", " Q   QN" ) {
    table : " - ~R   -   -   : -   - : N   N,   \
              -   R   H/L   H   : -   - : H/L   L/H,   \
              H/L   R   -   L   : -   - : H/L   L/H" ;
}
bus(Q) {
    bus_type : bus4;
    direction : output;
    inverted_output : false;
    internal_node : "Q";
    timing() {
        timing_type : rising_edge;
        related_pin : "CP";
    }
    pin(Q[0]) {
        input_map : "D[0] CP TI TE";
    }
    pin(Q[1]) {
        input_map : "D[1] CP Q[0] TE";
    }
    pin(Q[2]) {
        input_map : "D[2] CP Q[1] TE";
    }
}
```

Chapter 6: Defining Test Cells

Scan Cell Modeling Examples

```
pin(Q[3]) {
    input_map : "D[3] CP Q[2] TE";
}
}
bus(QN) {
    bus_type : bus4;
    direction : output;
    inverted_output : true;
    internal_node : "QN";
    timing() {
        timing_type : rising_edge;
        rise_transition(scalar) {values( " 0.1458 ");}
        fall_transition(scalar) {values( " 0.0523 ");}
        related_pin : "CP";
    }
    pin(QN[0]) {
        input_map : "D[0] CP TI    TE";
    }
    pin(QN[1]) {
        input_map : "D[1] CP Q[0] TE";
    }
    pin(QN[2]) {
        input_map : "D[2] CP Q[1] TE";
    }
    pin(QN[3]) {
        input_map : "D[3] CP Q[2] TE";
    }
}
test_cell() {
    bus (D) {
        bus_type : bus4;
        direction : input;
    }
    pin(CP) {
        direction : input;
    }
    pin(TI) {
        direction : input;
        signal_type : "test_scan_in";
    }
    pin(TE) {
        direction : input;
        signal_type : "test_scan_enable";
    }
    ff_bank("IQ","IQN", 4) {
        next_state : "D";
        clocked_on : "CP";
    }
    bus(Q) {
        bus_type : bus4;
        direction : output;
        function : "IQ";
        signal_type : "test_scan_out";
    }
}
```

Chapter 6: Defining Test Cells Scan Cell Modeling Examples

```
        }
    bus(QN) {
        bus_type : bus4;
        direction : output;
        function : "IQN";
        signal_type : "test_scan_out_inverted";
    }
}
cell(SCAN2) {
    area : 18;
    bundle(D) {
        members(D0, D1);
        direction : input;
        capacitance : 1;
        timing() {
            timing_type : setup_rising;
            related_pin : "T";
        }
        timing() {
            timing_type : hold_rising;
            related_pin : "T";
        }
    }
    pin(T) {
        direction : input;
        capacitance : 1;
    }
    pin(EN) {
        direction : input;
        capacitance : 2;
        timing() {
            timing_type : setup_rising;
            related_pin : "T";
        }
        timing() {
            timing_type : hold_rising;
            related_pin : "T";
        }
    }
    pin(SI) {
        direction : input;
        capacitance : 1;
        timing() {
            timing_type : setup_rising;
            related_pin : "T";
        }
        timing() {
            timing_type : hold_rising;
            related_pin : "T";
        }
    }
    pin(SM) {
```

Chapter 6: Defining Test Cells Scan Cell Modeling Examples

```
direction : input;
capacitance : 2;
timing() {
    timing_type : setup_rising;
    related_pin : "T";
}
timing() {
    timing_type : hold_rising;
    related_pin : "T";
}
statetable ( " T      D      EN      SI      SM",           " Q      QN" ) {
    table : " ~R      -      -      -      -      : -      -      : N      N , \
              -      -      L      -      L      : -      -      : N      N , \
              R      H/L     H      -      L      : -      -      : H/L     L/H , \
              R      -      -      H/L     H      : -      -      : H/L     L/H ";
}
bundle(Q) {
    members(Q0, Q1);
    direction : output;
    inverted_output : false;
    internal_node : "Q";
    timing() {
        timing_type : rising_edge;
        related_pin : "T";
    }
    pin(Q0) {
        input_map : "T D0 EN SI SM";
    }
    pin(Q1) {
        input_map : "T D1 EN Q0 SM";
    }
}
bundle(QN) {
    members(Q0N, Q1N);
    direction : output;
    inverted_output : true;
    internal_node : "QN";
    timing() {
        timing_type : rising_edge;
        related_pin : "T";
    }
    pin(Q0N) {
        input_map : "T D0 EN SI SM";
    }
    pin(Q1N) {
        input_map : "T D1 EN Q0 SM";
    }
}
test_cell() {
bundle (D) {
    members(D0, D1);
    direction : input;
```

```
        }
        pin(T) {
            direction : input;
        }
        pin(EN) {
            direction : input;
        }
        pin(SI) {
            direction : input;
            signal_type : "test_scan_in";
        }
        pin(SM) {
            direction : input;
            signal_type : "test_scan_enable";
        }
        ff_bank("IQ","IQN", 2) {
            next_state : "D";
            clocked_on : "T EN";
        }
        bundle(Q) {
            members(Q0, Q1);
            direction : output;
            function : "IQ";
            signal_type : "test_scan_out";
        }
        bundle(QN) {
            members(Q0N, Q1N);
            direction : output;
            function : "IQN";
            signal_type : "test_scan_out_inverted";
        }
    }
}
```

LSSD Scan Cell

Example 59 LSSD Scan Cell

```
cell(LSSD) {
    area : 12;
    pin(D) {
        direction : input;
        capacitance : 1;
        timing() {
            timing_type : setup_falling;
            related_pin : "MCLK";
        }
        timing() {
            timing_type : hold_falling;
            related_pin : "MCLK";
        }
    }
}
```

Chapter 6: Defining Test Cells Scan Cell Modeling Examples

```
        }
    pin(SI) {
        direction : input;
        capacitance : 1;
        prefer_tied : "0";
        timing() {
            timing_type : setup_falling;
            related_pin : "ACLK";
        }
        timing() {
            timing_type : hold_falling;
            related_pin : "ACLK";
        }
    }
    pin(MCLK, ACLK, SCLK) {
        direction : input;
        capacitance : 1;
    }
    pin(Q1) {
        direction : output;
        internal_node : "Q1";
        timing() {
            timing_type : rising_edge;
            related_pin : "MCLK";
        }
        timing() {
            timing_type : rising_edge;
            related_pin : "ACLK";
        }
        timing() {
            related_pin : "D";
        }
        timing() {
            related_pin : "SI";
        }
    }
    pin(Q1N) {
        direction : output;
        state_function : "Q1'";
        timing() {
            timing_type : rising_edge;
            related_pin : "MCLK";
        }
        timing() {
            timing_type : rising_edge;
            related_pin : "ACLK";
        }
        timing() {
            related_pin : "D";
        }
        timing() {
            related_pin : "SI";
        }
    }
```

Chapter 6: Defining Test Cells

Scan Cell Modeling Examples

```
        }
    pin(Q2) {
        direction : output;
        internal_node : "Q2";
        timing() {
            timing_type : rising_edge;
            related_pin : "SCLK";
        }
    }
    pin(Q2N) {
        direction : output;
        state_function : "Q2'";
        timing() {
            timing_type : rising_edge;
            related_pin : "SCLK";
        }
    }
    statetable("MCLK D      ACLK SCLK SI",           "Q1      Q2") {
        table :   " L   -   L   -   -   : -   -   : N   - , \
                   H   L/H  L   -   -   : -   -   : L/H  - , \
                   L   -   H   -   -   : L/H : -   -   : L/H  - , \
                   H   -   H   -   -   : -   -   : -   X   - , \
                   -   -   -   L   -   : -   -   : -   -   : N   , \
                   -   -   -   H   -   : L/H -   : -   -   : L/H ";
    }
    test_cell() /* for DLATCH */
    pin(D,MCLK) {
        direction : input;
    }
    pin(SI) {
        direction : input;
        signal_type : "test_scan_in";
    }
    pin(ACLK) {
        direction : input;
        signal_type : "test_scan_clock_a";
    }
    pin(SCLK) {
        direction : input;
        signal_type : "test_scan_clock_b";
    }
    latch ("IQ","IQN") {
        data_in : "D";
        enable : "MCLK";
    }
    pin(Q1) {
        direction : output;
        function : "IQ";
    }
    pin(Q1N) {
        direction : output;
        function : "IQN";
    }
}
```

Chapter 6: Defining Test Cells

Scan Cell Modeling Examples

```
pin(Q2) {
    direction : output;
    signal_type : "test_scan_out";
}
pin(Q2N) {
    direction : output;
    signal_type : "test_scan_out_inverted";
}
test_cell() { /* for MSFF1 */
    pin(D,MCLK,SCLK) {
        direction : input;
    }
    pin(SI) {
        direction : input;
        signal_type : "test_scan_in";
    }
    pin(ACLK) {
        direction : input;
        signal_type : "test_scan_clock_a";
    }
    ff ("IQ","IQN") {
        next_state : "D";
        clocked_on : "MCLK";
        clocked_on_also : "SCLK";
    }
    pin(Q1,Q1N) {
        direction : output;
    }
    pin(Q2) {
        direction : output;
        function : "IQ";
        signal_type : "test_scan_out";
    }
    pin(Q2N) {
        direction : output;
        function : "IQN";
        signal_type : "test_scan_out_inverted";
    }
}
```

Note:

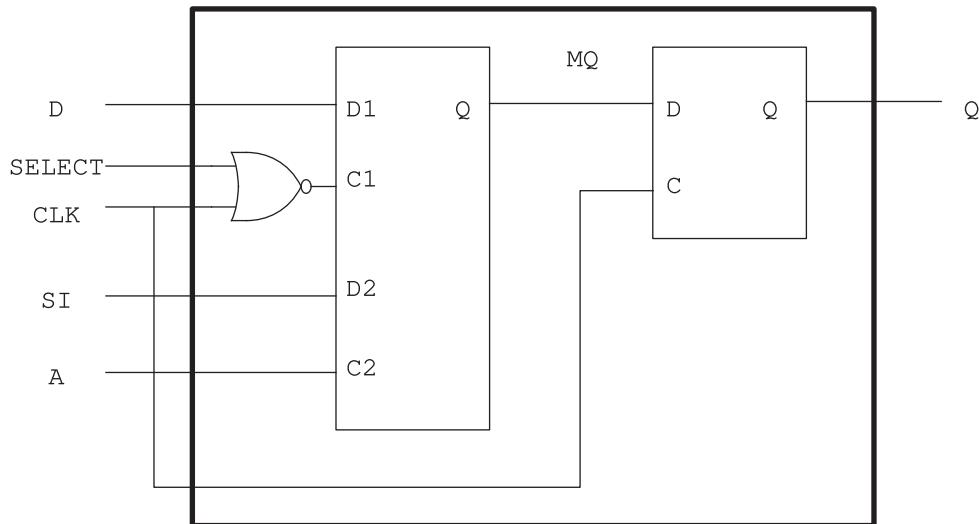
For latch-based designs, an LSSD scan cell has two `test_cell` groups for use in either single-latch or double-latch mode.

Scan-Enabled LSSD Cell

The scan-enabled LSSD cell is a variation of the LSSD scan cell in the double-latch mode. Unlike the double-latch LSSD scan cell, a single clock controls the enable pins of both the master and slave latches of the scan-enabled LSSD cell.

To recognize the scan-cell type, the `signal_type` attribute is used. If all the values of the `signal_type` attribute, namely, `test_scan_clock_a`, `test_scan_clock_b`, and `test_scan_enable` are present on the pins of a test cell, the corresponding scan cell is recognized as a scan-enabled LSSD.

Figure 20 Scan-Enabled LSSD Cell Schematic



Functional Model of the Scan-Enabled LSSD Cell

To model the cell shown in figure in [Scan-Enabled LSSD Cell](#), define the `signal_type` attribute on the pins of the `test_cell` group, such as the pins, CLK and SELECT.

[Example 60](#) shows the syntax to define the `signal_type` attribute on the CLK pin. When you set the `signal_type` attribute on the clock pin, CLK, to `test_scan_clock_b`, it indicates that the CLK input also enables the slave-latch in addition to the master-latch of the scan-enabled LSSD cell. In figure in [Scan-Enabled LSSD Cell](#), the clock pin, CLK, controls both enable pins, C1 and C.

Example 60 The signal_type attribute on the Scan-Enabled LSSD Cell CLK pin

```
cell(cell_name) {  
    ...  
    test_cell() {  
        ...  
    }  
}
```

```
pin (pin_name) {
    direction : input;
    signal_type : "test_scan_clock_b";
    ...
}/* End pin group */
}/* End test_cell */
...
}
```

[Example 61](#) shows the syntax to define the `signal_type` attribute on the select pin, SELECT. When you set the `signal_type` attribute on the select pin, SELECT, to `test_scan_enable`, the SELECT input is active and enables the scan mode of the LSSD cell. When the SELECT input is inactive, the cell is in the normal mode.

Example 61 The signal_type Attribute on the Scan-Enabled LSSD Cell SELECT Pin

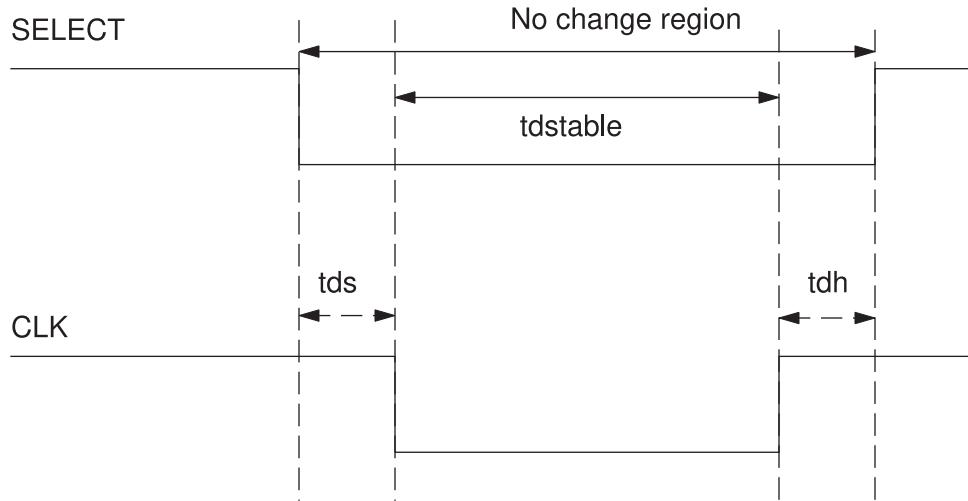
```
cell(cell_name) {
    ...
    test_cell() {
        ...
        pin (pin_name) {
            direction : input;
            signal_type : "test_scan_enable";
            ...
        }/* End pin group */
    }/* End test_cell */
    ...
}
```

Timing Model of the Scan-Enabled LSSD Cell

[Figure 21](#) shows a timing arc constraint, from the clock pin, CLK, to the select pin, SELECT, of the scan-enabled LSSD cell. In the normal mode, the constraint ensures that the CLK input works correctly for the duration of the CLK pulse. Therefore, the SELECT input must be stable for the setup period before the CLK pulse (`tds`), when the CLK pulse is active (`tdstable`), and the hold period after (`tdh`) the CLK pulse.

[Example 62](#) shows the syntax to model the timing arc, from the clock pin, CLK, to the select pin, SELECT. Timing arcs are modeled by setting the `timing_type` attribute to `nochange` values. As the CLK pin is active-low, set the `timing_type` attribute on the select pin, SELECT, to `nochange_low_low`.

Figure 21 A Timing-Arc Constraint of the Scan-Enabled LSSD Cell



Example 62 Scan-Enabled LSSD Timing Model Syntax

```
cell(cell_name) {
    ...
    pin (SELECT) {
        direction : input;
        timing() {
            timing_type: "nochange_low_low" ;
            related_pin: "CLK" ;
            fall_constraint(constraint) { /* tds */ ...
        }
        rise_constraint(constraint) { /* tdh */ ...
    }
    ...
}
```

Note:

Do not use the `hold_rising` (`tds`) and `setup_falling` (`tdh`) values to model, the clock pin, `CLK`, to the select pin, `SELECT`, timing arc. These values of the `timing_type` attribute do not cover the stable region of the `SELECT` input (`tdstable`).

Scan-Enabled LSSD Cell Model Example

[Example 63](#) uses the syntax in examples in [Functional Model of the Scan-Enabled LSSD Cell](#) and [Timing Model of the Scan-Enabled LSSD Cell](#) to model the scan-enabled LSSD cell shown in figure in [Scan-Enabled LSSD Cell](#).

Example 63 Example for the Scan-Enabled LSSD Cell Syntax

```
Cell(scan_enabled_LSSD) {
    ...
    statetable ("CLK D      SELECT A   SI",           "MQ   Q") {
        table : "      L   L/H   L      L - : - - : H/L - , \
                  H - - L - : - - : N - , \
                  L - H   L - : - - : N - , \
                           L - L   H - : - - : X - , \
                  H - L   H L/H : - - : L/H - , \
                  - - H   H L/H : - - : L/H - , \
                           L - - - - : - - : - - N , \
                  H - - - - : L/H - : - L/H";
    }
    pin (CLK) {
        direction : input ;
        timing() {
            timing_type: "min_pulse_width" ;
            related_pin: "CLK" ;
        ...
        }
    }
    pin (D) {
        direction : input ;
        timing() {
            timing_type: "hold_rising" ;
            related_pin: "CLK" ;
        ...
        }
        timing() {
            timing_type: "setup_rising" ;
            related_pin: "CLK" ;
        ...
        }
    }
    pin (SELECT) {
        direction : input;
        timing() {
            timing_type: "nochange_low_low" ;
            related_pin: "CLK" ;
        ...
        }
    }
    pin (SI) {
        direction : input;
        timing() {
```

Chapter 6: Defining Test Cells

Scan Cell Modeling Examples

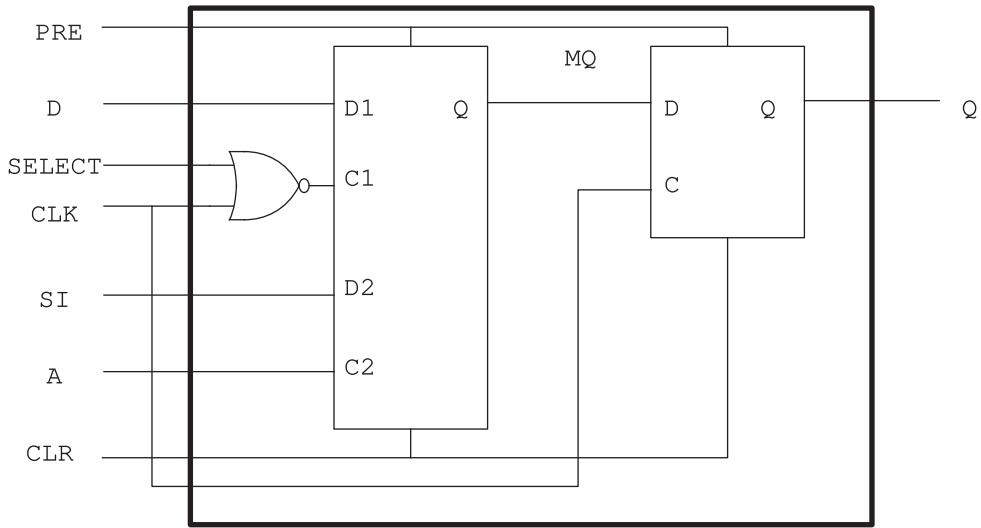
```
    timing_type: "setup_falling" ;
    related_pin: "A" ;
    ...
}
timing() {
    timing_type: "hold_falling" ;
    related_pin: "A"
    ...
}
pin (MQ) {
    direction : "internal";
    internal_node : "MQ";
}
pin (Q) {
    direction : output ;
    internal_node: "Q" ;
    timing() {
        timing_type: "rising_edge" ;
        related_pin: "CLK" ;
        ...
    }
}
...
test_cell () {
pin (CLK) {
    direction : input ;
    signal_type : "test_scan_clock_b";
}
pin (D) {
    direction : input ;
}
pin (A) {
    direction : input;
    signal_type : "test_scan_clock_a";
}
pin ("SELECT") {
    direction : input ;
    signal_type : "test_scan_enable";
}
pin (SI) {
    direction : input ;
    signal_type : "test_scan_in";
}
pin (Q) {
    direction : output ;
    function : "IQ";
    signal_type : "test_scan_out";
}
ff (IQ,IQN) {
    next_state : "D" ;
    clocked_on : "CLK" ;
}
```

```

}
...
}
```

Scan-Enabled LSSD Cell With Asynchronous Inputs

Figure 22 Scan-Enabled LSSD Cell With Asynchronous Inputs, PRE and CLR



[Example 64](#) shows the modeling syntax for the scan-enabled LSSD cell shown in Figure 22.

Example 64 Modeling Syntax for Scan-Enabled LSSD Cell With Preset and Clear Inputs

```

cell(LSSD_with_clear_preset) {
...
statetable (" CLK  D   SELECT  A  SI  CLR  PRE",      "MQ  Q") {
table : "
    L   L/H  L   -   L   L : - - : N   -,\n
    H   -     L   -   L   L : - - : N   -,\n
    L   -     L   -   L   L : - - : N   -,\n
                  L   -   H   -   L   L : - - : X   -,\n
    H   -     L   H   L/H L   L : - - : L/H   -,\n
    -   -     H   H   L/H L   L : - - : L/H   -,\n
                  L   -   -   -   L   L : - - : -   N,\n
    H   -     -   -   -   L   L : L/H - : -   L/H,\n
    -   -     -   -   -   H   L   L : - - : L   L,\n
    -   -     -   -   -   L   H   H : - - : H   H,\n
    -   -     -   -   -   H   H   H : - - : L   L";
}
...
test_cell () {
    pin (CLK) {
```

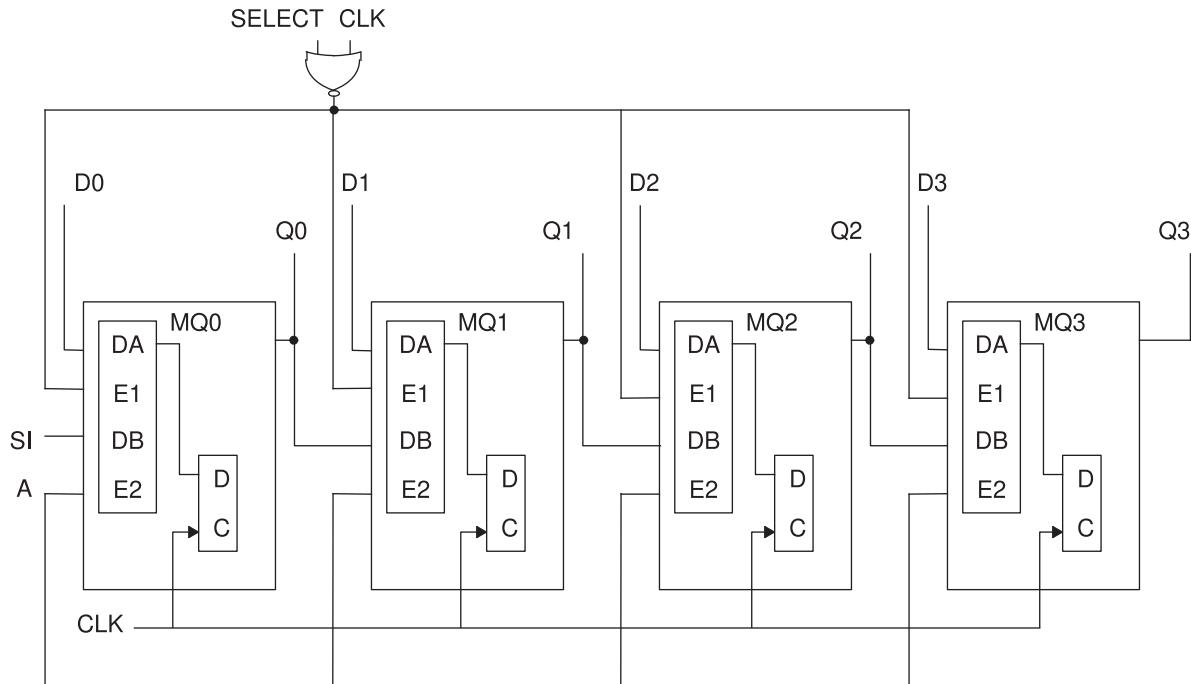
Chapter 6: Defining Test Cells

Scan Cell Modeling Examples

```
    direction : input ;
    signal_type : "test_scan_clock_b";
}
pin (D) {
    direction : input ;
}
pin (CLR) {
    direction : input ;
}
pin (PRE) {
    direction : input ;
}
pin (A) {
    direction : input;
    signal_type : "test_scan_clock_a";
}
pin ("SELECT") {
    direction : input ;
    signal_type : "test_scan_enable";
}
pin (SI) {
    direction : input ;
    signal_type : "test_scan_in";
}
pin (Q) {
    direction : output ;
    function : "IQ";
    signal_type : "test_scan_out";
}
ff (IQ,IQN) {
    next_state : "D" ;
    clocked_on : "CLK" ;
    clear : "CLR" ;
    preset : "PRE" ;
    clear_preset_var1 : L ;
    clear_preset_var2 : L ;
}
...
}
```

Multibit Scan-Enabled LSSD Cell

Figure 23 Schematic of a 4-Bit Scan-Enabled LSSD Cell



[Example 65](#) shows the modeling syntax for the cell shown in [Figure 23](#).

Example 65 4-Bit Scan-Enabled LSSD Cell Modeling Syntax

```
cell(LSSD_multibit) {
...
statetable (" CLK   D   SELECT  A      SI",      "MQ  Q") {
    table : "   L   L/H L   L   - : -   - : H/L -, \
              H   -   -   L   - : -   - : N   -, \
              L   -   H   L   - : -   - : N   -, \
                  L   -   L   L   H   - : -   - : X   -, \
              H   -   L   H   L/H : -   - : L/H -, \
              -   -   H   H   L/H : -   - : L/H -, \
                  L   -   -   -   - : -   - : -   N, \
              H   -   -   -   - : L/H - : -   L/H";
}
bus(MQ) {
    direction : internal;
    internal_node: MQ;
    bus_type : bus4;
```

Chapter 6: Defining Test Cells

Scan Cell Modeling Examples

```
pin (MQ[0]) { input_map : "CLK" D[0] SELECT A SI"; }
pin (MQ[1]) { input_map : "CLK" D[1] SELECT A Q[0]"; }
pin (MQ[2]) { input_map : "CLK" D[2] SELECT A Q[1]"; }
pin (MQ[3]) { input_map : "CLK" D[3] SELECT A Q[2]"; }

...
}

bus(Q) {
    direction : output;
    internal_node: Q;
    bus_type : bus4;
    pin (Q[0]) { input_map : "CLK" D[0] SELECT A SI MQ[0]"; }
    pin (Q[1]) { input_map : "CLK" D[1] SELECT A Q[0] MQ[1]"; }
    pin (Q[2]) { input_map : "CLK" D[2] SELECT A Q[1] MQ[2]"; }
    pin (Q[3]) { input_map : "CLK" D[3] SELECT A Q[2] MQ[3]"; }

...
}

...
test_cell() {
    bus(D) {
        bus_type : bus4;
        direction : input; }
    pin(CLK) {
        direction : input;
        signal_type : test_scan_clock_b; }
    pin(SI) {
        direction : input;
        signal_type : test_scan_in; }
    pin(A) {
        direction : input;
        signal_type : test_scan_clock_a; }
    pin(SELECT) {
        direction : input;
        signal_type : test_scan_enable; }
    ff_bank(IQ,IQN,4) {
        next_state : "D";
        clocked_on : "CLK";
    }
    bus(Q) {
        direction : output;
        bus_type : bus4;
        function : "IQ"; }
    pin(SO) {
        direction : output;
        signal_type : "test_scan_out"; }
} /* End of test_cell */
...
}
```

Clocked-Scan Test Cell

[Example 66](#) shows the model of a level-sensitive latch with separate scan clocking. This example shows the scan cell used in clocked-scan implementation.

Example 66 Clocked-Scan Test Cell

```
cell(SC_DLATCH) {
    area : 12;
    pin(D) {
        direction : input;
        capacitance : 1;
        timing() {
            timing_type : setup_falling;
            related_pin : "G";
        }
        timing() {
            timing_type : hold_falling;
            related_pin : "G";
        }
    }
    pin(SI) {
        direction : input;
        capacitance : 1;
        prefer_tied : "0";
        timing() {
            timing_type : setup_rising;
            related_pin : "ScanClock";
        }
        timing() {
            timing_type : hold_rising;
            related_pin : "ScanClock";
        }
    }
    pin(G,ScanClock) {
        direction : input;
        capacitance : 1;
    }
    statetable( "D      SI      G ScanClock",      " Q      QN" ) {
        table :   "L/H  -      H      L      : -      - : L/H  H/L,\n                  -      L/H  L      R      : -      - : L/H  H/L,\n                  -      -      L      ~R     : -      - : N      N";
    }
    pin(Q) {
        direction : output;
        internal_node : "Q";
        timing() {
            timing_type : rising_edge;
            related_pin : "G";
        }
        timing() {
            related_pin : "D";
        }
    }
}
```

Chapter 6: Defining Test Cells

Scan Cell Modeling Examples

```
        }
        timing() {
            timing_type : rising_edge;
            related_pin : "ScanClock";
        }
    }
    pin(QN) {
        direction : output;
        internal_node : "QN";
        timing() {
            timing_type : rising_edge;
            related_pin : "G";
        }
        timing() {
            related_pin : "D";
        }
        timing() {
            timing_type : rising_edge;
            related_pin : "ScanClock";
        }
        timing() {
            timing_type : rising_edge;
            related_pin : "ScanClock";
        }
    }
    test_cell() {
        pin(D,G) {
            direction : input;
        }
        pin(SI) {
            direction : input;
            signal_type : "test_scan_in";
        }
        pin(ScanClock) {
            direction : input;
            signal_type : "test_scan_clock";
        }
        pin(SE) {
            direction : input;
            signal_type : "test_scan_enable";
        }
        latch ("IQ","IQN") {
            data_in : "D";
            enable : "G";
        }
        pin(Q) {
            direction : output;
            function : "IQ";
            signal_type : "test_scan_out";
        }
        pin(QN) {
            direction : output;
            function : "IQN";
            signal_type : "test_scan_out_inverted";
        }
    }
}
```

```
        }
    }
}
```

Scan D Flip-Flop With Auxiliary Clock

Example 67 Scan D Flip-Flop With an Input and an Auxiliary Clock

```
cell(AUX_DFF1) {
    area : 12;
    pin(D) {
        direction : input;
        capacitance : 1;
        timing() {
            timing_type : setup_rising;
            related_pin : "CK";
        }
        timing() {
            timing_type : hold_rising;
            related_pin : "CK";
        }
        timing() {
            timing_type : setup_rising;
            related_pin : "IH";
        }
        timing() {
            timing_type : hold_rising;
            related_pin : "IH";
        }
    }
    pin(CK,IH,A,B) {
        direction : input;
        capacitance : 1;
    }
    pin(SI) {
        direction : input;
        capacitance : 1;
        prefer_tied : "0";
        timing() {
            timing_type : setup_falling;
            related_pin : "A";
        }
        timing() {
            timing_type : hold_falling;
            related_pin : "A";
        }
    }
    pin(Q) {
        direction : output;
        timing() {
            timing_type : rising_edge;
```

Chapter 6: Defining Test Cells

Scan Cell Modeling Examples

```

        related_pin : "CK IH";
    }
    timing() {
        timing_type : rising_edge;
        related_pin : "B";
    }
}
pin(QN) {
    direction : output;
    timing() {
        timing_type : rising_edge;
        related_pin : "CK IH";
    }
    timing() {
        timing_type : rising_edge;
        related_pin : "B";
    }
}
statetable( "C  TC  D  A  B  SI",  "IQ1  IQ2" ) {
    table :
    /* C  TC  D  A  B  SI          IQ1  IQ2 */
    H  -  -  L  -  - : -  -  : N  -, /* mast hold */
    -  H  -  L  -  - : -  -  : N  -, /* mast hold */
    H  -  -  H  -  L/H : -  -  : L/H  -, /* scan mast */
    -  H  -  H  -  L/H : -  -  : L/H  -, /* scan mast */
    L  L  L/H  L  -  - : -  -  : L/H  -, /* D in mast */
    L  L  -  H  -  - : -  -  : X  -, /* both active */
    H  -  -  -  L  -  : L/H  -  : -  L/H, /* slave loads */
    -  H  -  -  L  -  : L/H  -  : -  L/H, /* slave loads */
    L  L  -  -  -  : -  -  : -  N, /* slave loads */
    -  -  -  -  H  -  : -  -  : -  N; /* slave loads */
}
test_cell(){
    pin(D,CK){
        direction : input
    }
    pin(IH){
        direction : input;
        signal_type : "test_clock";
    }
    pin(SI){
        direction : input;
        signal_type : "test_scan_in";
    }
    pin(A){
        direction : input;
        signal_type : "test_scan_clock_a";
    }
    pin(B){
        direction : input;
        signal_type : "test_scan_clock_b";
    }
ff ("IQ","IQN") {

```

Chapter 6: Defining Test Cells

Scan Cell Modeling Examples

```
    next_state : "D";
    clocked_on : "CK";
}
pin(Q) {
    direction : output;
    function : "IQ";
    signal_type : "test_scan_out";
pin(QB) {
    direction : output;
    function : "IQN";
    signal_type : "test_scan_out_inverted";
}
}
```

7

Timing Arcs

Timing arcs are divided into two major areas: timing delays (the actual circuit timing) and timing constraints (at the boundaries). This chapter explains the timing concepts and describes the timing group attributes for setting constraints and defining delay.

The following sections describe how to specify timing delays, how to use timing constraints:

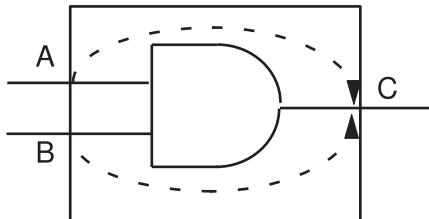
- [Understanding Timing Arcs](#)
- [Modeling Method Alternatives](#)
- [Defining the timing Group](#)
- [Describing Three-State Timing Arcs](#)
- [Describing Edge-Sensitive Timing Arcs](#)
- [Describing Clock Insertion Delay](#)
- [Describing Intrinsic Delay](#)
- [Describing Transition Delay](#)
- [Modeling Load Dependency](#)
- [Describing Slope Sensitivity](#)
- [Describing State-Dependent Delays](#)
- [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](#)
- [Impossible Transitions](#)

- Examples of NLDM Libraries
- Describing a Transparent Latch Clock Model
- Driver Waveform Support
- Sensitization Support
- Phase-Locked Loop Support

Understanding Timing Arcs

Timing arcs, along with netlist interconnect information, are the paths followed by the path tracer during path analysis. Each timing arc has a startpoint and an endpoint. The startpoint can be an input, output, or I/O pin. The endpoint is always an output pin or an I/O pin. The only exception is a constraint timing arc, such as a setup or hold constraint between two input pins. All delay information in a library refers to an input-to-output pin pair or an output-to-output pin pair.

Figure 24 *Timing Arcs (AC and BC) for AND Gate*



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

The Design Compiler tool uses combinational timing arc information to calculate the physical delays in timing propagation and to trace paths. The timing analyzer uses path-tracing arcs for circuit timing analysis.

The Design Compiler tool uses sequential timing arc information to determine rule-based design optimization constraints. See the *Synopsys Timing Constraints and Optimization User Guide* for more information about optimization constraints.

Combinational Timing Arcs

A combinational timing arc describes the timing characteristics of a combinational element. The timing arc is attached to an output pin, and the related pin is either an input or an output.

A combinational timing arc is of one of the following types:

- combinational
- combinational_rise
- combinational_fall
- three_state_disable
- three_state_disable_rise
- three_state_disable_fall
- three_state_enable
- three_state_enable_rise
- three_state_enable_fall

For information about describing combinational timing types, see [timing Group Attributes on page 220](#).

Sequential Timing Arcs

Sequential timing arcs describe the timing characteristics of sequential elements. In descriptions of the relationship between a clock transition and data output (input to output), the timing arc is considered a *delay* arc. In descriptions of the relationship between a clock transition and data input (input to input), the timing arc is considered a *constraint* arc.

A sequential timing arc is of one of the following types:

- Edge-sensitive (rising_edge or falling_edge)
- Preset or clear
- Setup or hold (setup_rising, setup_falling, hold_rising, or hold_falling)
- Nonsequential setup or hold (non_seq_setup_rising, non_seq_setup_falling, non_seq_hold_rising, non_seq_hold_falling)
- Recovery or removal (recovery_rising, recovery_falling, removal_rising, or removal_falling)

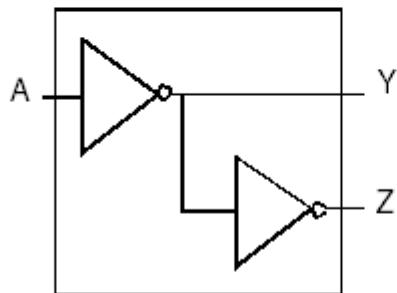
- No change (nochange_high_high, nochange_high_low, nochange_low_high, nochange_low_low)

For information about describing sequential timing types, see [timing Group Attributes on page 220](#).

Modeling Method Alternatives

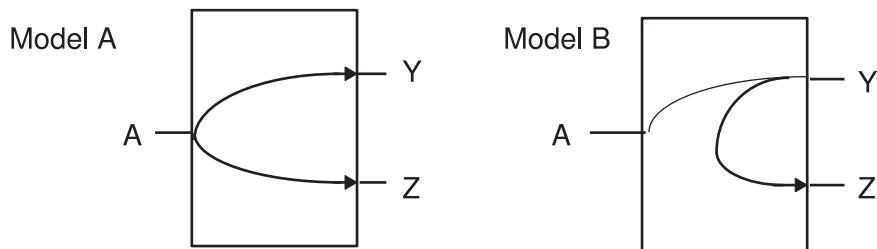
Timing information for combinational cells such as the one in [Figure 25](#) can be modeled in one of two ways, as [Figure 26](#) shows.

Figure 25 Two-Inverter Cell



In [Figure 26](#), 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 26 Two Modeling Techniques for Two-Inverter Cell



Model B for this cell also has two arcs but is more accurate than Model A. The first arc starts at pin A and ends at pin Y. This arc is modeled like the AY arc in Model A. The second arc is different; it starts with primary output Y and ends with primary output Z, modeling the effect the load on Y has on the delay on Z. Output-to-output timing arcs can be used in either combinational or sequential cells.

Defining the timing Group

The `timing` group contains information to model timing arcs and trace paths. The `timing` group defines the timing arcs through a cell and the relationships between clock and data input signals.

The `timing` group describes

- Timing relationships between an input pin and an output pin
- Timing relationships between two output pins
- Timing arcs through a noncombinational element
- Setup and hold times on flip-flop or latch input
- Optionally, the names of the timing arcs

The `timing` group describes setup and hold information when the constraint information refers to an input-to-input pin pair.

The `timing` group is defined in the `pin` group. This is the syntax:

```
library (lib_name) {  
    cell (cell_name) {  
        pin (pin_name) {  
            timing () {  
                ... timing description ...  
            }  
        }  
    }  
}
```

Define the `timing` group in the `pin` group of the endpoint of the timing arc, as illustrated by pin C in figure in [Understanding Timing Arcs](#).

Naming Timing Arcs Using the timing Group

Within the `timing` group, you can identify the name or names of different timing arcs.

A single timing arc can occur between an identified pin and a single related pin identified with the `related_pin` attribute.

Multiple timing arcs can occur in many ways. The following list shows six possible multiple timing arcs. The following descriptive sections explain how you configure other possible multiple timing arcs:

- Between a single related pin and the identified multiple members of a bundle.
- Between multiple related pins and the identified multiple members of a bundle.

- Between a single related pin and the identified multiple bits on a bus.
- Between multiple related pins and the identified multiple bits of a bus.
- Between the identified multiple bits of a bus and the multiple pins of related bus pins (of a designated width).
- Between the internal pin and all the bits of the endpoint `bus` group.

The following sections provide descriptions and examples for various timing arcs.

Timing Arc Between a Single Pin and a Single Related Pin

Identify the timing arc that occurs between a single pin and a single related pin by entering a name in the `timing` group, as shown in the following example:

Example

```
cell (my_inverter) {  
    ...  
    pin (A) {  
        direction : input;  
        capacitance : 1;  
    }  
    pin (B) {  
        direction : output  
        function : "A'";  
        timing (A_B) {  
            related_pin : "A";  
            ...  
        }/* end timing() */  
    }/* end pin B */  
}/* end cell */
```

The timing arc is as follows:

From pin	To pin	Label
A	B	A_B

Timing Arcs Between a Pin and Multiple Related Pins

This section describes how to identify the timing arcs when a `timing` group is within a `pin` group and the timing arc has more than a single related pin.

Example

You identify the multiple timing arcs on a name list entered with the `timing` group as shown in the following example.

Chapter 7: Timing Arcs

Defining the timing Group

```
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.

See the following example.

Example

```
...
bundle (Q) {
    members (Q0, Q1, Q2, Q3);
    direction : output;
    function : "IQ";
    timing (G_Q0, G_Q1, G_Q2, G_Q3) {
        timing_type : rising_edge;
        related_pin : "G";
    }
}
```

If G is a pin, as opposed to another `bundle` group, the timing arcs are as follows:

From pin	To pin	Label
G	Q0	G_Q0
G	Q1	G_Q1
G	Q2	G_Q2
G	Q3	G_Q3

If G is another bundle of member size 4 and G0, G1, G2, and G3 are members of bundle G, the timing arcs are as follows:

From pin	To pin	Label
G0	Q0	G_Q0
G1	Q1	G_Q1
G2	Q2	G_Q2
G3	Q3	G_Q3

Timing Arcs Between a Bundle and Multiple Related Pins

When the `timing` group is within a `bundle` group that has several members, each having a corresponding related pin, enter the names of the resulting multiple timing arcs as a name list in the `timing` group.

Example

```
bundle (Q) {
    members (Q0, Q1, Q2, Q3);
    direction : output;
    function : "IQ";
    timing (G_Q0, H_Q0, G_Q1, H_Q1, G_Q2, H_Q2, G_Q3, H_Q3) {
        timing_type : rising_edge;
        related_pin : "G H";
    }
}
```

If G is a pin, as opposed to another `bundle` group, the timing arcs are as follows:

From pin	To pin	Label
G	Q0	G_Q0

Chapter 7: Timing Arcs
Defining the timing Group

H	Q0	H_Q0
G	Q1	G_Q1
H	Q1	H_Q1
G	Q2	G_Q2
H	Q2	H_Q2
G	Q3	G_Q3
H	Q3	H_Q3

If G was another bundle of member size 4 and G0, G1, G2, and G3 are members of bundle G, the timing arcs are as follows:

From pin	To pin	Label
G0	Q0	G_Q0
H	Q0	H_Q0
G1	Q1	G_Q1
H	Q1	H_Q1
G2	Q2	G_Q2
H	Q2	H_Q2
G3	Q3	G_Q3
H	Q3	H_Q3

The same rule applies if H is a size-4 bundle.

Timing Arcs Between a Bus and a Single Related Pin

This section describes how to identify the timing arcs created when a `timing group` is within a `bus` group that has several bits with the same single related pin. You identify the resulting multiple timing arcs by entering a name list with the `timing group`.

Example

```
...
bus (X) {
/*assuming MSB is X[0] */
bus_type : bus4;
```

Chapter 7: Timing Arcs

Defining the timing Group

```
direction : output;
capacitance : 1;
pin (X[0:3]) {
    function : "B'";
    timing (B_X0, B_X1, B_X2, B_X3) {
        related_pin : "B";
    }
}
```

If B is a pin, as opposed to another 4-bit bus, the timing arcs are as follows:

From pin	To pin	Label
B	X[0]	B_X0
B	X[1]	B_X1
B	X[2]	B_X2
B	X[3]	B_X3

If B is another 4-bit bus and B[0] is the MSB for bus B, the timing arcs are as follows:

From pin	To pin	Label
B[0]	X[0]	B_X0
B[1]	X[1]	B_X1
B[2]	X[2]	B_X2
B[3]	X[3]	B_X3

Timing Arcs Between a Bus and Multiple Related Pins

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 ){
```

Chapter 7: Timing Arcs
Defining the timing Group

```
        related_pin : "B C";
    }
}
}
```

If B and C are pins, as opposed to another 4-bit bus, the timing arcs are as follows:

From pin	To pin	Label
B	X[0]	B_X0
C	X[0]	C_X0
B	X[1]	B_X1
C	X[1]	C_X1
B	X[2]	B_X2
C	X[2]	C_X2
B	X[3]	B_X3
C	X[3]	C_X3

If B is another 4-bit bus and B[0] is the MSB for bus B, the timing arcs are as follows:

From pin	To pin	Label
B[0]	X[0]	B_X0
C	X[0]	C_X0
B[1]	X[1]	B_X1
C	X[1]	C_X1
B[2]	X[2]	B_X2
C	X[2]	C_X2
B[3]	X[3]	B_X3
C	X[3]	C_X3

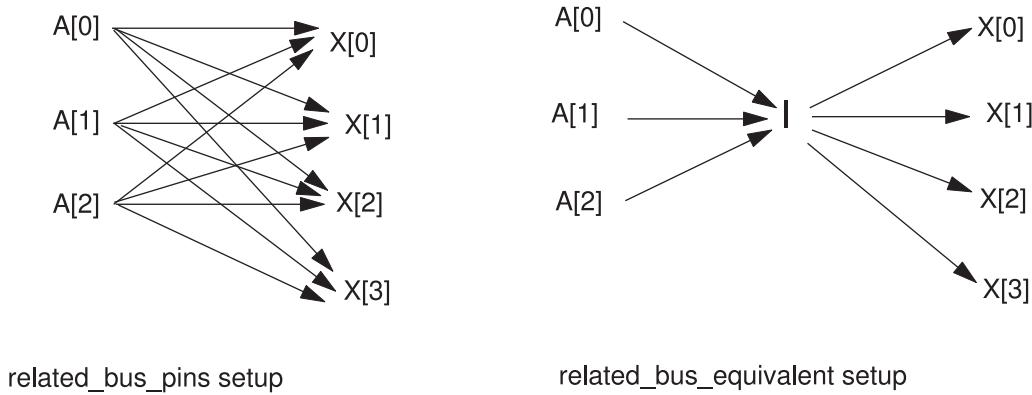
The same rule applies if C is a 4-bit bus.

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 27](#) compares the setup created with the `related_bus_pins` attribute with a setup created with the `related_bus_equivalent` attribute.

Figure 27 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.

It is assumed 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 Model

The `timing` groups are defined by the `timing` group attributes. The delay model determines the set of delay calculation attributes you specify in a `timing` group.

Cell delays are often modeled in a synthesis logic 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 logic library.

The NLDM is characterized by tables that define the timing arcs. To describe delay or constraint arcs with this model,

- Use the library-level `lu_table_template` group to define templates of common information to use in lookup tables.

- Use the templates and the timing groups described in this chapter to create lookup tables.

Lookup tables and their corresponding templates can be one-dimensional, two-dimensional, or three-dimensional. Delay arcs allow a maximum of two dimensions. Device degradation constraint tables allow only one dimension. Load-dependent constraint modeling requires three dimensions.

delay_model Attribute

To specify the delay model, use the `delay_model` attribute in the `library` group.

The `delay_model` attribute must be the first attribute in the `library` if a `technology` attribute is not present. Otherwise, it follows the `technology` attribute.

Example

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

Defining the NLDM Template

Table templates store common table information that multiple lookup tables can use. A table template specifies the table parameters and the breakpoints for each axis. Assign each template a name so that lookup tables can refer to it.

lu_table_template Group

Define your lookup table templates in the `library` group.

Syntax

```
lu_table_template(name) {  
    variable_1 : value;  
    variable_2 : value;  
    variable_3 : value;  
    index_1 ("float, ..., float");  
    index_2 ("float, ..., float");  
    index_3 ("float, ..., float");  
}
```

Template Variables for Timing Delays

The table template specifying timing delays can have up to three variables (`variable_1`, `variable_2`, and `variable_3`). The variables indicate the parameters used to index into the lookup table along the first, second, and third table axes. The parameters are the input transition time of a constrained pin, the output net length and capacitance, and the output loading of a related pin.

The following is a list of the valid values (divided into sets) that you can assign to a table:

- Set 1:

```
input_net_transition
```

- Set 2:

```
total_output_net_capacitance
output_net_length
output_net_wire_cap
output_net_pin_cap
```

- Set 3:

```
related_out_total_output_net_capacitance
related_out_output_net_length
related_out_output_net_wire_cap
related_out_output_net_pin_cap
```

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

For every table, the value you assign to a variable must be from a set different from the set from which you assign a value to the other variables. For example, if you want a two-dimensional table and you assign `variable_1` with the `input_net_transition` value from set 1, then you must assign `variable_2` with one of the values from set 2. [Table 16](#) lists the combinations of values you can assign to the different variables for the varying dimensional tables specifying timing delays.

Table 16 Variable Values for Timing Delays

Template dimension	Variable_1	Variable_2	Variable_3
1	set1		
1	set2		
2	set1	set2	
2	set2	set1	
3	set1	set2	set3
3	set1	set3	set2
3	set2	set1	set3
3	set2	set3	set1
3	set3	set1	set2

Template dimension	Variable_1	Variable_2	Variable_3
3	set3	set2	set1

Template Variables for Load-Dependent Constraints

The table template specifying load-dependent constraints can have up to three variables (`variable_1`, `variable_2`, and `variable_3`). The variables indicate the parameters used to index into the lookup table along the first, second, and third table axes. The parameters are the input transition time of a constrained pin, the transition time of a related pin, and the output loading of a related pin.

The following is a list of the valid values (divided into sets) that you can assign to a table.

- Set 1:

```
constrained_pin_transition
```

- Set 2:

```
related_pin_transition
```

- Set 3:

```
related_out_total_output_net_capacitance
related_out_output_net_length
related_out_output_net_wire_cap
related_out_output_net_pin_cap
```

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

For every table, the value you assign to a variable must be from a set different from the set from which you assign a value to the other variables. For example, if you want a two-dimensional table and you assign `variable_1` with the `constrained_pin_transition` value from set 1, then you must assign `variable_2` with one of the values from set 2.

[Table 17](#) lists the combination of values you can assign to the different variables for the varying dimensional tables specifying load-dependent constraints.

Table 17 Variable Values for Load-Dependent Constraint Tables

Template dimension	Variable_1	Variable_2	Variable_3
1	set1		
1	set2		
2	set1	set2	

Template dimension	Variable_1	Variable_2	Variable_3
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 237](#) 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.
- 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.

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 frequency
- 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_transition
constrained_pin_transition
```

- Set 2:

```
total_output_net_capacitance
output_net_length
```

Chapter 7: Timing Arcs

Defining the timing Group

```
output_net_wire_cap  
output_net_pin_cap  
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
```

- Set 4:

```
frequency  
temperature  
voltagei  
parametern
```

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

Table 18 Attributes in timing Group for NLDM

Purpose	Attribute or group
To specify a default timing arc	<code>default_timing</code>
To identify a timing arc startpoint	<code>related_pin</code> <code>related_bus_pins</code>
To describe a logical effect of input pin on output pin	<code>timing_sense</code>
To identify an arc as combinational or sequential	<code>timing_type</code>
To specify a propagation delay in total cell delay. (Used with transition delay)	<code>rise_propagation</code> <code>fall_propagation</code>

Purpose	Attribute or group
To specify a cell delay independent of transition delay	cell_rise cell_fall
To specify a retain delay within the delay arc	retaining_rise retaining_fall
To specify an output or I/O pin for load-dependency model	related_output_pin
To specify when a timing arc is active	mode

related_pin Simple Attribute

The `related_pin` attribute defines the pin or pins that are the startpoint of the timing arc. The primary use of `related_pin` is for multiple signals in ganged-logic timing. This attribute is a required component of all timing groups.

Example

```
pin (B) {
    direction : output ;
    function : "A'";
    timing () {
        related_pin : "A" ;
        ...
    }
}
```

You can use the `related_pin` attribute statement as a shortcut for defining two identical timing arcs for a cell. For example, for a 2-input NAND gate with identical delays from both input pins to the output pin, you need to define only one timing arc with two related pins, as shown in the following example.

```
pin (Z) {
    direction : output;
    function : "(A * B)'";
    timing () {
        related_pin : "A B" ;
        ... timing information ...
    }
}
```

When you use a bus name in a `related_pin` attribute statement, the bus members or the range of members is distributed across all members of the parent bus. In the following

example, the timing arcs described are from each element in bus A to each element in bus B: A(1) to B(1) and A(2) to B(2).

The width of the bus or range must be the same as the width of the parent bus.

```
bus (A) {  
    bus_type : bus2;  
    ...  
}  
bus (B) {  
    bus_type : bus2;  
    direction : output;  
    function : "A'";  
    timing () {  
        related_pin : "A" ;  
        ... timing information ...  
    }  
}
```

related_bus_pins Simple Attribute

The `related_bus_pins` attribute defines the pin or pins that are the startpoint of the timing arc. The primary use of `related_bus_pins` is for module generators.

Example

In this example, the timing arcs described are from each element in bus A to each element in bus B: A(1) to B(1), A(1) to B(2), A(1) to B(3), and so on. The widths of bus A and bus B do not need to be identical.

```
bus (A){  
    bus_type : bus2;  
    ...  
}  
bus (B){  
    bus_type : bus4;  
    direction : output ;  
    function : "A";  
    timing () {  
        related_bus_pins : "A" ;  
        ... timing information ...  
    }  
}
```

timing_sense Simple Attribute

The `timing_sense` attribute describes the way an input pin logically affects an output pin.

Example

```
timing () {  
    timing_sense : positive_unate;  
}
```

The Design Compiler tool typically derives the `timing_sense` value from the pin's logic function. For example, the value derived for an AND gate is `positive_unate`, the value for a NAND gate is `negative_unate`, and the value for an XOR gate is `non_unate`.

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.

A function is *unate* if a rising (or falling) change on a positive unate input variable causes the output function variable to rise (or fall) or not change. A rising (or falling) change on a negative unate input variable causes the output function variable to fall (or rise) or not change. For a nonunate variable, further state information is required to determine the effects of a particular state transition.

It is possible that one path is positive unate while another is negative unate. In this case, the first timing arc gets a `positive_unate` designation and the second arc gets a `negative_unate` designation.

Note:

When `timing_sense` describes the transition edge used to calculate delay for the `three_state_enable` or `three_state_disable` pin, it has a meaning different from its traditional one. If a 1 value on the control pin of a three-state cell causes a Z value on the output pin, `timing_sense` is `positive_unate` for the `three_state_disable` timing arc and `negative_unate` for the `three_state_enable` timing arc. If a 0 value on the control pin of a three-state cell causes a Z value on the output pin, `timing_sense` is `negative_unate` for the `three_state_disable` timing arc and `positive_unate` for the `three_state_enable` timing arc.

If a `related_pin` is an output pin, you must define `timing_sense` for that pin.

timing_type Simple Attribute

The `timing_type` attribute distinguishes between combinational and sequential cells, by defining the type of timing arc. If this attribute is not assigned, the cell is considered combinational.

You must distinguish between combinational and sequential timing types, because each type serves a different purpose.

The Design Compiler tool uses the combinational timing arcs information to calculate the physical delays in timing propagation and to trace paths. The timing analyzer uses path tracing arcs for circuit timing analysis.

The Design Compiler tool uses the sequential timing arcs information to determine rule-based design optimization constraints. More information about optimization constraints is available in the Design Compiler documentation.

Values for Combinational Timing Arcs

The timing type and timing sense define the signal propagation pattern. The default timing type is combinational.

Timing type	Timing sense		
	positive_unate	negative_unate	non_unate
combinational	R->R,F->F	R->F,F->R	{R,F}->{R,F}
combinational_rise	R->R	F->R	{R,F}->R
combinational_fall	F->F	R->F	{R,F}->F
three_state_disable	R->{0Z,1Z}	F->{0Z,1Z}	{R,F}->{0Z,1Z}
three_state_enable	R->{Z0,Z1}	F->{Z0,Z1}	{R,F}->{Z0,Z1}
three_state_disable_rise	R->0Z	F->0Z	{R,F}->0Z
three_state_disable_fall	R->1Z	F->1Z	{R,F}->1Z
three_state_enable_rise	R->Z1	F->Z1	{R,F}->Z1
three_state_enable_fall	R->Z0	F->Z0	{R,F}->Z0

Values for Sequential Timing Arcs

You use sequential timing arcs to model the timing requirements for sequential cells.

`rising_edge`

Identifies a timing arc whose output pin is sensitive to a rising signal at the input pin.

`falling_edge`

Identifies a timing arc whose output pin is sensitive to a falling signal at the input pin.

`preset`

Preset arcs affect only the rise arrival time of the arc's endpoint pin. A preset arc implies that you are asserting a logic 1 on the output pin when the designated related pin is asserted.

`clear`

Clear arcs affect only the fall arrival time of the arc's endpoint pin. A clear arc implies that you are asserting a logic 0 on the output pin when the designated related pin is asserted.

hold_rising

Designates the rising edge of the related pin for the hold check.

hold_falling

Designates the falling edge of the related pin for the hold check.

setup_rising

Designates the rising edge of the related pin for the setup check on clocked elements.

setup_falling

Designates the falling edge of the related pin for the setup check on clocked elements.

recovery_rising

Uses the rising edge of the related pin for the recovery time check. The clock is rising-edge-triggered.

recovery_falling

Uses the falling edge of the related pin for the recovery time check. The clock is falling-edge-triggered.

skew_rising

The timing constraint interval is measured from the rising edge of the reference pin (specified in `related_pin`) to a transition edge of the parent pin in the `timing group`. The `rise_constraint` value is the maximum skew time between the reference pin rising and the parent pin rising. The `fall_constraint` value is the maximum skew time between the reference pin falling and the parent pin falling.

skew_falling

The timing constraint interval is measured from the falling edge of the reference pin (specified in `related_pin`) to a transition edge of the parent pin in the `timing group`. The `rise_constraint` value is the maximum skew time between the reference pin falling and the parent pin rising. The `fall_constraint` value is the maximum skew time between the reference pin falling and the parent pin falling.

removal_rising

Used when the cell is a low-enable latch or a rising-edge-triggered flip-flop. For active-low asynchronous control signals, define the removal time with the `rise_constraint` group. For active-high asynchronous control signals, define the removal time with the `fall_constraint` group.

`removal_falling`

Used when the cell is a high-enable latch or a falling-edge-triggered flip-flop. For active-low asynchronous control signals, define the removal time with the `rise_constraint` group. For active-high asynchronous control signals, define the removal time with the `fall_constraint` group.

`min_pulse_width`

This value, together with the `minimum_period` value, lets you specify the minimum pulse width for a clock pin. The timing check is performed on the pin itself, so the related pin should be the same. You can also include rise and fall constraints, as with other timing checks.

Besides scalar values, table-based minimum pulse width is supported. For an example, see “A Library With `timing_type` Statements” (example in [min_pulse_width and minimum_period Example](#)).

`minimum_period`

This value, together with the `min_pulse_width` value, lets you specify the minimum pulse width for a clock pin. The timing check is performed on the pin itself, so the related pin should be the same. You can also include rise and fall constraints as with other timing checks.

`max_clock_tree_path`

Used in `timing` groups under a clock pin. Defines the maximum clock tree path constraint.

`min_clock_tree_path`

Used in `timing` groups under a clock pin. Defines the minimum clock tree path constraint.

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.

`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.

`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.

`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.

Values for No-Change Timing Arcs

You use no-change timing arcs to model the timing requirement for latch devices with latch-enable signals. The four no-change timing types define the pulse waveforms of both the constrained signal and the related signal in NLDM. The information is used in static timing verification during synthesis.

`nochange_high_high`

Indicates a positive pulse on the constrained pin and a positive pulse on the related pin.

`nochange_high_low`

Indicates a positive pulse on the constrained pin and a negative pulse on the related pin.

`nochange_low_high`

Indicates a negative pulse on the constrained pin and a positive pulse on the related pin.

`nochange_low_low`

Indicates a negative pulse on the constrained pin and a negative pulse on the related pin.

mode Complex Attribute

You define the `mode` attribute within a `timing group`. A `mode` attribute pertains to an individual timing arc. The timing arc is active when `mode` is instantiated with a name and a value. You can specify multiple instances of the `mode` attribute, but only one instance for each timing arc.

Syntax

```
mode (mode_name, mode_value);
```

Example

```
timing() {
    mode(rw, read);
}
```

Example 68 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 69 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) {
```

Chapter 7: Timing Arcs

Defining the timing Group

```
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 ");
        }
    }
}
```

Chapter 7: Timing Arcs

Defining the timing Group

```
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;
```

Chapter 7: Timing Arcs

Defining the timing Group

```
capacitance          : 1.0;
max_transition      : 2.0;
timing() {
    timing_type      : setup_rising;
    related_pin      : "CK";
    mode(RAM_MODE, MODE_1);
    rise_constraint(scalar) {
        values( " 0.0 ");
    }
    fall_constraint(scalar) {
        values( " 0.0 ");
    }
}
timing() {
    timing_type      : hold_rising;
    related_pin      : "CK";
    mode(RAM_MODE, MODE_1);
    rise_constraint(scalar) {
        values( " 0.0 ");
    }
    fall_constraint(scalar) {
        values( " 0.0 ");
    }
}
pin(CK) {
    timing() {
        timing_type : "min_pulse_width";
        related_pin : "CK";
        mode(RAM_MODE , MODE_4);
        fall_constraint(scalar) {
            values( " 0.0 ");
        }
        rise_constraint(scalar) {
            values( " 0.0 ");
        }
    }
    timing() {
        timing_type : "minimum_period";
        related_pin : "CK";
        mode(RAM_MODE , MODE_4);
        rise_constraint(scalar) {
            values( " 0.0 ");
        }
        fall_constraint(scalar) {
            values( " 0.0 ");
        }
    }
    clock           : true;
    direction       : input;
    capacitance     : 1.0;
    max_transition  : 1.0;
}
```

```
    cell_leakage_power : 0.0;  
}  
}
```

Describing Three-State Timing Arcs

Three-state arcs describe a three-state output pin in a cell.

Describing Three-State-Disable Timing Arcs

To designate a three-state-disable timing arc when defining a three-state pin,

1. Assign `related_pin` to the enable pin of the three-state function.
2. Define the 0-to-Z propagation time with the `cell_rise` or `rise_transition` statement.
3. Define the 1-to-Z propagation time with the `cell_fall` or `fall_transition` statement.
4. Include the `timing_type:three_state_disable` statement.

Example

```
timing () {  
    related_pin : "OE" ;  
    timing_type : three_state_disable ;  
/* 0 to Z modeling */  
    cell_rise(scalar) {  
        values( " 0.0 " );  
    }  
    rise_transition(scalar) {  
        values( " 0.0 " );  
    }  
  
/* 1 to Z modeling */  
    cell_fall(scalar) {  
        values( " 0.0 " );  
    }  
    fall_transition(scalar) {  
        values( " 0.0 " );  
    }  
}
```

Note:

The `timing_sense` attribute, which describes the transition edge used to calculate delay for a timing arc, has a nontraditional meaning when it is included in a timing group that also contains a `three_state_disable` attribute. See [timing_sense Simple Attribute on page 222](#) for more information.

Describing Three-State-Enable Timing Arcs

To designate a three-state-enable timing arc when defining a three-state pin,

1. Assign `related_pin` to the enable pin of the three-state function.
2. Define the Z-to-1 propagation time with the `cell_rise` or `rise_transition` group.
3. Define the Z-to-0 propagation time with the `cell_fall` or `fall_transition` group.
4. Include the `timing_type : three_state_enable` statement.

Example

```
timing () {
    related_pin : "OE" ;
    timing_type : three_state_enable ;

/* 0 to Z modeling */
    cell_rise(scalar) {
        values( " 0.0 ");
    }
    rise_transition(scalar) {
        values( " 0.0 ");
    }
/* 1 to Z modeling */
    cell_fall(scalar) {
        values( " 0.0 ");
    }
    fall_transition(scalar) {
        values( " 0.0 ");
    }
}
```

Example 70 Three-State Cell With Both Disable and Enable Timing Arcs

```
library(example) {
    date : "January 14, 2015";
    revision : 2015.01;
    technology (cmos);
    ...
    cell(TRI_INV2) {
        area : 3;
        pin(A) {
            direction : input;
            capacitance : 2;
        }
        pin(E) {
            direction : input;
            capacitance : 2;
        }
        pin(Z) {
            direction : output;
```

Chapter 7: Timing Arcs

Describing Edge-Sensitive Timing Arcs

```
function : "A'";
three_state : "E'";
timing() {
    related_pin : "A";
}
timing() {
    timing_type : three_state_enable;
    related_pin : "E";
}
timing() {
    timing_type : three_state_disable;
    related_pin : "E";
}
}
```

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 222](#) for more information.

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 223](#) for information about the `timing_type` attribute.

The following example shows the timing arc for the QN pin of a JK flip-flop.

Example

```
pin(QN) {
    direction : output ;
```

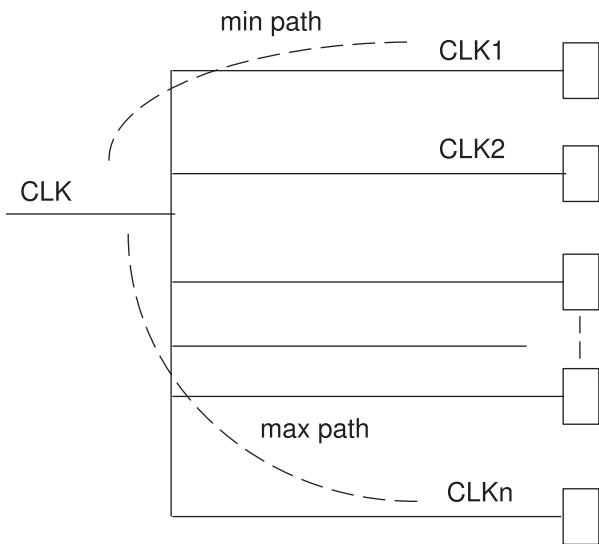
```
function : "IQN" ;
timing() {
    related_pin : "CP" ;
    timing_type : rising_edge ;
}
}
```

The QN pin makes a transition after the clock signal rises.

Describing 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 28](#).

Figure 28 Minimum and Maximum Clock Tree Paths



The `max_clock_tree_path` and `min_clock_tree_path` attributes let you define the maximum and minimum clock tree path constraints.

The clock tree path for any one clock can have up to eight values depending on the unateness of the pins and the fastest and slowest paths.

You can use lookup tables to model the cell delays. Lookup tables are indexed only by the input transition time of the clock.

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 to model both positive unate and negative unate.

Describing Intrinsic Delay

The intrinsic delay of an element is the zero-load (fixed) component of the total delay equation. Intrinsic delay attributes have different meanings, depending on whether they are for an input or an output pin.

When describing an output pin, the intrinsic delay attributes define the fixed delay from input to output pin. These values are used to calculate the intrinsic delay of the total delay equation.

When describing an input pin, such as in a setup or hold timing arc, intrinsic attributes define the timing requirements for that pin. Timing constraints are not used in the delay equation. The description of intrinsic delay is inherent in the lookup tables you create.

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.

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 218](#) 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.

Transition time is the time it takes for an output signal to make a transition between the high and low logic states. It is computed by table lookup and interpolation. Transition delay is a function of capacitance at the output pin and input transition time.

Defining Delay Arcs With Lookup Tables

These timing group attributes provide valid lookup tables for delay arcs:

- cell_rise
- cell_fall
- rise_propagation
- fall_propagation
- retaining_rise
- retaining_fall
- retain_rise_slew
- retain_fall_slew

Note:

For timing groups with timing type clear, only fall groups are valid. For timing groups with timing type preset, only rise groups are valid.

There are two methods for defining delay arcs. Choose the method that best fits your library data characterization. .

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

retaining_rise and retaining_fall Groups

The retaining delay is the time during which an output port retains its current logical value after a voltage rise or fall at a related input port.

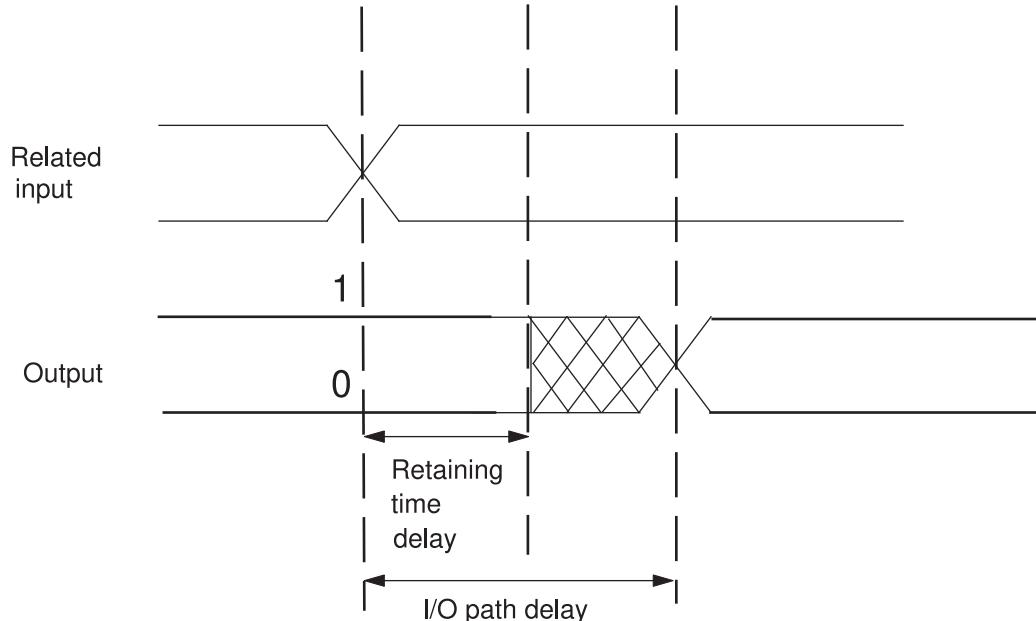
The retaining delay is part of the arc delay (I/O path delay); therefore, its time cannot exceed the arc delay time. Because retaining delay is part of the arc delay, the retaining delay tables are placed within the timing arc.

The value you enter for the `retaining_rise` attribute determines how long the output pin retains its current value, 0, after the value at the related input port has changed.

The value you enter for the `retaining_fall` attribute determines how long the output retains its current value, 1, after the value at the related input port has changed.

[Figure 29](#) shows retaining delay in regard to changes in a related input port.

Figure 29 Retaining Time Delay



[Example 71](#) shows how to use the `retaining_rise` and `retaining_fall` attributes.

Example 71 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 56](#) for information about calculating delay factors.

retain_rise_slew and retain_fall_slew Groups

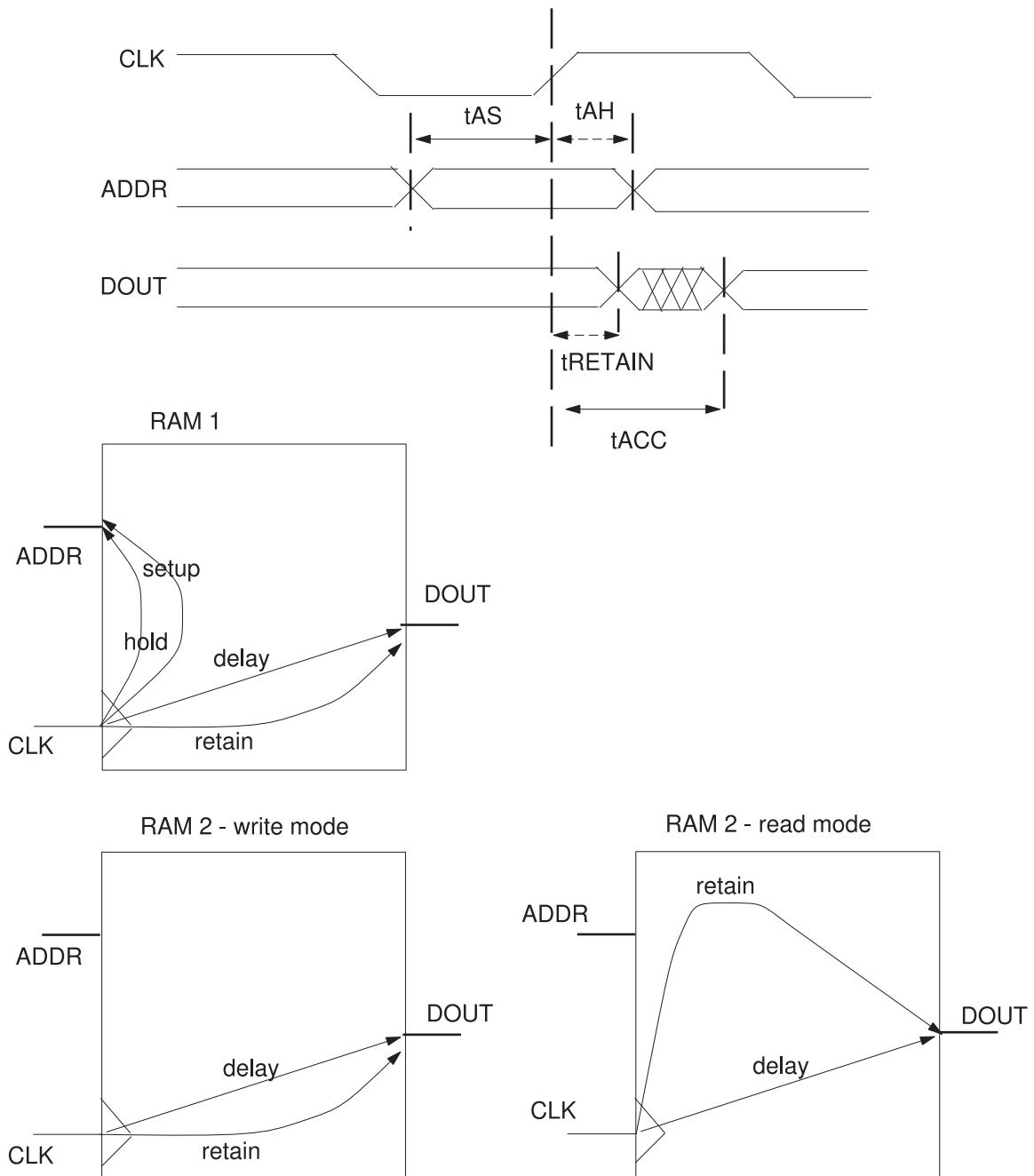
These groups let you specify a slew table for the retain arc that is separate from the table of the parent delay arc. This retain arc represents the time it takes until an output pin starts losing its current logical value after a related input pin is changed. This decaying of the output logic value occurs at a different time than the propagation of the final logical value, and at a different rate.

The retain delay is part of the arc delay (I/O path delay), and therefore its time cannot exceed the arc delay time. Because the retain delay is part of the arc delay, the retain delay tables are placed within the timing arc.

The value you enter for the `retain_rise_slew` attribute determines how long the output pin retains its current value, 0, after the value at the related input port has changed.

The value you enter for the `retain_fall_slew` attribute determines how long the output retains its current value, 1, after the value at the related input port has changed.

Figure 30 Timing Diagram of Synchronous RAM



Example

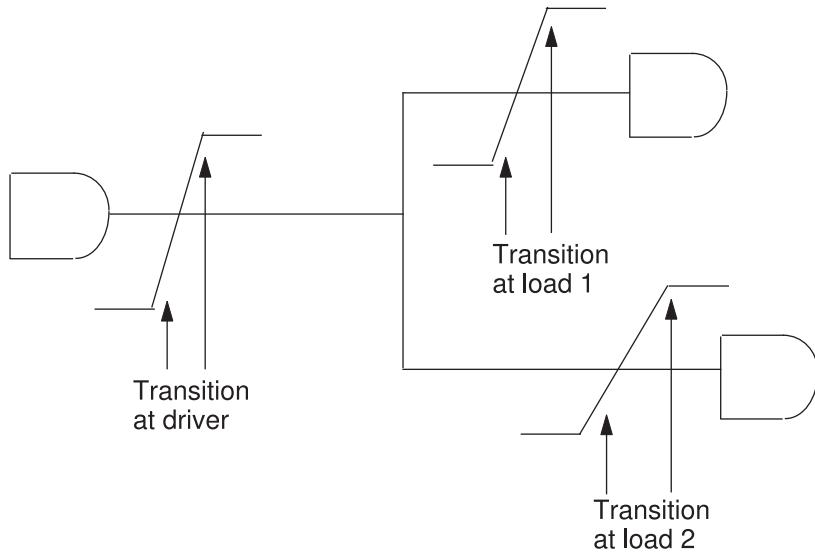
```
library(library_name) {
    ...
    lu_table_template (retaining_table_template) {
        ...
        variable_1: total_output_net_capacitance;
        variable_2: input_net_transition;
        index_1 ("0.0, 1.5");
        index_2 ("1.0, 2.1");
    }
    ...
    cell (cell_name) {
        ...
        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 56](#) for information about calculating delay factors.

Modeling Transition Time Degradation

Current nonlinear delay models are based on the assumption that the transition time at the load pin is the same as the transition time created at the driver pin. In reality, the net acts as a low-pass filter and the transition flattens out as it propagates from the driver of the net to each load, as shown in [Figure 31](#). The higher the interconnect load, the greater the flattening effect and the greater the transition delay.

Figure 31 Transition Time Degradation



To model the degradation of the transition time as it propagates from an output pin over the net to the next input pin, include these library-level groups in your library:

- `rise_transition_degradation`
- `fall_transition_degradation`

These groups contain the values describing the degradation functions for rise and fall transitions in the form of lookup tables. The lookup tables are indexed by

- Transition time at the net driver
- Connect delay between the driver and a load

These are the supported values for transition degradation (`variable_1` and `variable_2`):

- `output_pin_transition`
- `connect_delay`

You can assign either `connect_delay` or `output_pin_transition` to `variable_1` or `variable_2`, if the index and table values are consistent with the assignment.

The values you use in the table compute the degradation transition according to the following formula:

```
degraded_transition =  
table_lookup(f(output_pin_transition, connect_delay))
```

Design Compiler uses transition degradation tables when it indexes into any delay table in a library that uses the table parameters `input_net_transition`, `constrained_pin_transition`, or `related_pin_transition` in an `lu_table_template` group. Transition degradation tables in a library have no effect on computations related to cells from other libraries.

The k-factors for process, voltage, and temperature are not supplied for the new tables. The `output_pin_transition` value and the `connect_delay` value are computed at the current, rather than nominal, operating conditions.

In Example 72, `trans_deg` is the name of the template for the transition degradation.

Example 72 Using Degradation Tables

```
library (simple_tlu) {
delay_model : table_lookup;

/* define the table templates */

lu_table_template(prop) {
    variable_1 : input_net_transition ;
    variable_2 : total_output_net_capacitance ;
    index_1("0, 1, 2");
    index_2("0, 1, 2");
}

lu_table_template(tran) {
    variable_1 : total_output_net_capacitance ;
    variable_2 : input_net_transition ;
    index_1("0, 1, 2");
    index_2("0, 1, 2");
}

lu_table_template(constraint) {
    variable_1 : constrained_pin_transition ;
    index_1("0, 1, 2");
    variable_2 : related_pin_transition ;
    index_2("0, 1, 2");
}

lu_table_template(trans_deg) {
    variable_1 : output_pin_transition ;
    index_1("0, 1");
    variable_2 : connect_delay ;
    index_2("0, 1");
}

/* the new degradation tables */

rise_transition_degradation(trans_deg) {
    values("0.0, 0.6", "1.0, 1.6");
}
```

```
fall_transition_degradation(trans_deg) {
    values("0.0, 0.8", "1.0, 1.8");
}

/* other library level defaults */

default inout_pin_cap : 1.0;
...
k_process_fall_transition : 1.0;
...

nom_process : 1.0;
nom_temperature : 25.0;
nom_voltage : 5.0;

operating_conditions(BASIC_WORST) {
    process : 1.5 ;
    temperature : 70 ;
    voltage : 4.75 ;
    tree_type : "worst_case_tree" ;
}

/* list of cell descriptions */
cell(AN2) {
    ...
}
```

Any linear function of `output_pin_transition` and `connect_delay` can be represented with four table points, because the Design Compiler interpolation and extrapolation mechanism is linear. Larger tables are required to represent more complex degradation functions, and breakpoints must be chosen so that the interpolation error is acceptable.

Modeling Load Dependency

[Describing Transition Delay on page 236](#) describes how to model the transition time dependency of a constrained pin and its related pin on timing constraints. You can further model the effect of unbuffered output on timing constraints by modeling load dependency. Constraints can also be load-dependent.

This is the procedure for modeling load dependency.

1. In the `timing` group of the output pin, set the `timing_type` attribute .
2. Use the `related_output_pin` attribute in the `timing` group to specify which output pin to use to calculate the load dependency.
3. Create a three-dimensional table template that uses two variables and indexes to model transition time and the third variable and index to model load. The variable values for representing output loading on the `related_output_pin` are

```
related_out_total_output_net_capacitance
related_out_output_net_length
related_out_output_net_wire_cap
related_out_output_net_pin_cap
```

See [Defining the NLDM Template on page 214](#).

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 217](#).) The following groups are valid lookup tables for output load modeling:

- `rise_constraint`
- `fall_constraint`

See [Setting Setup and Hold Constraints on page 252](#) for information about these groups.

Example 73 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 value.
2. Specify the output pin used to figure the load dependency with the `related_output_pin` attribute described later.
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 218](#).

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 coefficients, using the `variable_3` variable and the `variable_3_range` attribute in the `poly_template` group.

- `rise_constraint`
- `fall_constraint`

[Example 74](#) is an example of a library that includes a load-dependent model.

Example 74 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");
}
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");
}
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.

Describing State-Dependent Delays

These timing attributes describe the delay values for specified conditions.

In the `timing` group of a logic library, you need to specify state-dependent delays that correspond to entries in Open Verilog International Standard Delay Format (OVI SDF 2.1) syntax.

To define a state-dependent timing arc, you can specify the following attributes in each `timing` group:

- `when`

- char_when
- char_when_rise
- char_when_fall
- sdf_cond

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 [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.

Table 19 Valid Boolean Operators

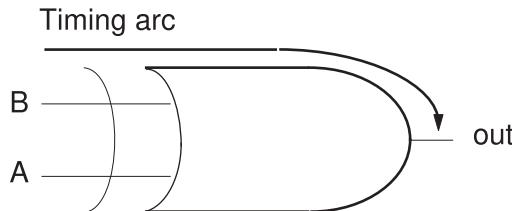
Operator	Description
,	Invert previous expression
!	Invert following expression
^	Logical XOR
*	Logical AND
&	Logical AND
space	Logical AND
+	Logical OR
	Logical OR
1	Signal tied to logic 1
0	Signal tied to logic 0

The order of precedence of the operators is left to right, with inversion performed first, then XOR, then AND, then OR.

Example

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

Figure 32 XOR Gate With State-Dependent Timing Arc



The example in [sdf_cond Simple Attribute](#) 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 223](#) 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";
```

Synopsys tools, such as PrimeTime, Design Compiler, and IC Compiler use the `when` expression instead of the `sdf_cond` expression. For a library file with the `sdf_cond` attribute, these tools:

- Read the corresponding SDF file without issuing an error.
- Write the `sdf_cond` expressions in the SDF file.

The `sdf_cond` attribute must be logically equivalent to the `when` attribute for the same timing arc. If the two Boolean expressions are not equivalent, back-annotation is not performed properly.

The `sdf_cond` expressions must be syntax-compliant with SDF 2.1. If the expressions do not meet this standard, errors are generated later in the flow during the generation and reuse of the SDF files.

For simple delay paths, such as IOPATH, you can use the Boolean operators, such as `&&` and `||`, with the `sdf_cond` attribute. However, Verilog timing check statements, including setup, hold, recovery, and removal do not support Boolean operators.

Chapter 7: Timing Arcs

Describing State-Dependent Delays

In [Example 75](#), the delay between pin A and pin OUT is 1.3 for rising and 1.5 for falling when pin B = 1. There is an additional timing arc between the same two pins that has a rise delay of 1.4 and a fall delay of 1.6 when pin B = 0.

Example 75 2-Input XOR Cell With State-Dependent Timing

```
cell(XOR) {
    pin(A) {
        direction : input;
        ...
    }
    pin(B) {
        direction : input;
        ...
    }
    pin(out) {
        direction : output;
        function : "A ^ B";
        timing() {
            related_pin : "A";
            timing_sense : negative_unate;
            when : "B";
            sdf_cond : " B == 1'B1 ";
            cell_rise(scalar) {
                values( " 1.3 " );
            }
            cell_fall(scalar) {
                values( " 1.5 " );
            }
        }
        timing() {
            related_pin : "A";
            timing_sense : positive_unate;
            when : "!B";
            sdf_cond : " B == 1'B0 ";
            cell_rise(scalar) {
                values( " 1.4 " );
            }
            cell_fall(scalar) {
                values( " 1.6 " );
            }
        }
        timing() /* default timing arc */
        related_pin : "A";
        timing_sense : non_unate;
        cell_rise(scalar) {
            values( " 1.4 " );
        }
        cell_fall(scalar) {
            values( " 1.6 " );
        }
    }
    timing() {
```

```
related_pin : "B";
timing_sense : negative_unate;
when : "A";
sdf_cond : "A == 1'B1 ";
cell_rise(scalar) {
    values( " 1.3 " ) ;
}
cell_fall(scalar) {
    values( " 1.5 " ) ;
}
}
timing() {
    related_pin : "B";
    timing_sense : positive_unate;
    when : "!A";
    sdf_cond : "A == 1'B0 ";
    cell_rise(scalar) {
        values( " 1.4 " ) ;
    }
    cell_fall(scalar) {
        values( " 1.6 " ) ;
    }
}
timing() { /* default timing arc */
    related_pin : "B";
    timing_sense : non_unate;
    cell_rise(scalar) {
        values( " 1.4 " ) ;
    }
    cell_fall(scalar) {
        values( " 1.6 " ) ;
    }
}
}
```

Setting Setup and Hold Constraints

Signals arriving at an input pin have ramp times. Therefore, you must ensure that the data signal has stabilized before latching its value by defining setup and hold arcs as timing requirements.

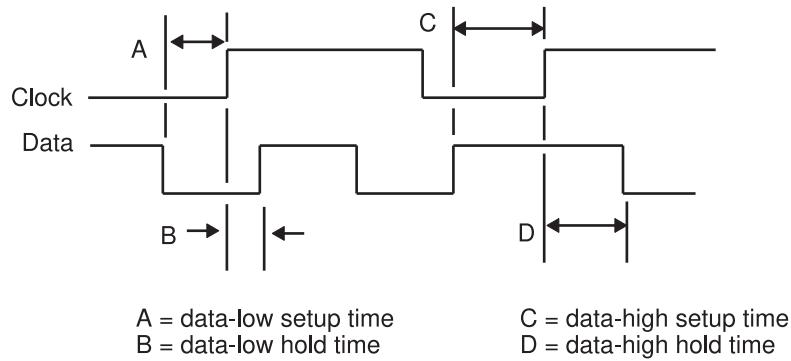
- Setup constraints describe the minimum time allowed between the arrival of the data and the transition of the clock signal. During this time, the data signal must remain constant. If the data signal makes a transition during the setup time, an incorrect value might be latched.
- Hold constraints describe the minimum time allowed between the transition of the clock signal and the latching of the data. During this time, the data signal must remain

constant. If the data signal makes a transition during the hold time, an incorrect value might be latched.

By combining a setup time and a hold time, you can ensure the stability of data that is latched.

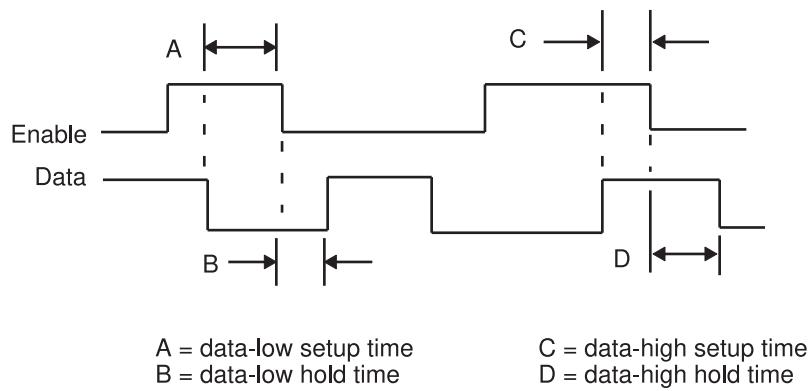
The timing checks for flip-flops use the activating edge of the clock, which is the rising edge for the case shown in [Figure 33](#).

Figure 33 Setup and Hold Constraints for Rising-Edge-Triggered Flip-Flop



The timing checks for latches generally use the deactivating edge of the enable signal, which is the falling edge for the case shown in [Figure 34](#). However, the method used depends on the vendor.

Figure 34 Setup and Hold Constraints for High-Enable Latch



NLDM supports timing constraints sensitive to clock or data-input transition times.

Each constraint is defined by a `timing` group with two lookup tables:

- `rise_constraint` group
- `fall_constraint` group

`rise_constraint` and `fall_constraint` Groups

These are constraint tables. The format of the lookup table template and the format of the lookup table are the same as described previously in [Defining the NLDM Template on page 214](#) and [Creating Lookup Tables on page 217](#).

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 76 Using Lookup Tables to Specify Setup Constraints for Flip-Flop

```
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 77](#) shows how to specify constraint in a scalable polynomial delay model.

Example 77 CMOS Scalable Polynomial Delay Model Using Constraint

```
library(vendor_b) {
```

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..

Setting Nonsequential Timing Constraints

You can set constraints requiring that the data signal on an input pin remain stable for a specified amount of time before or after another pin in the same cell changes state. These cells are termed nonsequential cells, because the related pin is not a clock signal.

All Synopsys delay models supporting sequential setup and hold constraint modeling also support nonsequential setup and hold modeling.

Scaling of nonsequential setup and hold constraints based on the environment use k-factors for sequential setup and hold constraints.

The values you can assign to a `timing_type` attribute to model nonsequential setup and hold constraints are

`non_seq_setup_rising`

Designates the rising edge of the related pin for the setup check.

`non_seq_setup_falling`

Designates the falling edge of the related pin for the setup check.

`non_seq_hold_rising`

Designates the rising edge of the related pin for the hold check.

`non_seq_hold_falling`

Designates the falling edge of the related pin for the hold check.

To model nonsequential setup and hold constraints for a cell,

1. Assign a value to the `timing_type` attribute in a `timing` group of an input or I/O pin.
2. Specify a related pin with the `related_pin` attribute in the `timing` group. The related pin in a timing arc is the pin used for the timing check.

Use any pin in the same cell, except for output pins, and the constrained pin itself as the related pin.

You can use both rising and falling edges as the active edge of the related pin for one cell.

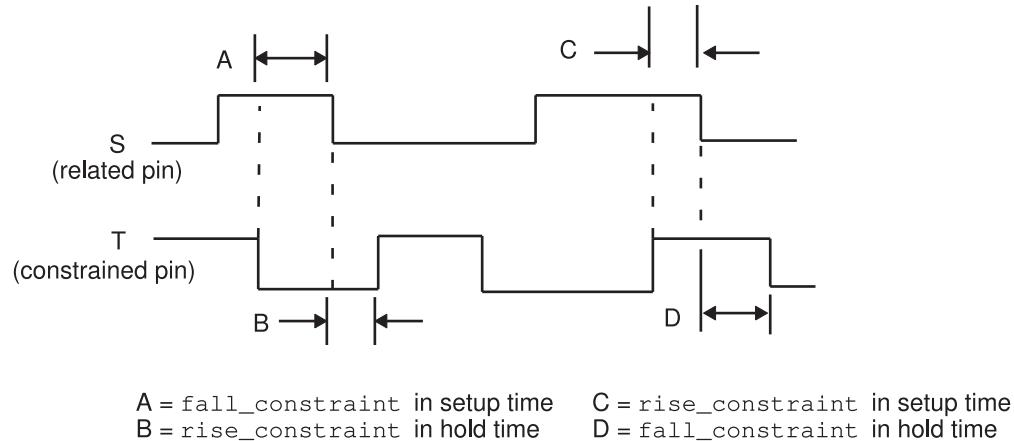
Example

```
pin(T) {  
    timing() {  
        timing_type : non_seq_setup_falling;  
        rise_constraint(scalar) {  
            values (" 1.5 ");  
        }  
        fall_constraint(scalar) {  
            values (" 1.5 ");  
        }  
        related_pin : "S";  
    }  
}
```

[Figure 35](#) 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 35 Nonsequential Setup and Hold Constraints



Setting Recovery and Removal Timing Constraints

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

Recovery Constraints

The recovery timing arc describes the minimum allowable time between the control pin transition to the inactive state and the active edge of the synchronous clock signal (time between the control signal going inactive and the clock edge that latches data in).

The asynchronous control signal must remain constant during this time, or else an incorrect value might appear at the outputs.

Figure 36 Recovery Timing Constraint for a Rising-Edge-Triggered Flip-Flop With Active-Low Clear

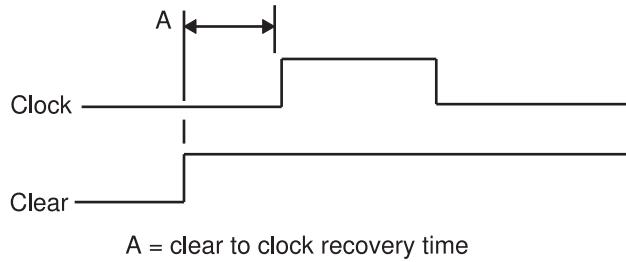
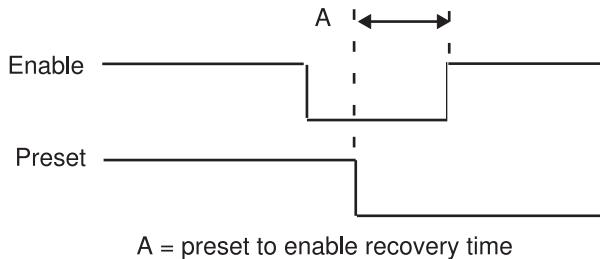


Figure 37 Recovery Timing Constraint for a Low-Enable Latch With Active-High Preset



The values you can assign to a `timing_type` attribute to define a recovery time constraint are

`recovery_rising`

Uses the rising edge of the related pin for the recovery time check; the clock is rising-edge-triggered.

`recovery_falling`

Uses the falling edge of the related pin for the recovery time check; the clock is falling-edge-triggered.

To define a recovery time constraint for an asynchronous control pin,

1. Assign a value to the `timing_type` attribute.

Use `recovery_rising` for rising-edge-triggered flip-flops and low-enable latches; use `recovery_falling` for negative-edge-triggered flip-flops and high-enable latches.

2. Identify the synchronous clock pin as the `related_pin`.

For active-low control signals, define the recovery time with the `rise_constraint` statement.

For active-high control signals, define the recovery time with the `fall_constraint` statement.

Example

This example shows a recovery timing arc for the active-low clear signal in a rising-edge-triggered flip-flop. The `rise_constraint` value represents clock recovery time A in [Figure 36](#).

```
pin (Clear) {
    direction : input ;
    capacitance : 1 ;
    timing() {
        related_pin : "Clock" ;
        timing_type : recovery_rising;
        rise_constraint(scalar) {
            values (" 1.0 " ) ;
        }
    }
}
```

The following example shows a recovery timing arc for the active-high preset signal in a low-enable latch. The `fall_constraint` value represents clock recovery time A in [Figure 37](#).

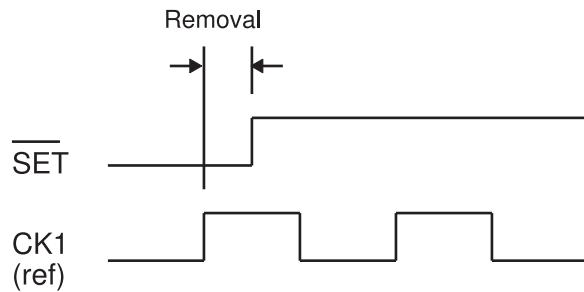
```
pin (Preset) {
    direction : input ;
    capacitance : 1 ;
    timing() {
        related_pin : "Enable" ;
        timing_type : recovery_rising;
        fall_constraint(scalar) {
            values (" 1.0 " ) ;
        }
    }
}
```

Removal Constraint

This constraint is also known as the asynchronous control signal hold time. The removal constraint describes the minimum allowable time between the active edge of the clock

pin while the asynchronous pin is active and the inactive edge of the same asynchronous control pin (see [Figure 38](#)).

Figure 38 Timing Diagram for Removal Constraint



The values you can assign to a `timing_type` attribute to define a removal constraint are

`removal_rising`

Use when the cell is a low-enable latch or a rising-edge-triggered flip-flop.

`removal_falling`

Use when the cell is a high-enable latch or a falling-edge-triggered flip-flop.

To define a removal constraint,

1. Assign a value to the `timing_type` attribute.
2. Identify the synchronous clock pin as the `related_pin`.
3. For active-low asynchronous control signals, define the removal time with the `rise_constraint` attribute.

For active-high asynchronous control signals, define the removal time with the `fall_constraint` attribute.

Example

```
pin ( SET ) {  
    ...  
    timing() {  
        timing_type : removal_rising;  
        related_pin : " CK1 ";  
        rise_constraint(scalar) {  
            values (" 1.0 ");  
        }  
    }  
}
```

Setting No-Change Timing Constraints

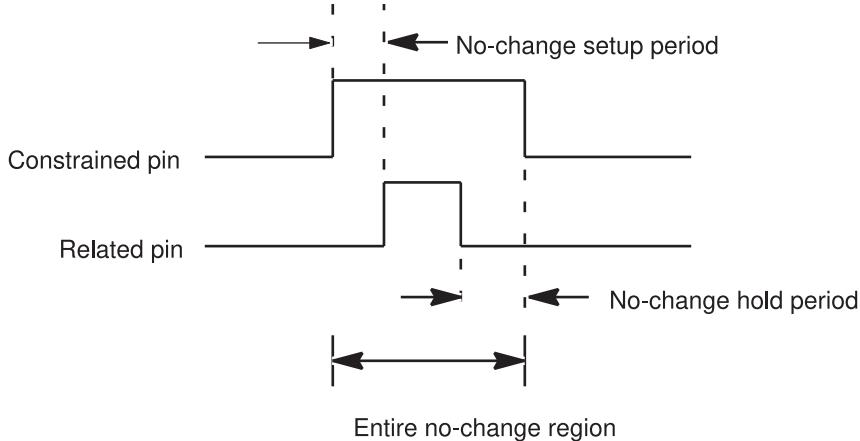
You can model no-change timing checks to use in static timing verification during synthesis.

A no-change timing check checks a constrained signal against a level-sensitive related signal. The constrained signal must remain stable during an established setup period, for the width of the related pulse, and during an established hold period.

For example, you can use the no-change timing check to model the timing requirements of latch devices with latch enable signals. To ensure correct latch sampling, the latch enable signal must remain stable during the clock pulse and the setup and hold time around the clock pulse.

You can also use the no-change timing check to model the timing requirements of memory devices. To guarantee correct read/write operations, the address or data must remain stable during a read/write enable pulse and the setup and hold margins around the pulse.

Figure 39 No-Change Timing Check Between a Constrained Pin and its Level-Sensitive Related Pin



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`

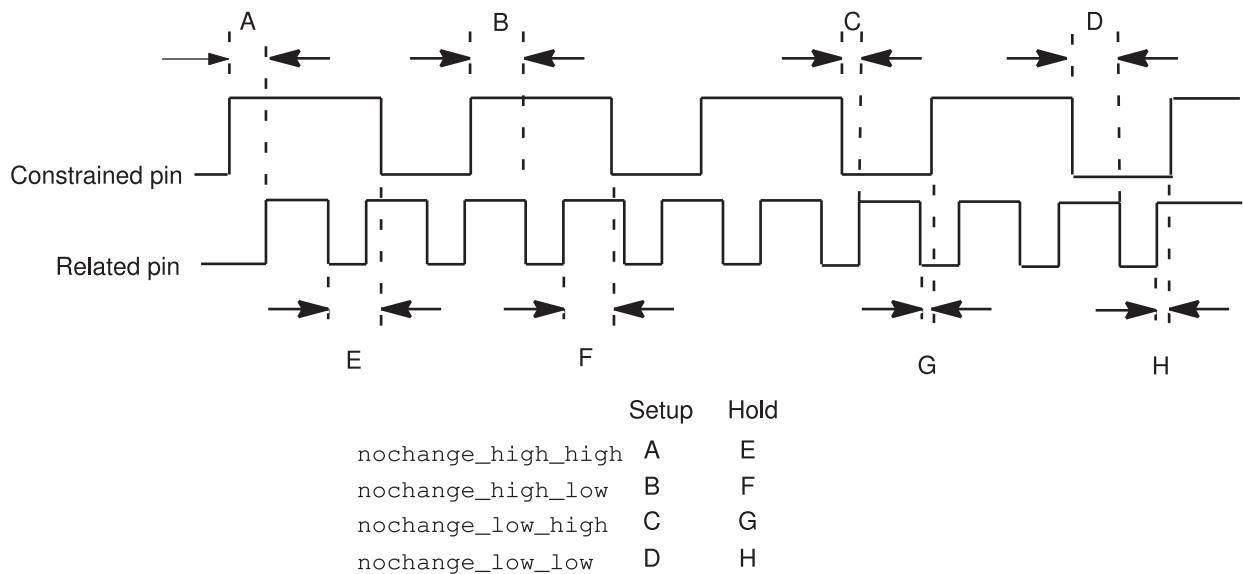
Specifies a negative pulse on the constrained pin and a positive pulse on the related pin.

`nochange_low_low`

Specifies a negative pulse on the constrained pin and a negative pulse on the related pin.

[Figure 40](#) shows the waveforms for these constraints.

Figure 40 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 related pin in the timing arc is the pin used for the timing check.

3. Specify delay attribute values according to the delay model you use, as summarized in the following table.

No-change constraint	Setup attribute for nonlinear delay models	Hold attribute for nonlinear delay models
nochange_high_high	rise_constraint	fall_constraint
nochange_high_low	rise_constraint	fall_constraint
nochange_low_high	fall_constraint	rise_constraint
nochange_low_low	fall_constraint	rise_constraint

Note:

With no-change timing constraints, conditional timing constraints have different interpretations than they do with other constraints. See [Setting Conditional Timing Constraints on page 266](#) for more information.

Specify setup time with the `rise_constraint` attribute and the hold time with the `fall_constraint` attribute.

This is the syntax for the no-change timing check:

```
timing () {
    timing_type : nochange_high_high | nochange_high_low | \
                  nochange_low_high | nochange_low_low ;
    related_pin : related_pinname;
    rise_constraint (template_name_id) {
        /* constrained signal rising */
        values (float, ..., float) ;
    }
    fall_constraint (template_name_id) {
        /* constrained signal falling */
        values (float, ..., float) ;
    }
}
```

Example

This is an example of a no-change timing check in a logic library:

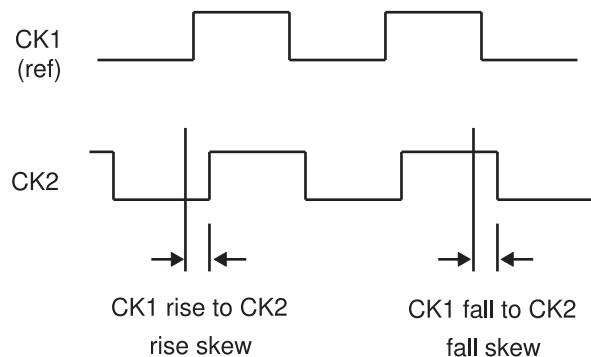
```
library (the_lib) {
    delay_model : table_lookup;
    k_process_nochange_rise : 1.0; /* no-change scaling factors */
    k_process_nochange_fall : 1.0;
    k_volt_nochange_rise : 0.0;
    k_volt_nochange_fall : 0.0;
    k_temp_nochange_rise : 0.0;
    k_temp_nochange_fall : 0.0;
    cell (the_cell) {
        pin(EN) {
            timing () {
                timing_type : nochange_high_low;
```

```
related_pin : CLK;
rise_constraint (temp) { /* setup time */
    values (2.98);
}
fall_constraint (temp) { /* hold time */
    values (0.98);
}
}
...
}
...
}
...
```

Setting Skew Constraints

The skew constraint defines the maximum separation time allowed between two clock signals.

Figure 41 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.

`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.

To set skew constraint,

1. Assign a value to the `timing_type` attribute.
2. Define only one of these attributes in the `timing` group:
 - `rise_constraint`
 - `fall_constraint`
3. Use the `related_pin` attribute in the `timing` group to specify a reference clock pin.
Only the following attributes in a skew timing group are used (all others are ignored):
 - `timing_type`
 - `related_pin`
 - `rise_constraint`
 - `fall_constraint`

Example

This example shows how to model constraint CK1 rise to CK2 rise skew:

```
pin (CK2) {  
    ....  
    timing() {  
        timing_type : skew_rising;  
        related_pin : " CK1 ";  
        rise_constraint(scalar) { /* constrained signal rising */  
            values (" float ") ;  
        }  
    }  
}
```

Setting Conditional Timing Constraints

A conditional timing constraint describes a check when a specified condition is met. You can specify conditional timing checks in `pin`, `bus`, and `bundle` groups.

Use the following attributes and groups to specify conditional timing checks.

Attributes:

- `when`
- `sdf_cond`
- `when_start`
- `sdf_cond_start`

- when_end
- sdf_cond_end
- sdf_edges

Groups:

- min_pulse_width
- minimum_period

when and sdf_cond Simple Attributes

The `when` attribute defines enabling conditions for timing checks such as setup, hold, and recovery.

If you define `when`, you must define `sdf_cond`.

Using the `when` and `sdf_cond` pair is a short way of specifying `when_start`, `sdf_cond_start`, `when_end`, and `sdf_cond_end` when the start condition is identical to the end condition.

[Describing State-Dependent Delays on page 247](#) 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. 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 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. 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 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. 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.

min_pulse_width Example

The following example shows the `min_pulse_width` group with the `constraint_high` and `constraint_low` attributes specified.

Example 78 min_pulse_width Example

```
min_pulse_width() {  
    constraint_high : 3.0; /* min_pulse_width_high */  
    constraint_low : 3.5; /* min_pulse_width_low */  
    when : "SE";  
    sdf_cond : "SE == 1'B1";  
}
```

constraint_high and constraint_low Simple Attributes

At least one of these attributes must be defined in the `min_pulse_width` group. The `constraint_high` attribute defines the minimum length of time the pin must remain at logic 1. The `constraint_low` attribute defines the minimum length of time the pin must remain at logic 0.

when and sdf_cond Simple Attributes

These attributes define the enabling condition for the timing check. Both attributes are required in the `min_pulse_width` group.

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`.

minimum_period Example

```
minimum_period() {  
    constraint : 9.5; /* min_period */  
    when : "SE";  
    sdf_cond : "SE == 1'B1";  
}
```

constraint Simple Attribute

This required attribute defines the minimum clock period for the pin.

when and sdf_cond Simple Attributes

These attributes define the enabling condition for the timing check. Both attributes are required in the `minimum_period` group.

min_pulse_width and minimum_period Example

[Example 79](#) shows how to specify a lookup table with the `timing_type` attribute and `min_pulse_width` and `minimum_period` values. The `rise_constraint` group defines the rising pulse width constraint for `min_pulse_width`, and the `fall_constraint` group defines the falling pulse width constraint. For `minimum_period`, the `rise_constraint` group is used to model the period when the pulse is rising and the `fall_constraint` group is used to model the period when the pulse is falling. You can specify the `rise_constraint` group, the `fall_constraint` group, or both groups.

Example 79 Library With timing_type Statements

```
library(example) {

    technology (cmos) ;
    delay_model : table_lookup ;

    /* 2-D table template */
    lu_table_template ( mpw ) {
        variable_1 : constrained_pin_transition;
        /* You can replace the constrained_pin_transition value with
           related_pin_transition, but you cannot specify both values. */
        variable_2 : related_out_total_output_net_capacitance;
        index_1("1, 2, 3");
        index_2("1, 2, 3");
    }

    /* 1-D table template */
    lu_table_template( f_ocap ) {
        variable_1 : total_output_net_capacitance;
        index_1 (" 0.0000, 1.0000 ");
    }

    cell( test ) {
        area : 200.000000 ;
        dont_use : true ;
        dont_touch : true ;

        pin ( CK ) {
            direction : input;
            rise_capacitance : 0.00146468;
        }
    }
}
```

Chapter 7: Timing Arcs
Setting Conditional Timing Constraints

```
fall_capacitance : 0.00145175;
capacitance : 0.00146468;
clock : true;

timing ( mpw_constraint) {
    related_pin : "CK";
    timing_type : min_pulse_width;
    related_output_pin : "Z";

    fall_constraint ( mpw) {
        index_1("0.1, 0.2, 0.3");
        index_2("0.1, 0.2");
        values( "0.10 0.11", \
                "0.12 0.13" \
                "0.14 0.15");
    }

    rise_constraint ( mpw) {
        index_1("0.1, 0.2, 0.3");
        index_2("0.1, 0.2");
        values( "0.10 0.11", \
                "0.12 0.13" \
                "0.14 0.15");
    }

    timing ( mpw_constraint) {
        related_pin : "CK";
        timing_type : minimum_period;
        related_output_pin : "Z";

        fall_constraint ( mpw) {
            index_1("0.2, 0.4, 0.6");
            index_2("0.2, 0.4");
            values( "0.20 0.22", \
                    "0.24 0.26" \
                    "0.28 0.30");
        }

        rise_constraint ( mpw) {
            index_1("0.2, 0.4, 0.6");
            index_2("0.2, 0.4");
            values( "0.20 0.22", \
                    "0.24 0.26" \
                    "0.28 0.30");
        }
    }
}

. . .
} /* end of arc */
} /* end of cell */
} /* end of library */
```

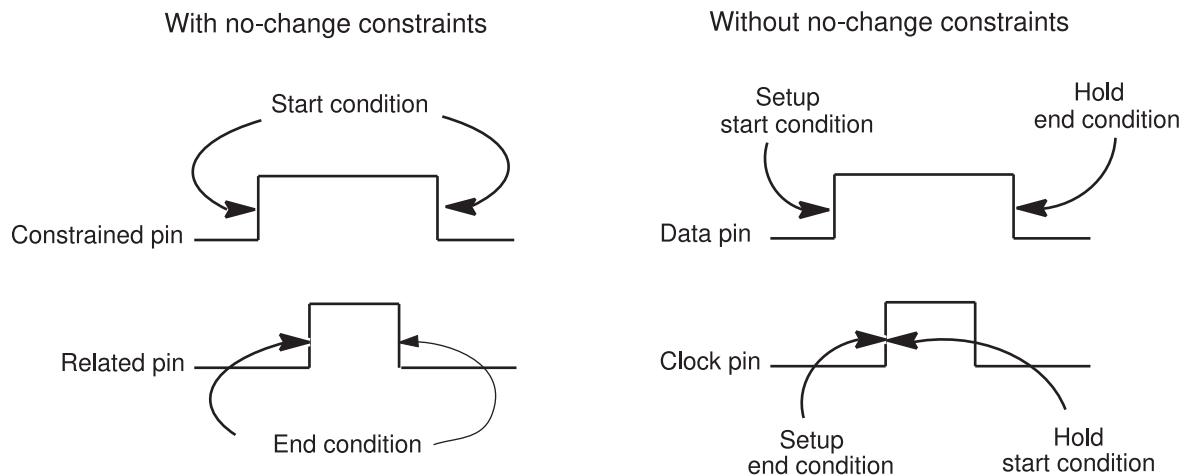
Using Conditional Attributes With No-Change Constraints

As shown in [Table 20](#), conditional timing check attributes have different interpretations when you use them with no-change timing constraints. See [Setting No-Change Timing Constraints on page 262](#) for a description of no-change timing constraint values.

Table 20 Conditional Timing Attributes With No-Change Constraints

Conditional attributes	With no-change constraints
when sdf_cond	Defines both the constrained and related signal-enabling conditions
when_start sdf_cond_start	Defines the constrained signal-enabling condition
when_end sdf_cond_end	Defines the related signal-enabling condition
sdf_edges : start_edge	Adds edge specification to the constrained signal
sdf_edges : end_edge	Adds edge specification to the related signal
sdf_edges : both_edges	Specifies edges to both signals
sdf_edges : noedges	Specifies edges to neither signal

Figure 42 Interpretation of Conditional Timing Constraints With No-Change Constraints



Impossible Transitions

This information applies to the table lookup delay model and only to combinational and three-state timing arcs. Certain output transitions cannot result from a single input change when function, three_state, and x_function share input.

In the following table, Y is the function of A, B, and C.

A	B	C	Y
0	0	0	Z
0	0	1	1
0	1	0	0
0	1	1	X
1	0	0	0
1	0	1	1
1	1	0	Z
1	1	1	X

No isolated signal change on C can cause the 0-to-1 or 1-to-0 transitions on Y. Therefore, there is no combinational arc from C to Y, although the two are functionally related. Further, no isolated change on A causes the 1-to-Z or Z-to-1 transitions on Y.

three_state_enable has no rising value (Z to 1), and three_state_disable has no falling value (1 to Z).

Examples of NLDM Libraries

This section contains examples of libraries using the CMOS nonlinear delay model.

CMOS Piecewise Linear Delay Model

In the CMOS piecewise linear D flip-flop description in [Example 80](#), pin D contains setup and hold constraints; pins Q and QN contain preset, clear, and edge-sensitive arcs.

Example 80 D Flip-Flop Description (CMOS Piecewise Linear Delay Model)

```
library (example) {
    date : "January 14, 2002";
```

Chapter 7: Timing Arcs
Examples of NLDM Libraries

```
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 */
        }
    }
}
```

Chapter 7: Timing Arcs
Examples of NLDM Libraries

```
    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 */
}
}
```

Library With Timing Constraints

In the nonlinear library description in [Example 81](#), pin D contains setup and hold constraints; pins Q and QN contain preset, clear, and edge-sensitive arcs.

Example 81 D Flip-Flop Description

```
library (NLDM) {
date : "January 14, 2015";
revision : 2015.01;
delay_model : table_lookup;
technology (cmos);
/* Define template of size 2 x 2*/
lu_table_template(cell_template) {
    variable_1 : input_net_transition;
    variable_2 : total_output_net_capacitance;
    index_1 ("0.0, 1.5");
    index_2 ("0.0, 4.0");
}
/* Define one-dimensional lu_table of size 4 */
lu_table_template(tran_template) {
    variable_1 : total_output_net_capacitance;
    index_1 ("0.0, 0.5, 1.5, 2.0");
}
cell( DFLOP_CLR_PRE ) {
    area : 11 ;
    ff ( IQ , IQN ) {
        clocked_on : " CLK ";
        next_state : " D ";
        clear : " CLR' " ;
        preset : " PRE' " ;
        clear_preset_var1 : L ;
        clear_preset_var2 : L ;
    }
}
```

Chapter 7: Timing Arcs
Examples of NLDM Libraries

```
}

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 () {
```

Chapter 7: Timing Arcs

Examples of NLDM Libraries

```
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");
}
}
}
}
```

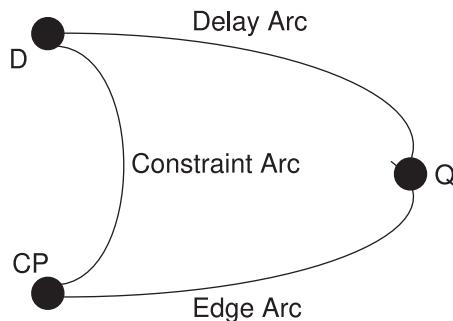
Library With Clock Insertion Delay

```
library( vendor_a ) {
delay_model :table_lookup;
/* 1. Define library-level one-dimensional lu_table of size 4 */
lu_table_template(lu_template) {
variable_1 :input_net_transition;
index_1 ("0.0, 0.5, 1.5, 2.0");
}
/* 2. Define a cell and pins within it which has clock tree path*/
cell (general) {
...
pin(clk) {
direction:input;
timing() {
timing_type :max_clock_tree_path;
timing_sense :positive_unate;
cell_rise(lu_template) {
values ("0.1, 0.15, 0.20, 0.29");
}
cell_fall(lu_template) {
values ("0.2, 0.25, 0.30, 0.39");
}
}
timing() {
timing_type :min_clock_tree_path;
timing_sense :positive_unate;
cell_rise(lu_template) {
values ("0.2, 0.35, 0.40, 0.59");
}
cell_fall(lu_template) {
values ("0.3, 0.45, 0.50, 0.69");
}
}
}
}
...
}
```

Describing a Transparent Latch Clock Model

The `tLatch` group lets you specify a functional latch when the latch arcs are absent. You use the `tLatch` group at the data pin level to specify the relationship between the data pin and the enable pin on a latch.

Figure 43 *Transparent Latch Timing Model*



Syntax

```
library (namestring) {
    cell (namestring) {
        ...
        timing_model_type : valueenum ;
        ...
        pin (data_pin_namestring) {
            tLatch (enable_pin_namestring) {
                edge_type : valueenum ;
                tDisable : valueBoolean ;
            }
        }
    }
}
```

The `tLatch` name specifies the enable pin that defines the latch clock pin. You define the `tLatch` group in a `pin` group, but it is only effective if you also define the `timing_model_type` attribute in the cell that the pin belongs to. The `timing_model_type` attribute can have the following values: abstracted, extracted, and quick timing model. A `tLatch` group is optional. You can define one or more `tLatch` groups for a pin, but you must not define more than two `tLatch` groups between the same pair of data and enable pins, one rising and one falling. Also, the data pin and the enable pin must be different.

Pins in the `tlatch` group can be input or inout pins or internal pins. When a `tlatch` group is not present, a timing analysis tool infers the latch clock pin based on the presence of:

- A `related_pin` statement in a `timing` group with either a `rising_edge` or `falling_edge` `timing_type` value within a latch output pin
- A `related_pin` statement in a `timing` group with a `setup`, `hold_rising`, or `falling` `timing_type` value within a latch input pin

The `edge_type` attribute defines whether the latch is positive (high) transparent or negative (low) transparent. The rising and falling `edge_type` values specify the opening edge, and therefore the transparent window of the latch, and completely define the latch to be level-high transparent or level-low transparent.

When a `tlatch` group is not present, a timing analysis tool infers transparency on an output pin based on the timing arc attribute and the presence of a latch functional construct on that pin.

The rising and falling `edge_type` attribute values explicitly define the transparency windows

- When the `rising_edge` and `falling_edge` `timing_type` values are missing
- When the `rising_edge` and `falling_edge` `timing_type` values are different from the latch transparency

The `tdisable` attribute disables transparency in a latch. During path propagation, timing analysis tools disable and ignore all data pin output pin arcs that reference a `tlatch` group whose `tdisable` attribute is set to true on an edge triggered flip-flop.

Example

```
pin (D) {
    tlatch (CP) {
        edge_type : rising ;
        tdisable : true ;
    }
}
pin (Q) {
    direction : output ;
    timing () {
        timing_)sense : positive_unate ;
        related_pin : D ;
        cell_rise ...
    }
    timing () {
        /* optional arc that can differ from edge_type */
        timing_type : falling_edge ;
        related_pin : CP ;
        cell_fall ...
    }
}
```

}

Driver Waveform Support

In cell characterization, the shape of the waveform driving the characterized circuit can have a significant impact on the final results. Typically, the waveform is generated by a simple piecewise linear (PWL) waveform or an active-driver cell (a buffer or inverter).

Liberty supports driver waveform syntax, which specifies the type of waveform that is applied to library cells during characterization. The driver waveform syntax helps facilitate the characterization process for existing libraries and correlation checking. The driver waveform requirements can vary.

Possible usage models include:

- Using a common driver waveform for all cells.
- Using a different driver waveform for different categories of cells.
- Using a pin-specific driver waveform for complex cell pins.
- Using a different driver waveform for rise and fall timing arcs.

Syntax

The driver waveform syntax is as follows:

```
library(library_name) {  
    ...  
    lu_table template (waveform_template_name) {  
        variable_1: input_net_transition;  
        variable_2: normalized_voltage;  
        index_1 ("float..., float");  
        index_2 ("float..., float");  
    }  
    normalized_driver_waveform(waveform_template_name) {  
        driver_waveform_name : string; /* Specifies the name of the driver  
                                         waveform table */  
        index_1 ("float..., float"); /* Specifies input net transition */  
        index_2 ("float..., float"); /* Specifies normalized voltage */  
        values ("float..., float", \ /* Specifies the time in library units */  
                ...,  
                "float..., float");  
    }  
    ...  
    cell (cell_name) {  
        ...  
        driver_waveform : string;  
        driver_waveform_rise : string;  
        driver_waveform_fall : string;  
        pin (pin_name) {  
            driver_waveform : string;  
            driver_waveform_rise : string;  
            driver_waveform_fall : string;  
        }  
        ...  
    }  
}
```

```
        }
    } /* end of library*/
```

In the driver waveform syntax, the first index value in the table specifies the input slew and the second index value specifies the voltage normalized to VDD. The values in the table specify the time in library units (not scaled) when the waveform crosses the corresponding voltages. The `driver_waveform_name` attribute specified for the driver waveform table differentiates the tables when multiple driver waveform tables are defined.

The cell-level `driver_waveform`, `driver_waveform_rise` and `driver_waveform_fall` attributes meet cell-specific and rise- and fall-specific requirements. The attributes refer to the driver waveform table name predefined at the library level.

Similar to the cell-level driver waveform attributes, the pin group includes the `driver_waveform`, `driver_waveform_rise` and `driver_waveform_fall` attributes to meet pin-specific predriver requirements for complex cell pins (such as macro cells). These attributes also refer to the predefined driver waveform table name.

Library-Level Tables, Attributes, and Variables

This section describes driver waveform tables, attributes, and variables that are specified at the library level.

normalized_voltage Variable

The `normalized_voltage` variable is specified under the `lu_table_template` table to describe a collection of waveforms with various input slew values. For a given input slew in `index_1` (for example, `index_1[0] = 1.0 ns`), the `index_2` values are a set of points that represent how the voltage rises from 0 to VDD in a rise arc, or from VDD to 0 in a fall arc.

Rise Arc Example

```
normalized_driver_waveform (waveform_template) {
    index_1 ("1.0"); /* Specifies the input net transition*/
    index_2 ("0, 0.1, 0.3, 0.5, 0.7, 0.9, 1.0"); /* Specifies
    the voltage normalized to VDD */
    values ("0, 0.2, 0.4, 0.6, 0.8, 0.9, 1.1"); /* Specifies the
    time when the voltage reaches the index_2 values*/
}
```

The `lu_table_template` table represents an input slew of 1.0 ns, when the voltage is 0%, 10%, 30%, 50%, 70%, 90% or 100% of VDD, and the time values are 0, 0.2, 0.4, 0.6, 0.8, 0.9, 1.1 (ns). The time value can go beyond the corresponding input slew because a long tail might exist in the waveform before it reaches the final status.

normalized_driver_waveform Group

The library-level `normalized_driver_waveform` group represents a collection of driver waveforms under various input slew values. The `index_1` specifies the input slew and `index_2` specifies the normalized voltage.

The slew index in the `normalized_driver_waveform` table is based on the slew derate and slew trip points of the library (global values). When applied on a pin or cell with different slew or slew derate, the new slew should be interpreted from the waveform.

driver_waveform_name Attribute

The `driver_waveform_name` string attribute differentiates the driver waveform table from other driver waveform tables when multiple tables are defined. Cell-specific, rise-specific, and fall-specific driver waveform usage modeling depend on this attribute.

The `driver_waveform_name` attribute is optional. You can define a driver waveform table without the attribute, but there can be only one table in a library, and that table is regarded as the default driver waveform table for all cells in the library. If more than one table is defined without the attribute, the last table is used. The other tables are ignored and not stored in the library database.

Cell-Level Attributes

This section describes driver waveform attributes defined at the cell level.

driver_waveform Attribute

The `driver_waveform` attribute is an optional string attribute that allows you to define a cell-specific driver waveform. The value must be the `driver_waveform_name` predefined in the `normalized_driver_waveform` table.

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 284](#).

`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 284](#).

Driver Waveform Example

```
library(test_library) {  
  
    lu_table_template(waveform_template) {  
        variable_1 : input_net_transition;  
        variable_2 : normalized_voltage;  
        index_1 ("0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7");  
        index_2 ("0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9");  
    }  
  
    /* Specifies the default library-level driver waveform table (the  
       default driver waveform without the driver_waveform attribute) */  
    normalized_driver_waveform (waveform_template) {  
        values ("0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09", \  
               "0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19", \  
               ... ... ...  
               "0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9");  
    }  
}
```

Chapter 7: Timing Arcs

Driver Waveform Support

```
}

/* Specifies the driver waveform for the clock pin */
normalized_driver_waveform (waveform_template) {
    driver_waveform_name : clock_driver;
    index_1 ("0.15, 0.25, 0.35, 0.45, 0.55, 0.65, 0.75");
    index_2 ("0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9");
    values ("0.012, 0.03, 0.045, 0.06, 0.075, 0.090, 0.105, 0.13,
0.145", \
            ...
            "0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9");
}

/* Specifies the driver waveform for the bus */
normalized_driver_waveform (waveform_template) {
    driver_waveform_name : bus_driver;
    index_1 ("0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7");
    index_2 ("0.15, 0.25, 0.35, 0.45, 0.55, 0.65, 0.75, 0.85, 0.95");
    values ("0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09",
            \
            "0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19",
            ...
            "0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9");
}

/* Specifies the driver waveform for the rise */
normalized_driver_waveform (waveform_template) {
    driver_waveform_name : rise_driver;
    index_1 ("0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7");
    index_2 ("0.15, 0.25, 0.35, 0.45, 0.55, 0.65, 0.75, 0.85, 0.95");
    values ("0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09",
            \
            "0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19",
            ...
            "0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9");
}

/* Specifies the driver waveform for the fall */
normalized_driver_waveform (waveform_template) {
    driver_waveform_name : fall_driver;
    index_1 ("0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7");
    index_2 ("0.15, 0.25, 0.35, 0.45, 0.55, 0.65, 0.75, 0.85, 0.95");
    values ("0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09",
            \
            "0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19",
            ...
            "0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9");
}

...

cell (my_cell1) {
    driver_waveform : clock_driver;
    ...
}
cell (my_cell2) {
    driver_waveform : bus_driver;
    ...
}
cell (my_cell3) {
    driver_waveform_rise : rise_driver;
    driver_waveform_fall : fall_driver;
```

```
    ...
}
cell (my_cell4) {
/* No driver_waveform attribute is specified. Use the default driver
   waveform */
...
}
/* End library */
```

Sensitization Support

Timing information specified in libraries results from circuit simulator (library characterization) tools. The models themselves often represent only a partial record of how a particular arc was sensitized during characterization. To fully reproduce the conditions that allowed the model to be generated, the states of all the input pins on the cell must be known.

In composite current source (CCS) models, the accuracy requirements are very high (the expectation is as high as 2 percent compared to SPICE). To achieve this level of accuracy, correlation with SPICE requires that the cell conditions be represented exactly as during characterization. For more information about CCS models, see [Chapter 10, Composite Current Source Modeling.](#)

The following sections describe the pin sensitization condition information details used to characterize the timing data. The syntax predeclares state vectors as reusable sensitization patterns in the library. The patterns are referenced and instantiated as stimuli waveforms specific to timing arcs. The same sensitization pattern can be referenced by multiple cells or multiple timing arcs. One cell can also reference multiple sensitization patterns, which saves storage resources. The Liberty attributes and groups highlighted in the following sections help to specify the sensitization information of timing arcs during simulation and characterization.

sensitization Group

The `sensitization group` defined at the library level describes the complete state patterns for a specific list of pins (defined by the `pin_names` attribute) that is referenced and instantiated as stimuli in the timing arc.

Vector attributes in the group define all possible pin states used as stimuli. Actual stimulus waveforms can be described by a combination of these vectors. Multiple `sensitization group` 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 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. Waveform
               interpretation of wave_rise is (for "A, B, Z" pins): 10 1 01 */
                wave_fall (3, 6);
            ...
            }
            timing() {
                related_pin : "B";
                wave_rise (7, 4); /* 7, 4 - vector id in sensi_2in_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).

Example

```
wave_rise (2, 5, 7, 6); /* wave_rise ( wave_rise[0],  
wave_rise[1], wave_rise[2], wave_rise[3] ); */
```

In the previous example, the wave rise vector delay is measured from the last transition (vector 7 changing to vector 6) to the output transition. The default `wave_rise_sampling_index` value is the last entry in the vector, which is 3 in this case (because the numbering begins at 0).

To override this default, set the `wave_rise_sampling_index` attribute, as shown:

```
wave_rise_sampling_index : 2 ;
```

When you specify this attribute, the delay is measured from the second last transition of the sensitization vector to the final output transition, in other words from the transition of vector 5 to vector 7.

Note:

You cannot specify a value of 0 for the `wave_rise_sampling_index` attribute.

wave_rise_time_interval and wave_fall_time_interval Attributes

The `wave_rise_time_interval` and `wave_fall_time_interval` attributes control the time interval between transitions. By default, the stimuli (specified in `wave_rise` and `wave_fall`) are widely spaced apart during characterization (for example, 10 ns from one vector to the next) to allow all output transitions to stabilize. The attributes allow you to specify a short duration between one vector to the next to characterize special purpose cells and pessimistic timing characterization.

The `wave_rise_time_interval` and `wave_fall_time_interval` attributes are optional when the default time interval is used for all transitions, and they are required when you

need to define special time intervals between transitions. Usually, the special time interval is smaller than the default time interval.

The `wave_rise_time_interval` and `wave_fall_time_interval` attributes can have an argument count from 1 to $n-1$, where n is the number of arguments in corresponding `wave_rise` or `wave_fall`. Use 0 to imply the default time interval used between vectors.

Example

```
wave_rise (2, 5, 7, 6); /* wave_rise ( wave_rise[0],  
wave_rise[1], wave_rise[2], wave_rise[3] ); */  
wave_rise_time_interval (0.0, 0.3);
```

The previous example suggests the following:

- Use the default time interval between `wave_rise[0]` and `wave_rise[1]` (in other words, vector 2 and vector 5).
- Use 0.3 between `wave_rise[1]` and `wave_rise[2]` (in other words, vector 5 and vector 7).
- Use the default time interval between `wave_rise[2]` and `wave_rise[3]` in other words, vector 7 and vector 6).

timing Group Syntax

```
library(library_name) {  
    ...  
    sensitization (sensitization_group_name) {  
        pin_names (string..., string);  
        vector (integer, string);  
        ...  
        vector (integer, string);  
    }  
    ...  
  
    cell(cell_name) {  
        sensitization_master : sensitization_group_name;  
        pin_name_map (string..., string);  
        ...  
        pin(pin_name) {  
            ...  
            timing() {  
                related_pin : string;  
                sensitization_master : sensitization_group_name;  
                pin_name_map (string,..., string);  
                wave_rise (integer,..., integer);  
                wave_fall (integer,..., integer);  
                wave_rise_sampling_index : integer;  
                wave_fall_sampling_index : integer;
```

```
wave_rise_timing_interval (float..., float);
wave_fall_timing_interval (float..., float);
...
}/*end of timing */
}/*end of pin */
}/*end of cell */
...
}/* end of library*/
```

NAND Cell Example

The following is an example of a NAND cell with sensitization information.

```
library(cell1) {
    sensitization(sensitization_nand2) {
        pin_names ( IN1, IN2, OUT1 );
        vector ( 1, "0 0 1" );
        vector ( 2, "0 1 1" );
        vector ( 3, "1 0 1" );
        vector ( 4, "1 1 0" );
    }

    cell (nand2) {
        sensitization_master : sensitization_nand2;
        pin_name_map (A, B, Y);
        pin (A) {
            direction : input ;
            ...
        }
        pin (B) {
            direction : input ;
            ...
        }
        pin (Y) {
            direction : output;
            ...
            timing() {
                related_pin : "A";
                timing_sense : negative_unate;
                wave_rise ( 4, 2 ); /* 10 1 01 */
                wave_fall ( 2, 4 ); /* 01 1 10 */
                ...
            }
            timing() {
                related_pin : "B";
                timing_sense : negative_unate;
                wave_rise ( 4, 3 );
                wave_fall ( 3, 4 );
                ...
            }
        } /* end pin(Y) */
    } /* end cell(nand2) */
}
```

```
 } /* end library */
```

Complex Macro Cell Example

The following is an example of a complex macro cell highlighting the usage of the sensitization_master group inside the timing group.

```
library(cell1) {
    sensitization(sensitization_2in_1out) {
        pin_names ( IN1, IN2, OUT );
        vector ( 1, "0 0 1" );
        vector ( 2, "0 1 1" );
        vector ( 3, "1 0 1" );
        vector ( 4, "1 1 0" );
    }
    sensitization(sensitization_3in_1out) {
        pin_names ( IN1, IN2, IN3, OUT );
        vector ( 0, "0 0 0 0" );
        vector ( 1, "0 0 0 1" );
        vector ( 2, "0 0 1 0" );
        vector ( 3, "0 0 1 1" );
        vector ( 4, "0 1 0 0" );
        vector ( 5, "0 1 0 1" );
        vector ( 6, "0 1 1 0" );
        vector ( 7, "0 1 1 1" );
        vector ( 8, "1 0 0 0" );
        vector ( 9, "1 0 0 1" );
        vector ( 10, "1 0 1 0" );
        vector ( 11, "1 0 1 1" );
        vector ( 12, "1 1 0 0" );
        vector ( 13, "1 1 0 1" );
        vector ( 14, "1 1 1 0" );
        vector ( 15, "1 1 1 1" );
    }
    cell (nand2) {
        sensitization master : sensitization_2in_1out;
        pin_name_map (A, B, Y);
        pin (A) {
            direction : input ;
            ...
        }
        pin (B) {
            direction : input ;
            ...
        }
        pin (CIN0) {
            direction : input ;
            ...
        }
        pin (CIN1) {
            direction : input ;
            ...
        }
    }
}
```

Chapter 7: Timing Arcs Sensitization Support

```
    }
    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_lout; /* 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_lout group. */

            wave_fall (15, 9, 3, 1, 6, 7);
            wave_rise_sampling_index : 3; /* sampling index, specific for
this timing arc. */

            wave_fall_sampling_index : 3;
            wave_rise_timing_interval(0, 0.3, 0.3); /* special timing
interval, specific for this timing arc. */

            wave_fall_timing_interval(0, 0.3, 0.3);
        }
    } /* end pin (Z) */
} /* end cell(nand2) */
} /* end library */
```

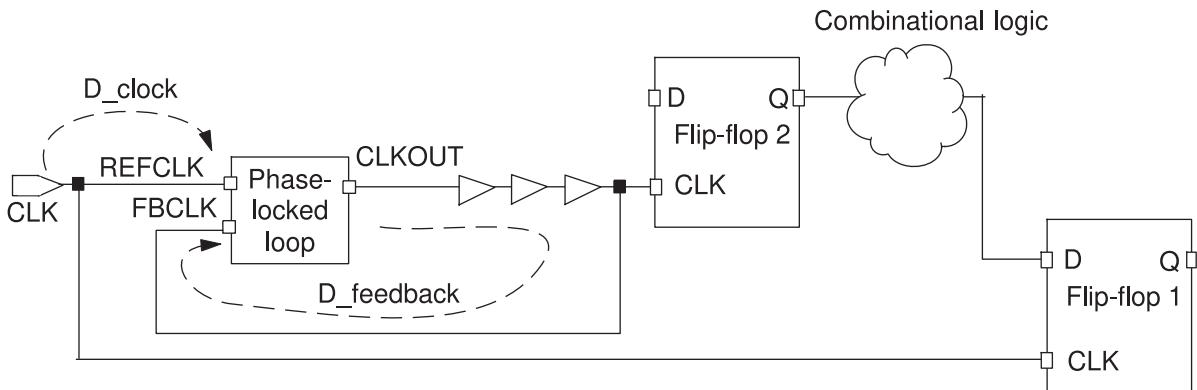
Phase-Locked Loop Support

A phase-locked loop (PLL) is a feedback control system that automatically adjusts the phase of a locally-generated signal to match the phase of an input signal. Phase-locked loops contain the following pins:

- The reference clock pin where the reference clock is connected
- The phase-locked loop output clock pin where the phase-locked loop generates a phase-shifted version of the reference clock
- The feedback pin where the feedback path from the output of the clock ends

A phase-locked loop reduces the large skew between the clocks arriving at the launch point and at the flip-flops. The large skew results in an extremely tight delay constraint for the combinational logic.

Figure 44 Phase-Locked Loop Circuit



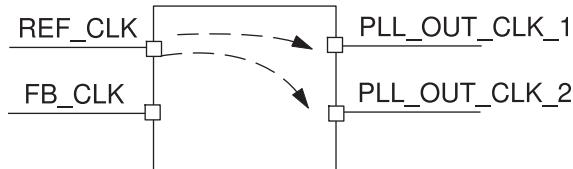
The phase-locked loop generates a phase-shifted version of the reference clock at the output pin so that the phase of the feedback clock matches the phase of the reference clock.

In Figure 44, if the clock arrives at the reference pin of the phase-locked loop at the time, D_{clock} , and the delay of the feedback path is $D_{feedback}$, the source latency of the clock generated at the output of the phase-locked loop is $D_{clock} - D_{feedback}$. The clock arrives at the feedback pin at time, D_{clock} , which is the same as the time the clock arrives at the reference pin. Therefore, the phase-locked loop eliminates the latency on the launch path and relaxes the delay constraint on the combinational logic.

The phase-locked loop information that a library must contain are pins and timing arcs inside the phase-locked loop. In Figure 45, the forward arcs from `REF_CLK` to `PLL_OUT_CLK_*` in the figure simulate the phase shift behavior of the phase-locked loop.

There are two half-unate arcs from the reference clock to each output of the phase-locked loop.

Figure 45 Simple Phase-Locked Loop Model



Phase-Locked Loop Syntax

```
cell (cell_name) {
    is_pll_cell : true;
    pin (ref_pin_name) {
        is_pll_reference_pin : true;
        direction : output;
        ...
    }
    pin (feedback_pin_name) {
        is_pll_feedback_pin : true;
        direction : output;
        ...
    }
    pin (output_pin_name) {
        is_pll_output_pin : true;
        direction : output;
        ...
    }
}
```

Cell-Level Attribute

This section describes a cell-level attribute.

is_pll_cell Attribute

The `is_pll_cell` Boolean attribute identifies a phase-locked loop cell.

Pin-Level Attributes

This section describes pin-level attributes.

is_pll_reference_pin Attribute

The `is_pll_reference_pin` Boolean attribute tags a pin as a reference pin on the phase-locked loop. In a phase-locked loop cell group, the `is_pll_reference_pin` attribute should be set to true in only one input pin group.

is_pll_feedback_pin Attribute

The `is_pll_feedback_pin` Boolean attribute tags a pin as a feedback pin on a phase-locked loop. In a phase-locked loop cell group, the `is_pll_feedback_pin` attribute should be set to true in only one input pin group.

is_pll_output_pin Attribute

The `is_pll_output_pin` Boolean attribute tags a pin as an output pin on a phase-locked loop. In a phase-locked loop cell group, the `is_pll_output_pin` attribute should be set to true in one or more output pin groups.

Phase-Locked Loop Example

```
cell(my_pll) {
    is_pll_cell : true;

    pin( REFCLK ) {
        direction : input;
        is_pll_reference_pin : true;
    }

    pin( FBKCLK ) {
        direction : input;
        is_pll_feedback_pin : true;
    }

    pin (OUTCLK_1x) {
        direction : output;
        is_pll_output_pin : true;
        timing() /* Timing Arc */
            related_pin: "REFCLK";
            timing_type: combinational_rise;
            timing_sense: positive_unate;
        . . .
    }
    timing() /* Timing Arc */
        related_pin: "REFCLK";
        timing_type: combinational_fall;
        timing_sense: positive_unate;
    . . .
}
```

Chapter 7: Timing Arcs
Phase-Locked Loop Support

```
pin (OUTCLK_2x) {
    direction : output;
    is_pll_output_pin : true;
    timing() /* Timing Arc */
        related_pin: "REFCLK";
        timing_type: combinational_rise;
        timing_sense: positive_unate;
    . . .
}
timing() /* Timing Arc */
    related_pin: "REFCLK";
    timing_type: combinational_fall;
    timing_sense: positive_unate;
. . .
}
/* End pin group */
} /* End cell group */
```

8

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 for Leakage Power](#)
- [Representing Leakage Power Information](#)
- [Threshold Voltage Modeling](#)
- [Modeling for Internal and Switching Power](#)
- [Representing Internal Power Information](#)
- [Defining Internal Power Groups](#)
- [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

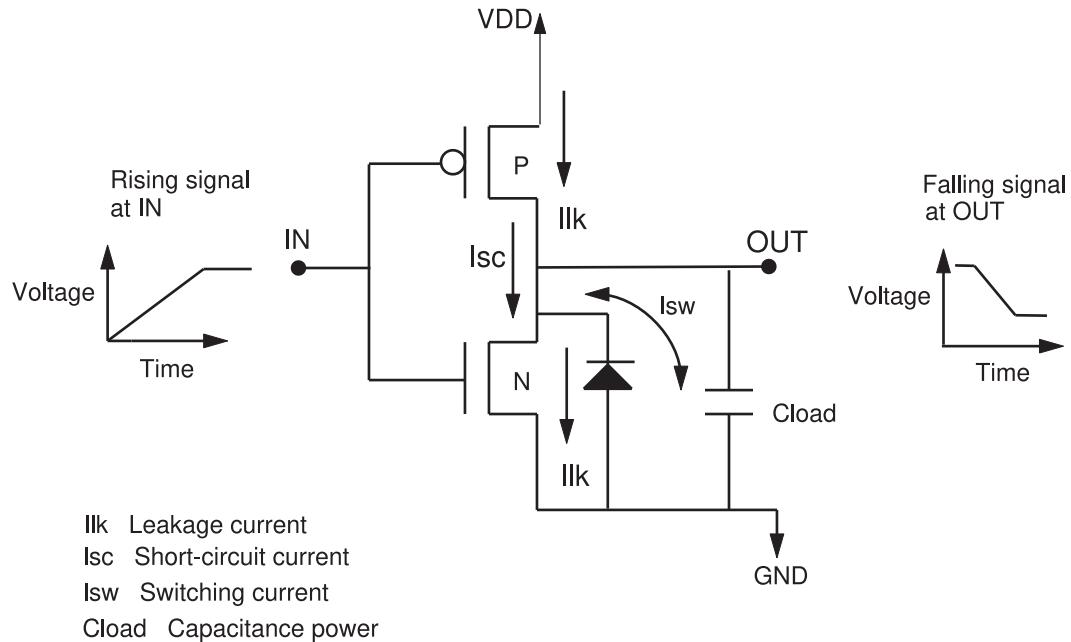
- Internal power
- Switching power

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 46](#) 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 46 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.

$$(1/2)CV^2 \cdot TR_i$$

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 46](#) 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.

For more information about switching activity and power consumption, see the *Power Compiler User Guide*.

Modeling for Leakage Power

Regardless of the physical reasons for leakage power, library developers can annotate gates with the approximate total leakage power dissipated by the gate.

Leakage power characterization requires measuring the supply current (I_{supply}) of the quiescent state of the cell. This requires a DC analysis of the circuit with steady-state voltages at the inputs to the cell.

Vendors can characterize leakage power of multiple combinations of input states to generate state-dependent leakage power models. This is especially important when leakage power dissipation varies greatly from one state to another and requires iterating

across all possible inputs and measuring the supply current for each vector. For each vector or state, leakage power can be computed as

$$P_{\text{leakage}} = V_{\text{DD}} \times I_{\text{supply}}$$

Alternatively, leakage power can also be characterized with two simulation runs: one for output high and one for output low. The average of these two measurements is then used as the cell's leakage power. This method reduces the characterization effort at the expense of accuracy.

Representing Leakage Power Information

You can represent leakage power information in the library as:

- Cell-level state-independent leakage power with the `cell_leakage_power` attribute
- Cell-level state-dependent leakage power with the `leakage_power` group
- The associated library-level attributes that specify scaling factors, units, and a default for both leakage and power density

cell_leakage_power Simple Attribute

This attribute specifies the leakage power of a cell. If this attribute is missing or negative, the value of the `default_cell_leakage_power` attribute is used.

Syntax

```
cell_leakage_power : value ;
```

Note:

You must define this attribute for cells with state-dependent leakage power to provide the leakage power value for those states where the state-specific leakage power has not been specified using the `leakage_power` group.

Using the leakage_power Group for a Single Value

This group specifies the leakage power of a cell when the leakage power depends on the logical condition of that cell. This type of leakage power is called state-dependent. To model state-dependent leakage power, use the following attributes:

- `when`
- `value`

Syntax

```
leakage_power ( ) {  
    mode (mode_name, mode_value);  
    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 21 Valid Boolean Operators

Operator	Description
,	invert previous expression
!	invert following expression
^	logical XOR
*	logical AND
&	logical AND
space	logical AND
+	logical OR
	logical OR
1	signal tied to logic 1
0	signal tied to logic 0

The order of precedence of the operators is left to right, with inversion performed first, then XOR, then AND, then OR.

You must define mutually exclusive conditions for state-dependent leakage power and internal power. Mutually exclusive means that only one condition defined in the `when` attribute can be met at any given time.

You cannot directly reference a bus or a bundle in the `when` attribute in the `leakage_power` group. However, you can reference the pins of buses and bundles in the `when` attribute in the `leakage_power` group, as shown here:

Example

```
cell(Z) {  
    ...  
    cell_leakage_power : 3.0 ;  
    leakage_power () {  
        when : "!A[0] * !A[1] * !A[2] * !A[3]" ;  
        value : 2.0 ;  
    }  
    bus(A) {  
        bus_type : bus4 ;  
        direction : input ;  
        fanout_load : 1.00 ;  
        capacitance : 0.15  
        ...  
    }  
}
```

value Simple Attribute

The `value` attribute represents the leakage power for a given state of a cell. The value for this attribute is a floating-point number.

Example

```
cell (my cell) {  
    ...  
    leakage_power () {  
        when : "! A";  
        value : 2.0;  
    }  
    cell_leakage_power : 3.0;  
}
```

leakage_power_unit Simple Attribute

This attribute indicates the units of the leakage-power values in the library.

Table 22 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;
```

default_cell_leakage_power Simple Attribute

The `default_cell_leakage_power` attribute indicates the default leakage power for those cells for which you have not set the `cell_leakage_power` attribute. This attribute must be a nonnegative floating-point number. The default is 0.0.

Example

```
default_cell_leakage_power : 0.5;
```

if the library has `cell_leakage_power` information but does not have the `default_cell_leakage_power` attribute defined.

Example

```
library(leakage) {
    delay_model : table_lookup;
```

Chapter 8: Modeling Power and Electromigration

Representing Leakage Power Information

```
/* unit attributes */
time_unit : "1ns";
voltage_unit : "1V";
current_unit : "1mA";
pulling_resistance_unit : "1kohm";
leakage_power_unit : "1pW";
capacitive_load_unit (1.0,PF);

cell (NAND2) {

    cell_leakage_power : 1.0 ;
    leakage_power() {
        related_pg_pin : "VDD1";
        when : "!A1 !A2" ;
        value : 1.5 ;
    }
    leakage_power() {
        related_pg_pin : "VDD1";
        when : "!A1 A2" ;
        value : 2.0 ;
    }
    leakage_power() {
        related_pg_pin : "VDD1";
        when : "A1 !A2" ;
        value : 3.0 ;
    }
    leakage_power() {
        related_pg_pin : "VDD1";
        when : "A1 A2" ;
        value : 4.0 ;
    }
    leakage_power() {
        related_pg_pin : "VDD2";
        when : "!A1 !A2" ;
        value : 3.5 ;
    }
    leakage_power() {
        related_pg_pin : "VDD2";
        when : "!A1 A2" ;
        value : 3.0 ;
    }
    leakage_power() {
        related_pg_pin : "VDD2";
        when : "A1 !A2" ;
        value : 4.0 ;
    }
    leakage_power() {
        related_pg_pin : "VDD2";
        when : "A1 A2" ;
        value : 5.0 ;
    }
    area : 1.0 ;
    pin(A1) {
```

Chapter 8: Modeling Power and Electromigration

Representing Leakage Power Information

```
    direction : input;
    capacitance : 0.1 ;
}
pin(A2) {
    direction : input;
    capacitance : 0.1 ;
}
pin(ZN) {
    direction : output;
    max_capacitance : 0.1;
    function : "(A1*A2)'";
    timing() {
        timing_sense : "negative_unate"
        related_pin : "A1"
        cell_rise( scalar ) {
            values("0.0");
        }
        rise_transition( scalar ) {
            values("0.0");
        }
        cell_fall( scalar ) {
            values("0.0");
        }
        fall_transition( scalar ) {
            values("0.0");
        }
        internal_power() {
            related_pin : " A1 "
            related_pg_pin : "VDD1";
            rise_power( scalar ) {
                values("0.0");
            }
            fall_power( scalar ) {
                values("0.0");
            }
        } /* end of internal power */
        internal_power() {
            related_pin : " A1 "
            related_pg_pin : "VDD2";
            rise_power( scalar ) {
                values("0.0");
            }
            fall_power( scalar ) {
                values("0.0");
            }
        }/* end of internal power */
    } /* end of timing for related pin A1 */
    timing() {
        timing_sense : "negative_unate"
        related_pin : "A2"
        cell_rise( scalar ) {
            values("0.0");
        }
    }
}
```

```
rise_transition( scalar ) {
    values("0.0");
}
cell_fall( scalar ) {
    values("0.0");
}
fall_transition( scalar ) {
    values("0.0");
}
internal_power() {
    related_pin : " A2 "
    related_pg_pin : "VDD1";
    rise_power( scalar ) {
        values("0.0");
    }
    fall_power( scalar ) {
        values("0.0");
    }
}/* end of internal power */
internal_power() {
    related_pin : " A2 "
    related_pg_pin : "VDD2";
    rise_power( scalar ) {
        values("0.0");
    }
    fall_power( scalar ) {
        values("0.0");
    }
} /* end of internal power */
} /* end of timing for related pin A2 */
} /* end of pin ZN */
} /* end of cell */
} /* end of library */
```

Threshold Voltage Modeling

Multiple threshold power saving flows require library cells to be categorized according to the transistor's threshold voltage characteristics, such as high threshold voltage cells and low threshold voltage cells. High threshold voltage cells exhibit lower power leakage but run slower than low threshold voltage cells.

You can specify low threshold voltage cells and high threshold voltage cells in the same library, simplifying library management and setup, by using the following optional attributes:

- `default_threshold_voltage_group`

Specify the `default_threshold_voltage_group` attribute at the library level, as shown, to specify a cell's category based on its threshold voltage characteristics:

```
default_threshold_voltage_group : group_nameid ;
```

The `group_name` value is a string representing the name of the category, such as `high_vt_cell` to represent a high voltage cell.

- `threshold_voltage_group`

Specify the `threshold_voltage_group` attribute at the cell level, as shown, to specify a cell's category based on its threshold voltage characteristics:

```
threshold_voltage_group : group_nameid ;
```

The `group_name` value is a string representing the name of the category. Typically, you would specify a pair of `threshold_voltage_group` attributes, one representing the high voltage and one representing the low voltage. However, there is no limit to the number of `threshold_voltage_group` attributes you can have in a library. If you omit this attribute, the value of the `default_threshold_voltage_group` attribute is applied to the cell.

Example

```
library ( mixed_vt_lib ) {
...
default_threshold_voltage_group : "high_vt_cell" ;
...
cell(ht_cell) {
    threshold_voltage_group : "high_vt_cell" ;
    ...
}
cell(lt_cell) {
    threshold_voltage_group : "low_vt_cell" ;
    ...
}
```

Modeling for Internal and Switching Power

These are two compatible definitions of internal or short-circuit power:

- Short-circuit power is the power dissipated by the instantaneous short-circuit connection between Vdd and GND while the gate is in transition.
- Internal power is all the power dissipated within the boundary of the gate. This definition does not distinguish between the cell's short-circuit power and the component of switching power that is being dissipated internally to the cell as a result of the drain-to-substrate capacitance that is being charged and discharged. In this definition, the interconnect switching power is the power dissipated because of lumped wire capacitance and input pin capacitances but not because of the output pin capacitance.

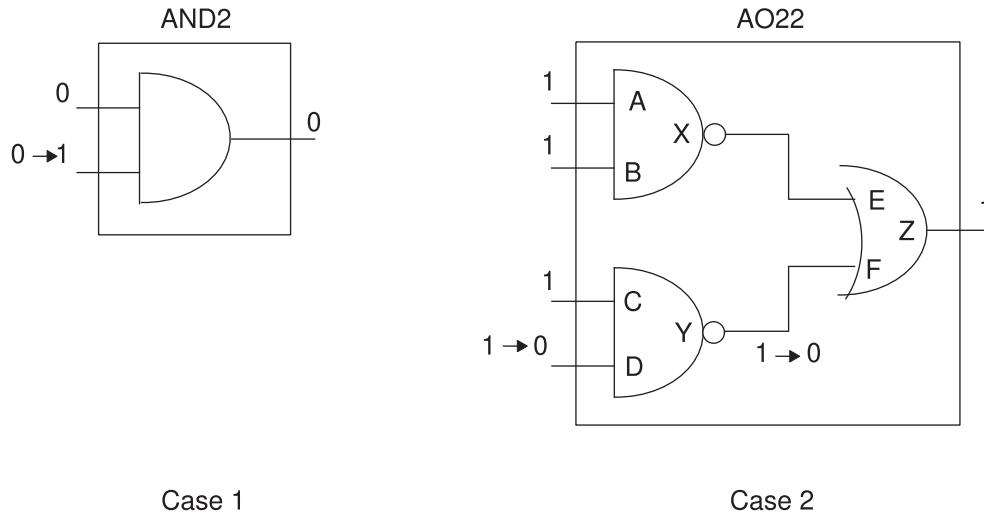
Library developers must choose one of these definitions and specify internal power and capacitance numbers accordingly. Library developers can choose

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

Together, internal power and switching power contribute to the total dynamic power dissipation. Like switching power, internal power is dissipated whenever an output pin makes a transition.

Some power is also dissipated as a result of internal transitions that do not cause output transitions. However, those are relatively minor in comparison (they consume much less power) and should be ignored.

Figure 47 Input Transitions Not Resulting in Output Transitions



In Figure 47 Case 1, input B of the AND2 gate undergoes a 0-to-1 transition but the output remains stable at 0. This might consume a small amount of power as one of the N-transistors opens, but the current flow is very small.

In Figure 47 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, one time for each input transition time and output load capacitance combination. For example, if internal power is modeled by use of a 3x3 table at the output of the cell, the design has 9 input voltage transitions—output load combinations where energy dissipation must be measured.

The library group supports a one-, two-, or three-dimensional internal power lookup table indexed by the total output load capacitances (best model), the input transition time, or both. The internal power lookup table uses the same syntax as the nonlinear lookup table for delay.

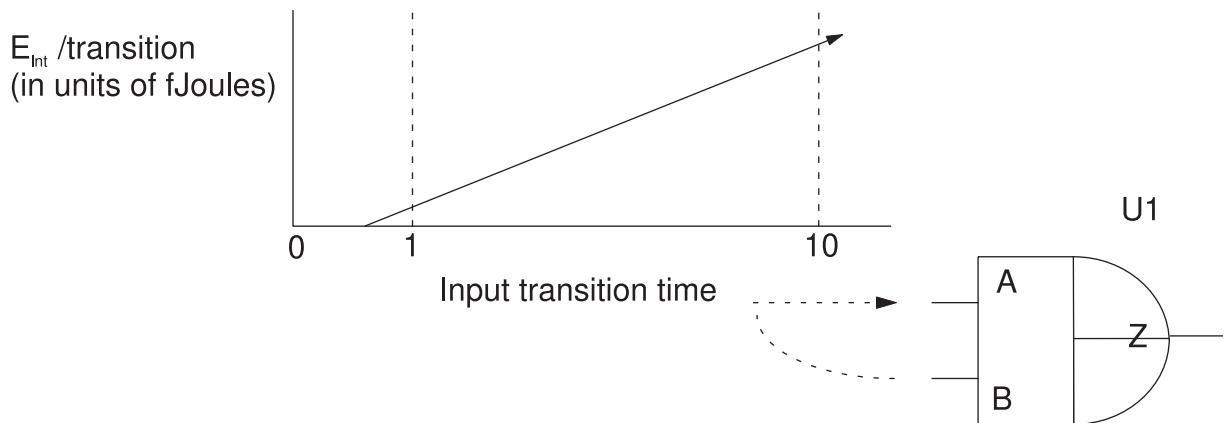
You can set the `internal_power` attributes for input pins, which are indexed by input transition time. In this way, you can model gate AO22 in figure in [Modeling for Internal and Switching Power](#) or, more important, the power consumed by the flip-flop clock or reset pins.

Note:

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

[Figure 48](#) shows how to calculate the input pin power information for the one-dimensional lookup table describing internal cell U1.

Figure 48 Internal Power for Cell U1



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

$$P_{\text{Int}} = (E_Z \cdot AAF_Z) + (E_A \cdot AAF_A) + (E_B \cdot AAF_B)$$

P_{Int}

Total internal power for the cell.

E

Internal energy for the pin.

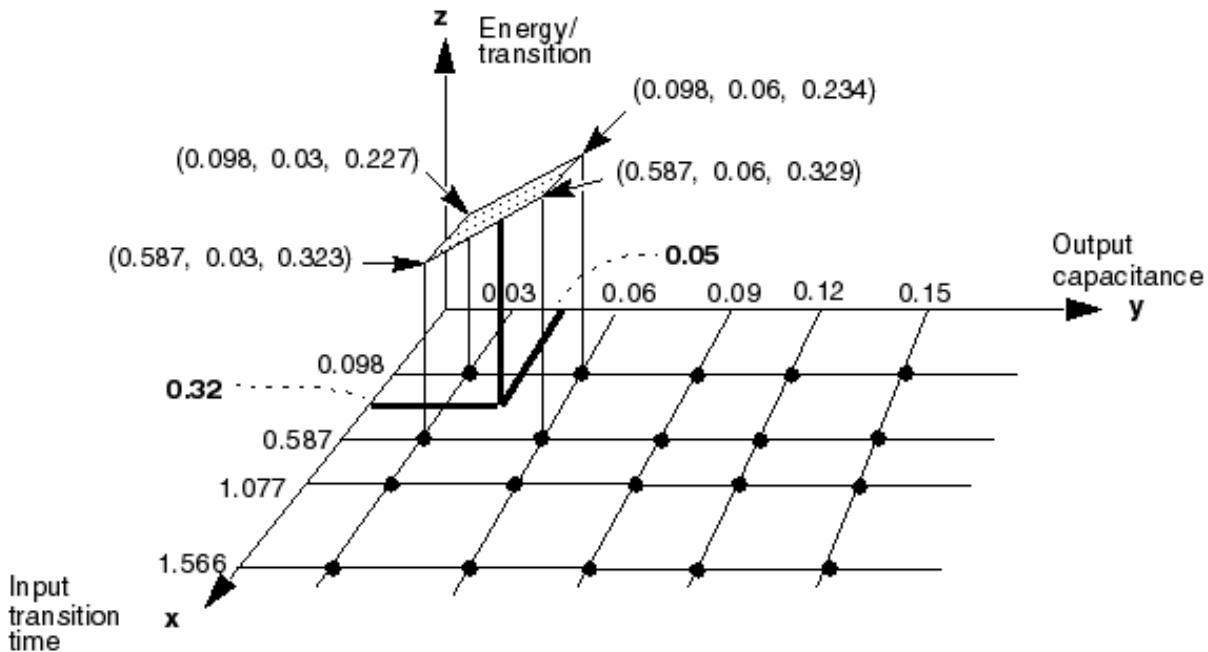
AF

Activity factor.

Accurate sequential modeling requires a separate table for the clock and for the output pin the clock controls. The two tables are used to ensure that clock pin power and output power are accounted for separately, because a clock pin often toggles without causing any observable state change on the output pin. The separate power table scheme ensures that power dissipated within the cell is accounted for properly when the clock pin toggles but the output pin does not. The following discussion pertains to single-output, single-clock sequential cells, but the concept is also extendable to multioutput, multiclock cells.

Figure 49 is an example of the two-dimensional lookup table for modeling output pin power in a cell.

Figure 49 Internal Power Table for Cell Output



Representing Internal Power Information

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

Specifying the Power Model

Use the library level `power_model` attribute to specify the power model for your library. The valid value is `table_lookup`.

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 318](#))
- The associated library-level attributes that specify the scaling factors and a default

power_lut_template Group

Use this library-level group to create templates of common information that multiple lookup tables can use. A table template specifies the table parameters and the breakpoints for each axis. Assign each template a name. Make the template name the group name of a `fall_power` group, `power` group, or `rise_power` group in the `internal_power` group.

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 318](#).

Template Breakpoints

The index statements define the breakpoints for an axis. The breakpoints defined by `index_1` correspond to the parameter values indicated by `variable_1`. The breakpoints defined by `index_2` correspond to the parameter values indicated by `variable_2`.

The breakpoints defined by `index_3` correspond to the parameter values indicated by `variable_3`.

You can overwrite the `index_1`, `index_2`, and `index_3` attribute values by providing the same `index_x` attributes in the `fall_power` group, `power` group, or `rise_power` group in the `internal_power` group.

The index values are lists of floating-point numbers greater than or equal to 0.0. The values in the list must be in an increasing order.

The number of floating-point numbers in the indexes determines the size of each dimension, as [Example 82](#) illustrates.

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

Example 82 power_lut_template Groups With One, Two, and Three-Dimensional Templates

```
power_lut_template (output_by_cap) {
    variable_1 : total_output_net_capacitance ;
    index_1 ("0.0, 5.0, 20.0") ;
}

power_lut_template (output_by_cap_and_trans) {
    variable_1 : total_output_net_capacitance ;
    variable_2 : input_transition_time ;
    index_1 ("0.0, 5.0, 20.0") ;
    index_2 ("0.1, 1.0, 5.0") ;
}

power_lut_template (input_by_trans) {
    variable_1 : input_transition_time ;
    index_1 ("0.0, 1.0, 5.0") ;
}
```

```
power_lut_template (output_by_cap2_and_trans) {  
    variable_1 : total_output_net_capacitance ;  
    variable_2 : input_transition_time ;  
    variable_3 : equal_or_output_net_capacitance ;  
    index_1 ("0.0, 5.0, 20.0") ;  
    index_2 ("0.1, 1.0, 5.0") ;  
    index_3 ("0.1, 0.5, 1.0") ;  
}
```

Scalar power_lut_template Group

Use this group to model cells with no power consumption.

The syntax has a predefined template named scalar; its value size is 1. You can specify scalar as the group name of a fall_power group, power group, or rise_power group in the internal_power group.

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.

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.

Example

```
...
bundle (Q) {
    members (Q0, Q1, Q2, Q3);
    direction : output;
    function : "IQ";
    internal_power (G_Q0, G_Q1, G_Q2, G_Q3) {
        related_pin : "G";
    }
}
```

If G is a pin, as opposed to another `bundle` group, the power relationships are as follows:

From pin	To pin	Label
G	Q0	G_Q0
G	Q1	G_Q1
G	Q2	G_Q2

G	Q3	G_Q3
---	----	------

If G is another bundle of member size 4 and G0, G1, G2, and G3 are members of bundle G, the power relationships are as follows:

From pin	To pin	Label
G0	Q0	G_Q0
G1	Q1	G_Q1
G2	Q2	G_Q2
G3	Q3	G_Q3

Note:

If G is a bundle of a member size other than 4, it is an error due to incompatible width.

Power Relationships Between a Bundle and Multiple Related Pins

When the `internal_power` group is within a `bundle` group that has several members, each having a corresponding related pin, enter the names of the resulting multiple power relationships as a name list in the `internal_power` group.

Example

```
bundle (Q) {
    members (Q0, Q1, Q2, Q3);
    direction : output;
    function : "IQ";
    internal_power (G_Q0, H_Q0, G_Q1, H_Q1, G_Q2, H_Q2, G_Q3, H_Q3) {
        related_pin : "G H";
    }
}
```

If G is a pin, as opposed to another `bundle` group, the power relationships are as follows:

From pin	To pin	Label
G	Q0	G_Q0
H	Q0	H_Q0
G	Q1	G_Q1

From pin	To pin	Label
----------	--------	-------

H	Q1	H_Q1
G	Q2	G_Q2
H	Q2	H_Q2
G	Q3	G_Q3
H	Q3	H_Q3

If G is another bundle of member size 4 and G0, G1, G2, and G3 are members of bundle G, the power relationships are as follows:

From pin	To pin	Label
G0	Q0	G_Q0
H	Q0	H_Q0
G1	Q1	G_Q1
H	Q1	H_Q1
G2	Q2	G_Q2
H	Q2	H_Q2
G3	Q3	G_Q3
H	Q3	H_Q3

The same rule applies if H is a size 4 bundle.

Note:

If G is a bundle of a member size other than 4, it's an error due to incompatible width.

Power Relationships Between a Bus and a Single Related Pin

This section describes how to identify the power relationships created when an `internal_power` group is within a `bus` group that has several bits that have the same single related pin. You identify the resulting multiple power relationships by entering a name list, using the `internal_power` group attribute.

Example

```
...
bus (X) {
```

Chapter 8: Modeling Power and Electromigration

Defining Internal Power Groups

```
/*assuming MSB is X[0] */
bus_type : bus4;
direction : output;
capacitance : 1;
pin (X[0:3]){
    function : "B'";
    internal_power (B_X0, B_X1, B_X2, B_X3) {
        related_pin : "B";
    }
}
}
```

If B is a pin, as opposed to another 4-bit bus, the power relationships are as follows:

From pin	To pin	Label
B	X[0]	B_X0
B	X[1]	B_X1
B	X[2]	B_X2
B	X[3]	B_X3

If B is another 4-bit bus and B[0] is the MSB for bus B, the power relationships are as follows:

From pin	To pin	Label
B[0]	X[0]	B_X0
B[1]	X[1]	B_X1
B[2]	X[2]	B_X2
B[3]	X[3]	B_X3

Power Relationships Between a Bus and Multiple Related Pins

This section describes the power relationships created when an `internal_power` group is within a `bus` group that has several bits, where each bit has its own related pin. You identify the resulting multiple power relationships by entering a name list, using the `internal_power` group attribute.

Example

```
bus (X){ /*assuming MSB is X[0] */
    bus_type : bus4;
    direction : output;
```

Chapter 8: Modeling Power and Electromigration
Defining Internal Power Groups

```
capacitance : 1;
pin (X[0:3]) {
    function : "B'";
    internal_power (B_X0, C_X0, B_X1, C_X1, B_X2, C_X2, B_X3, C_X3) {
        related_pin : "B' C";
    }
}
```

If B and C are pins, as opposed to another 4-bit bus, the power relationships are as follows:

From pin	To pin	Label
B	X[0]	B_X0
C	X[0]	C_X0
B	X[1]	B_X1
C	X[1]	C_X1
B	X[2]	B_X2
C	X[2]	C_X2
B	X[3]	B_X3
C	X[3]	C_X3

If B is another 4-bit bus and B[0] is the MSB for bus B, the power relationships are as follows:

From pin	To pin	Label
B[0]	X[0]	B_X0
C	X[0]	C_X0
B[1]	X[1]	B_X1
C	X[1]	C_X1
B[2]	X[2]	B_X2
C	X[2]	C_X2
B[3]	X[3]	B_X3
C	X[3]	C_X3

The same rule applies if C is a 4-bit bus.

Power Relationships Between a Bus and Related Bus Pins

This section describes the power relationships created when an `internal_power` group is within a `bus` group that has several bits that have to be matched with several related bus pins of a required width. Identify the resulting multiple power relationships by entering a name list, using the `internal_power` group attribute.

The first name in the name list is the relationship between the first pin of the related bus pins of the necessary width and the MSB in the `bus` group, the second name is the relationship between the second pin of the related bus pins and the second MSB in the `bus` group, and so on.

Example

```
/* assuming related_bus_pins is width of 2 bits */
bus (X) {
    /*assuming MSB is X[0] */
    bus_type : bus4;
    direction : output;
    capacitance : 1;
    pin (X[0:3]) {
        function : "B'";
        internal_power (B0_X0, B0_X1, B0_X2, B0_X3, B1_X0, B1_X1, \
                        B1_X2,B1_X3) {
            related_bus_pins : "B";
        }
    }
}
```

If B is another 2-bit bus and B[0] is its MSB, the power relationships are as follows:

From pin	To pin	Label
B[0]	X[0]	B0_X0
B[0]	X[1]	B0_X1
B[0]	X[2]	B0_X2
B[0]	X[3]	B0_X3
B[1]	X[0]	B1_X0
B[1]	X[1]	B1_X1
B[1]	X[2]	B1_X2
B[1]	X[3]	B1_X3

internal_power Group

To define an `internal_power` group in a `pin` group, use these simple attributes, complex attributes, and groups:

Simple attributes:

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

Complex attribute:

- `mode`

Groups:

- `fall_power`
- `power`
- `rise_power`

equal_or_opposite_output Simple Attribute

This attribute designates an optional output pin or pins whose capacitance can be used to access a three-dimensional table in the `internal_power` group.

Syntax

`equal_or_opposite_output : "name | name_list" ;`

`name | name_list`

Name of output pin or pins.

Note:

This pin (or these pins) have to be functionally equal to or the opposite of the pin named in the same `pin` group.

Example

```
equal_or_opposite_output : "Q" ;
```

Note:

The output capacitance of this pin (or pins) is used as the `equal_or_opposite_output_net_capacitance` value in the internal power lookup table.

falling_together_group Simple Attribute

Use the `falling_together_group` attribute to identify the list of two or more input or output pins that share logic and are falling together during the same time period. Set this time period with the `switching_interval` attribute. See [switching_interval Simple Attribute on page 329](#) for details.

Together, the `falling_together_group` attribute and the `switching_interval` attribute settings determine the level of power consumption.

Define a `falling_together_group` attribute in the `internal_power` group in a pin group, as shown here.

```
cell (name_string) {
    pin (name_string) {
        internal_power () {
            falling_together_group : "list of pins" ;
            rising_together_group : "list of pins" ;
            switching_interval : float;
            rise_power () {
                ...
            }
            fall_power () {
                ...
            }
        }
    }
}
```

list of pins

The names of the input or output pins that share logic and are falling during the same time period.

related_pin Simple Attribute

This attribute associates the `internal_power` group with a specific input or output pin. If `related_pin` is an output pin, it must be functionally equal to or the opposite of the pin in that `pin` group.

If `related_pin` is an input pin or output pin, the pin's transition time is used as a variable in the internal power lookup table.

Syntax

```
related_pin : "name | name_list" ;
```

name | name_list

Name of the input or output pin or pins

Example

```
related_pin : "A B" ;
```

The pin or pins in the `related_pin` attribute denote the path dependency for the `internal_power` group.

If you want to define a two-dimensional or three-dimensional table, specify all functionally related pins in a `related_pin` attribute.

rising_together_group Simple Attribute

The `rising_together_group` attribute identifies the list of two or more input or output pins that share logic and are rising during the same time period. This time period is defined with the `switching_interval` attribute. See the following “[switching_interval Simple Attribute](#),” section for details.

Together, the `rising_together_group` attribute and `switching_interval` attribute settings determine the level of power consumption.

Define the `rising_together_group` attribute in the `internal_power` group, as shown here.

```
cell (name_string) {
    pin (name_string) {
        internal_power () {
            falling_together_group : "list of pins" ;
            rising_together_group : "list of pins" ;
            switching_interval : float;
            rise_power () {
                ...
            }
            fall_power () {
                ...
            }
        }
    }
}
```

list of pins

The names of the input or output pins that share logic and are rising during the same time period.

switching_interval Simple Attribute

The `switching_interval` attribute defines the time interval during which two or more pins that share logic are falling, rising, or switching (either falling or rising) during the same time period.

Set the `switching_interval` attribute together with the `falling_together_group`, `rising_together_group`, or `switching_together_group` attribute. Together with one of these attributes, the `switching_interval` attribute defines a level of power consumption.

For details about the attributes that are set together with the `switching_interval` attribute, see [falling_together_group Simple Attribute on page 327](#); [rising_together_group Simple Attribute on page 328](#); and the following “[switching_together_group Simple Attribute](#),” section.

Syntax

```
switching_interval : float ;
```

float

A floating-point number that represents the time interval during which two or more pins that share logic are switching together.

Example

```
pin (Z) {
    direction : output;
    internal_power () {
        switching_together_group : "A B";
        /*if pins A, B, and Z switch*/
        switching_interval : 5.0;
        /*switching within 5 time units */;
        power () {
            ...
        }
    }
}
```

switching_together_group Simple Attribute

The `switching_together_group` attribute identifies the list of two or more input or output pins that share logic, are either falling or rising during the same time period, and are not affecting power consumption.

Define the time period with the `switching_interval` attribute. See [switching_interval Simple Attribute on page 329](#) for details.

Define the `switching_together_group` attribute in the `internal_power` group, as shown here.

```
cell (namestring) {
    pin (namestring) {
        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.

The table in [when Simple Attribute](#) lists the Boolean operators valid in a `when` statement.

Example

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

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 */
```

index_1, index_2, index_3 Attributes

These attributes identify internal cell consumption per fall transition. Define these attributes in the `internal_power` group or in the library-level `power_lut_template` group.

values Attribute

This attribute defines internal cell consumption per fall transition.

Example

```
values ("2.2, 3.7, 4.3", "1.7, 2.1, 3.5", "1.0, 1.5, 2.8");
```

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

The Design Compiler tool converts the `values` attribute information to power consumption by multiplying the unit by the factor `transition` or `per_unit_time`, as shown here:

- `nindex_1` floating-point numbers if the lookup table is one-dimensional
- `nindex_1 X nindex_2` floating-point numbers if the lookup table is two-dimensional
- `nindex_1 X nindex_2 X nindex_3` floating-point numbers if the lookup table is three-dimensional

The `nindex_1`, `nindex_2`, and `nindex_3` numbers are the size of `index_1`, `index_2`, and `index_3` in this group or in the `power_lut_template` group it inherits. Quotation marks enclose a group. Each group represents a row in the table.

power Group

The power group is defined within an internal_power group in a pin group at the cell level, as shown here:

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

Complex Attributes

```
index_1 ("float, ..., float") ; /* lookup table */
index_2 ("float, ..., float") ; /* lookup table */
index_3 ("float, ..., float") ; /* lookup table */
values ("float, ..., float") ; /* lookup table */
```

index_1, index_2, index_3 Attributes

These attributes identify internal cell consumption per transition. Define these attributes in the internal_power group or in the library-level power_lut_template group.

values Attribute

This attribute defines internal cell power consumption per rise or fall transition.

This power information is accessed when the pin has a rise transition or a fall transition. The values in the table specify the average power per transition.

Note:

The internal power value is derived from the capacitance unit exponent (with mantissa discarded) and the voltage unit.

rise_power Group

A rise_power group is defined in an internal_power group at the cell level, as shown here:

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

```
    }  
}
```

Rise power is accessed when the pin has a rise transition. If you have a `rise_power` group, you must have a `fall_power` group.

Complex Attributes

```
index_1 ("float, ..., float") ; /* lookup table */  
index_2 ("float, ..., float") ; /* lookup table */  
index_3 ("float, ..., float") ; /* lookup table */  
values ("float, ..., float") ; /* lookup table */
```

index_1, index_2, index_3 Attributes

These attributes identify internal cell consumption per rise transition. Define these attributes in the `internal_power` group or in the library-level `power_lut_template` group.

values Attribute

This attribute defines internal cell power consumption per rise transition.

Note:

The Design Compiler tool converts the `values` attribute information to power consumption, by multiplying the unit by the factor transition or `per_unit_time`.

For more information, see [values Attribute on page 331](#).

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");  
    }  
}
```

Chapter 8: Modeling Power and Electromigration

Defining Internal Power Groups

```
        }

        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" ;
}
```

Example 83 Library With Internal Power Information in pin Groups

```
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) {

```

Chapter 8: Modeling Power and Electromigration

Defining Internal Power Groups

```
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 ;
    ...
}
}
```

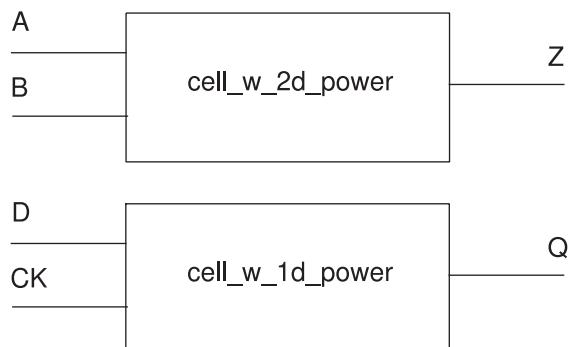
The syntax for a three-dimensional table using the `related_pin`, `fall_power`, and `rise_power` groups is shown here:

```
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" ;
}
```

Internal Power Examples

The examples in this section show how to describe the internal power of the 2-input sequential gate in [Figure 50](#), using one-, two-, and three-dimensional lookup tables.

Figure 50 Library Cells With Internal Power Information



One-Dimensional Power Lookup Table

[Example 84](#) is the library description of the cell shown in figure in [Internal Power Examples](#), cell with one-dimensional internal power table defined in the `pin` groups.

Example 84 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 84](#), the input transition time at pin CP is 1.0. The output load at pin Q is 5.0. The toggle rate at pin CP is two transitions per 100 ns. The toggle rate at pin Q is one transition per 100 ns. Using this information to index into the table, you get the following results:

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

If the `time_unit` attribute value in the logic library is 1 ns, the total internal power consumed by this gate is

$$P_{int} = (E_{CP} \times AF_{CP}) + (E_Q \times AF_Q)$$

$$\begin{aligned} &= (2.6 \times 2 \times 10^{-2}) + (5.0 \times 1 \times 10^{-2}) \\ &= 0.102 \end{aligned}$$

The activity factors AF_{CP} and AF_Q are adjusted to the `time_unit` specified in the logic library. The unit of P_{int} depends on the unit of the `voltage_unit` and `capacitive_load_unit` attributes in the logic library.

If the following attributes and values are specified in the logic library, the unit of the `internal_power` group is 10^{-15} joule/transition.

```
voltage_unit : "1V";
capacitive_load_unit(1.0, "ff");
```

Because the unit of the activity factor is transition/ns, the total internal power dissipation of this gate is

$$0.102 \times 10^{-15} \text{ joule/transition} \times \text{transition}/10^{-9}\text{second} = 0.102 \text{ uW.}$$

Note:

For a detailed explanation of how to calculate internal power, see the *Power Compiler User Guide*.

Two-Dimensional Power Lookup Table

[Example 85](#) is the library description of the cell in figure in [Internal Power Examples](#), a cell with a two-dimensional internal power table defined in the `pin` groups.

Example 85 Two-Dimensional Internal Power Table

```
library(internal_power_example) {
    ...
    power_lut template(output_by_cap_and_trans) {
        variable_1 : total_output_net_capacitance ;
        variable_2 : input_transition_time ;
        index_1 ("0.0, 5.0, 20.0");
        index_2 ("0.0, 1.0, 20.0");
    }
    ...
    cell(AN2) {
        pin(Z) {
            direction : output ;
            internal_power {
                power(output_by_cap_and_trans) {
                    values ("2.2, -3.7, 4.3", "1.7, 2.1, 3.5", "1.0, 1.5, 2.8");
                }
                related_pin : "A B" ;
            }
        }
        pin(A) {
            direction : input ;
            ...
        }
        pin(B) {
            direction : input ;
            ...
        }
    }
}
```

```

        }
    }
}
```

In this example, the input transition time at pin A is 1.25. The input transition time at pin B is 2.5. The output load at pin Z is 5.0. The toggle rate at pin A is 2 transitions per 100 ns. The toggle rate at pin B is three transitions per 100 ns. The toggle rate at pin Z is one transition per 100 ns.

With this information, the weighted input transition time is calculated as follows. (In this case, you use the weighted input transition time because there are two inputs.)

$$(1.25 \times 2 \times 10^{-2} + 2.5 \times 3 \times 10^{-2}) / (2 \times 10^{-2} + 3 \times 10^{-2}) = 2.0$$

Because the output load at pin Z is 5.0, the following power table values are used:

$$\begin{aligned} 2.1 &= 1.0A + B \\ 3.5 &= 20.0A + B \end{aligned}$$

Consequently, the values for A and B become

$$\begin{aligned} A &= 0.0739 \\ B &= 2.0263 \end{aligned}$$

and the resulting values for Ez and Pint are

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

$$\begin{aligned} Pint &= Ez \times AFz \\ &= 2.2 \times 1 \times 10^{-E2} \\ &= 0.022 \end{aligned}$$

The activity factor AF_Z is adjusted to the time_unit specified in the logic library. The unit of internal_power can be applied here.

Three-Dimensional Power Lookup Table

[Example 86](#) is the library description for the cell in figure in [Internal Power Examples](#), a cell with a three-dimensional internal power table.

Example 86 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") ;
    }
}
```

Chapter 8: Modeling Power and Electromigration
Defining Internal Power Groups

```
...
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,
                    .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 logic 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 logic library. The unit of `internal_power` can be applied here.

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

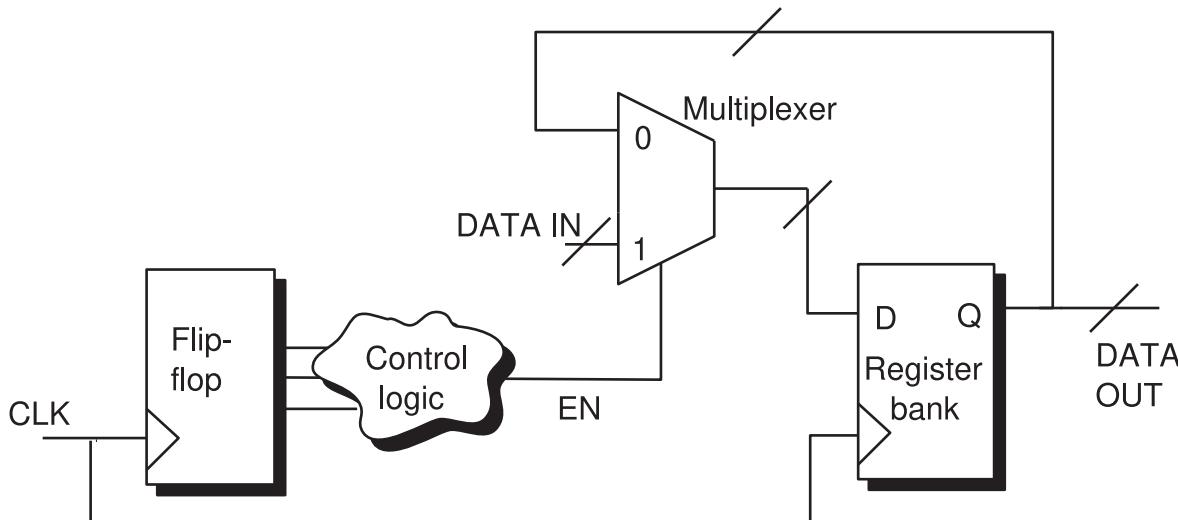
Clock gating provides a power-efficient implementation of register banks that are disabled during some clock cycles.

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

Figure 51 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 343.

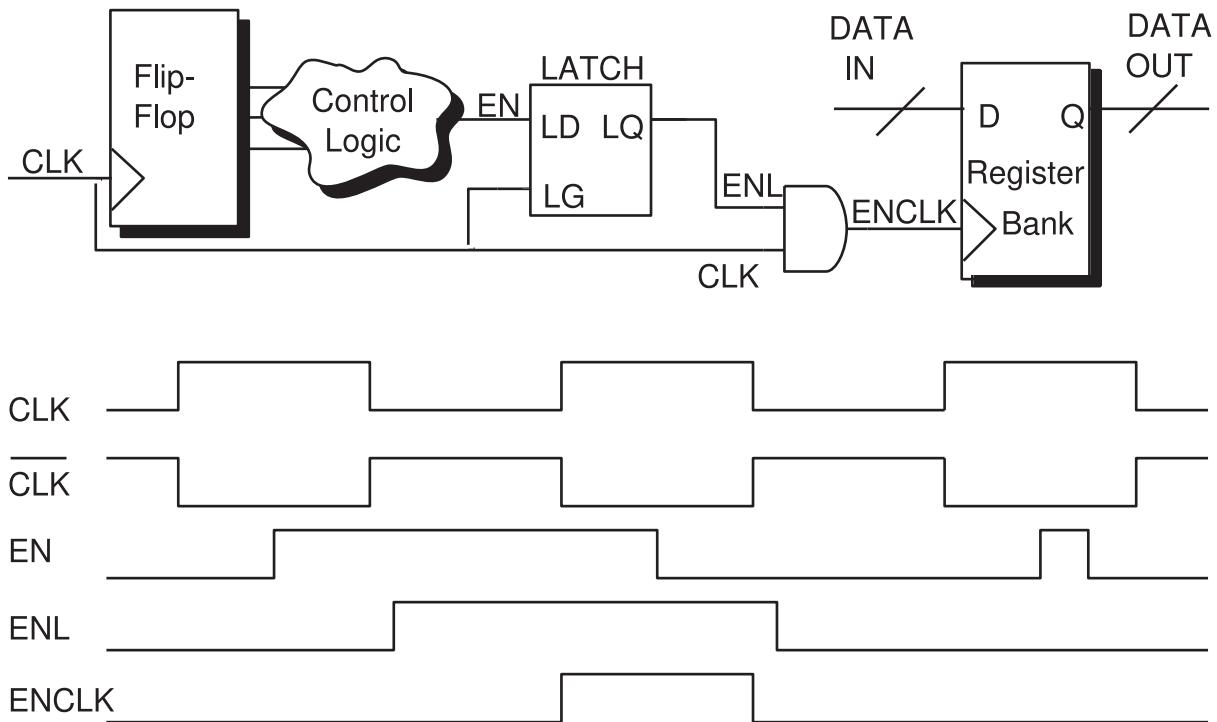
Looking at a Gated Clock

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 in [What Clock Gating Does](#), 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 52](#) 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.

Figure 52 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 in [What Clock Gating Does](#). 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 logic library.

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 23](#). Setting some of the pin attributes, such as those for test and observability, is optional.

Table 23 Pin Attributes for Integrated Clock-Gating Cells

Integrated cell pin name	Data direction	Required attribute
clock	in	clock_gate_clock_pin
enable	in	clock_gate_enable_pin
test_mode or scan_enable	in	clock_gate_test_pin
observability	out	clock_gate_obs_pin
enable_clock	out	clock_gate_out_pin

For more details about the `clock_gating_integrated_cell` attribute and the corresponding pin attributes, see the *Power Compiler User Guide*.

clock_gate_clock_pin Attribute

The `clock_gate_clock_pin` attribute identifies an input pin connected to a clock signal.

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

The `clock_gate_enable_pin` attribute a design containing gated clocks.

The `clock_gate_enable_pin` attribute identifies an input pin connected to an enable signal for nonintegrated clock-gating cells and integrated clock-gating cells.

Syntax

```
clock_gate_enable_pin : true | false ;
```

true | false

A true value labels the input pin of a clock-gating cell connected to an enable signal as the enable pin. A false value does not label the input pin as an enable pin.

Example

```
clock_gate_enable_pin : true;
```

For clock-gating cells, you can set this attribute to `true` on only one input port of a 2-input AND, NAND, OR, or NOR gate. If you do so, the other input port is the clock.

For more information about identifying pins on integrated clock-gating cells, see [Using an Integrated Clock-Gating Cell on page 342](#).

For additional information about integrated clock gating, see the *Power Compiler User Guide*.

clock_gate_test_pin Attribute

The `clock_gate_test_pin` attribute identifies an input pin connected to a `test_mode` or `scan_enable` signal.

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.

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.

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

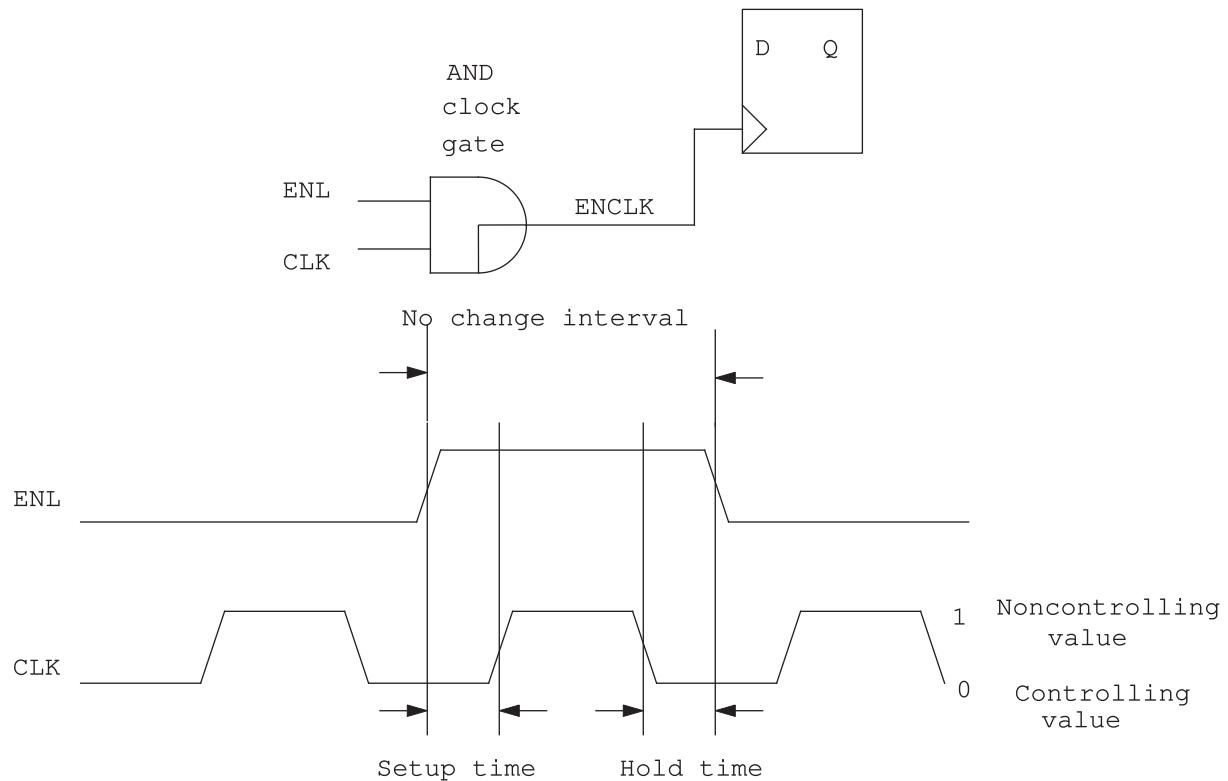
```
clock_gate_out_pin : true;
```

Clock-Gating Timing Considerations

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

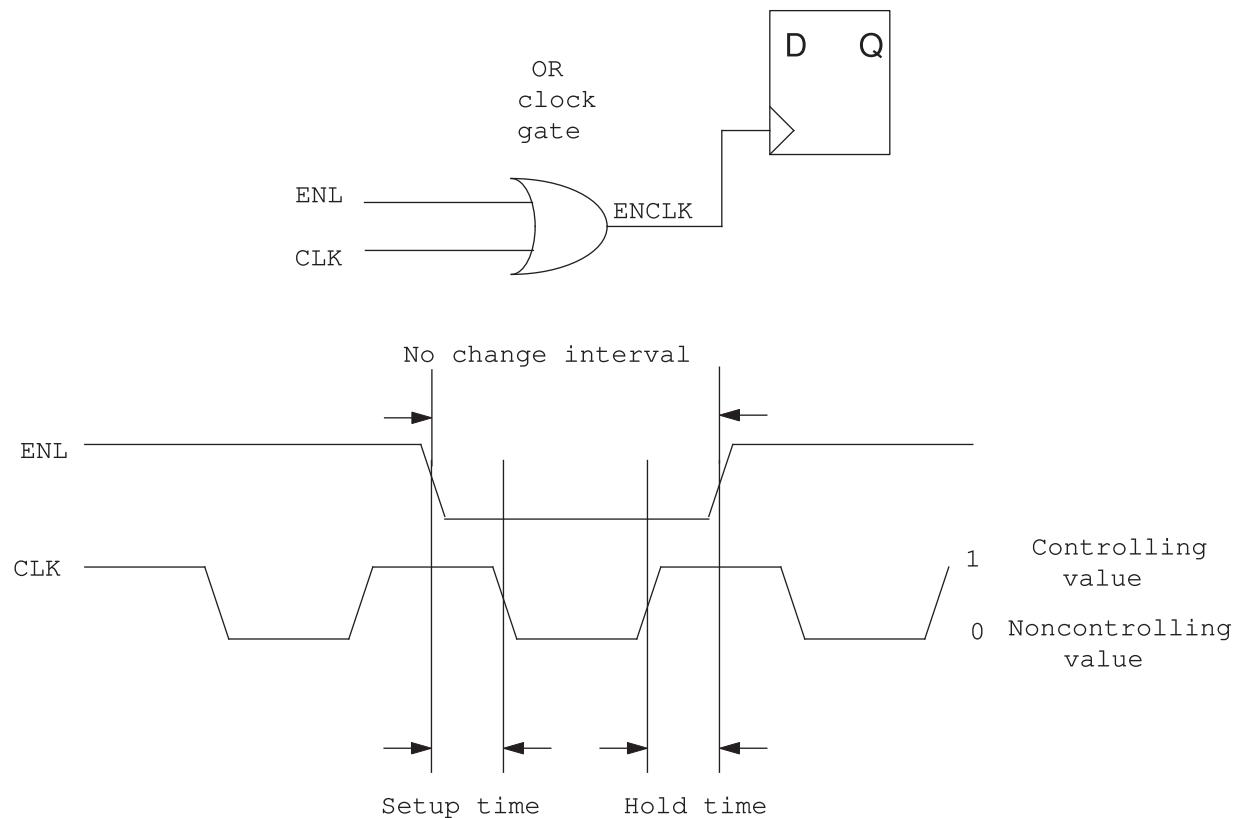
[Figure 53](#) and [Figure 54](#) show the relationship of setup and hold times to a clock waveform for AND and OR clock gates.

Figure 53 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 54 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.

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 24 Values of the `clock_gating_integrated_cell` Attribute for Setup and Hold Arcs

<code>clock_gating_integrated_cell</code> attribute value	Setup arc	Hold arc
<code>latch_posedge</code>	rising	rising
<code>latch_negedge</code>	falling	falling
<code>none_posedge</code>	falling	rising
<code>none_negedge</code>	rising	falling
<code>ff_posedge</code>	falling	falling
<code>ff_negedge</code>	rising	rising

Integrated Clock-Gating Cell Example

[Example 87](#) uses the `latch_posedge_precontrol_obs` value option for the `clock_gating_integrated_cell` attribute.

Example 87 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() {
```

Chapter 8: Modeling Power and Electromigration
Modeling Libraries With Integrated Clock-Gating Cells

```
timing_type : setup_rising;
cell_rise(scalar) {
    values( " 0.4 ");
}
cell_fall(scalar) {
    values( " 0.4 ");
}
related_pin : "CLK";
}
timing() {
    timing_type : hold_rising;
    cell_rise(scalar) {
        values( " 0.4 ");
    }
    cell_fall(scalar) {
        values( " 0.4 ");
    }
    related_pin : "CLK";
}
pin(SE) {
    direction : input;
    capacitance : 0.017997;
    clock_gate_test_pin : true;
}
pin(CLK) {
    direction : input;
    capacitance : 0.031419;
    clock_gate_clock_pin : true;
    min_pulse_width_low : 0.319;
}
pin(GCLK) {
    direction : output;
    state_function : "CLK * IQ";
    max_capacitance : 0.500;
    clock_gate_out_pin : true;
    timing() {
        timing_sense : positive_unate;
        cell_rise(scalar) {
            values( " 0.4 ");
        }
        rise_transition(scalar) {
            values( " 0.1 ");
        }
        cell_fall(scalar) {
            values( " 0.4 ");
        }
        fall_transition(scalar) {
            values( " 0.1 ");
        }
        related_pin : "EN CLK";
    }
    internal_power () {
        rise_power(li4X3) {
            index_1("0.0150, 0.0400, 0.1050, 0.3550");
            index_2("0.050, 0.451, 1.501");
            values("0.141, 0.148, 0.256", \
"0.162, 0.145, 0.234", \
"0.192, 0.200, 0.284", \
"0.199, 0.219, 0.297");
        }
    }
}
```

```
fall_power(li4X3){  
    index_1("0.0150, 0.0400, 0.1050, 0.3550");  
    index_2("0.050, 0.451, 1.500");  
    values("0.141, 0.148, 0.256", \  
           "0.162, 0.145, 0.234", \  
           "0.192, 0.200, 0.284", \  
           "0.199, 0.219, 0.297");  
}  
    related_pin : "CLK EN" ;  
}  
}  
pin(OBS) {  
    direction : output;  
    state_function : "EN";  
    max_capacitance : 0.500;  
    clock_gate_obs_pin : true;  
}  
}
```

Modeling Electromigration

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

Controlling Electromigration

You can control electromigration by establishing an upper boundary for the output toggle rate. You achieve this by annotating cells with electromigration characterization tables representing the net's maximal toggle rates.

Specifically, do the following to control electromigration:

Use the `em_lut_template` group to name an index template to be used by the `em_max_toggle_rate` group, which you use to set the net's maximum toggle rates. The `em_lut_template` group has `variable_1`, `variable_2`, `index_1`, and `index_2` attributes.

- Use the `variable_1` and the `variable_2` attributes to specify the variables. Both `variable_1` and `variable_2` can take the `input_transition_time` or the `total_output_net_capacitance` value.
- Use the `index_1` attribute to specify the points along the `variable_1` table axis, and the `index_2` attribute to specify the points along the `variable_2` table axis.

The pin-level `electromigration` group contains the `em_max_toggle_rate` group, which you use to specify the maximum toggle rate and the `related_pin` and `related_bus_pins` attributes.

- Use the `related_pin` attribute to identify the input pin with which the electromigration group is associated.
- Use the `related_bus_pins` attribute to identify the `bus` group of the pin or pins with which the electromigration group is associated.
- To select the template you want the `em_max_toggle_rate` group to use, assign the name of the library-level `em_lut_template` group to the pin-level `em_max_toggle_rate` group.

The `em_max_toggle_rate` group contains the `values` complex attribute; the `related_pin` and `related_bus_pins` simple attributes; and, optionally, the `index_1` and `index_2` complex attributes.

- Use the `values` complex attribute to specify the nets' maximum toggle rates for every `index_1` breakpoint along the `variable_1` axis if the table is one-dimensional.
- If the table is two-dimensional, use `values` to specify the nets' maximum toggle rates for all points where the breakpoints of `index_1` intersect with the breakpoints of `index_2`. The value for these points is equal to `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) {
    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 */
```

Group Statements

```
em_max_toggle_rate (em_template_name) {}
```

These attributes and groups are described in the following sections.

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.

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.

lifetime_profile Simple Attribute

The optional `lifetime_profile` attribute specifies a label that describes the lifetime of a chip. To support electromigration analysis at different lifetimes in downstream tools, define multiple `electromigration` groups with different values of the `lifetime_profile` attribute. This attribute does not have a default value.

An electromigration group where the `lifetime_profile` attribute is not specified, is considered to be the default cell electromigration model.

Syntax

```
lifetime_profile : profile_name ;
```

Example

```
lifetime_profile : lpf_alpf_a ;
```

em_max_toggle_rate Group

The `em_max_toggle_rate` group is a pin-level group that is defined within the `electromigration` `pin` group. The toggle rate is the number of toggles per unit of time where the unit of time is specified by the library-level `time_unit` attribute.

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

Simple Attribute

`current_type`

The `current_type` attribute is defined in the following section.

current_type Simple Attribute

The optional `current_type` attribute specifies the type of current for the `em_max_toggle_rate` lookup table. Valid values are `average`, `rms`, and `peak`.

Syntax

```
current_type: average | rms | peak ;
```

Example

```
current_type: average ;
```

Complex Attributes

```
index_1 ("float, ..., float") ; /*this attribute is
optional*/
index_2 ("float, ..., float") ; /*this attribute is
```

```
optional*/
values ("float, ..., float") ;
```

These attributes are defined in the following sections.

Index_1 and Index_2 Complex Attributes

You can use the `index_1` optional attribute to specify the breakpoints of the first dimension of an electromigration table to characterize cells for electromigration within the library. You can use the `index_2` optional attribute to specify breakpoints of the second dimension of an electromigration table used to characterize cells for electromigration within the library.

You can overwrite the values entered for the `em_lut_template` group's `index_1`, by entering values for the `em_max_toggle_rate` group's `index_1`. You can overwrite the values entered for the `em_lut_template` group's `index_2`, by entering values for the `em_max_toggle_rate` group's `index_2`.

Syntax

```
index_1 ("float, ..., float") ; /*this attribute is
optional*/
index_2 ("float, ..., float") ; /*this attribute is
optional*/
```

float

Floating-point numbers that identify the maximum toggle rate frequency.

Example

```
index_1 ("0.0, 5.0, 20.0") ;
index_2 ("0.0, 1.0, 2.0") ;
```

values Complex Attribute

This complex attribute is used to specify the net's maximum toggle rates.

This attribute can be a list of `nindex_1` positive floating-point numbers if the table is one-dimensional.

This attribute can also be `nindex_1 X nindex_2` positive floating-point numbers if the table is two-dimensional, where `nindex_1` is the size of `index_1` and `nindex_2` is the size of `index_2` that is specified for these two indexes in the `em_max_toggle_rate` group or in the `em_lut_template` group.

Syntax

```
values("float, ..., float") ;
```

float

Floating-point numbers that identify the maximum toggle rate frequency.

Example (One-Dimensional Table)

```
values : ("1.5, 1.0, 0.5") ;
```

Example (Two-Dimensional Table)

```
values : ("2.0, 1.0, 0.5", "1.5, 0.75, 0.33", "1.0, 0.5, 0.15",) ;
```

em_temp_degradation_factor Simple Attribute

The `em_temp_degradation_factor` attribute specifies the electromigration exponential degradation factor.

Syntax

```
em_temp_degradation_factor : valuefloat ;
```

value

A floating-point number in centigrade units consistent with other temperature specifications throughout the library.

Example

```
em_temp_degradation_factor : 40.0 ;
```

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.

Cell Electromigration Example

The following example shows a complete cell electromigration model.

```
library (em) {
...
    em_temp_degradation_factor : 40.0;
    em_lut_template(output_by_cap) {
        variable_1 : total_output_net_capacitance;
        index_1 ("0.0, 0.087, 0.272, 0.329");
        domain ( best ) {
            calc_mode : best;
        }
    }
    em_lut_template(input_by_trans) {
        variable_1 : input_transition_time;
```

Chapter 8: Modeling Power and Electromigration

Modeling Electromigration

```
    index_1 ("0.0, 0.889, 5.0");
}
em_lut_template(output_by_cap_and_trans) {
    variable_1 : total_output_net_capacitance;
    variable_2 : input_transition_time;
    index_1 ("0.0, 0.087, 0.272, 0.329");
    index_2 ("0.0, 0.889, 5.0");
}
cell(testcell) {
    ...
    pin(Z) {
        ...
        electromigration() {
            em_max_toggle_rate(output_by_cap_and_trans) {
                values("0.009, 0.163, 0.758",
                    "0.609, 0.628, 0.758",
                    "1.792, 1.792, 2.003",
                    "2.162, 2.162, 2.313");
            }
            related_pin : "A";
            when : "!A & !Z";
        }
        electromigration() {
            em_max_toggle_rate(output_by_cap) {
                values("1.0 2.087 2.272 6.329");
                domain ( best ) {
                    values("1.0 2.087 2.272 6.329");
                }
            }
            /* related_pin : "Z"; */
            when : "!Z";
        }
        timing();
        ...
    } /* end pin group Z */
    ...
    pin(A) {
        direction : input;
        capacitance : 1.000;

        electromigration() {
            em_max_toggle_rate(input_by_trans) {
                values("1.0, 2.889, 5.0");
            }
        }
        /* related_pin : "A"; */
        when : "!A";
    }
}
```

Chapter 8: Modeling Power and Electromigration
Modeling Electromigration

```
electromigration() {
    em_max_toggle_rate(input_by_trans) {
        values("1.0, 2.889, 5.0");
    }
}

electromigration() {
    em_max_toggle_rate(input_by_trans) {
        values("1.0, 2.889, 5.0");
    }
}
} /* end pin group A */
} /* end cell */
} /* end library */
```

9

Advanced Low-Power Modeling

Advanced low-power design methodologies such as multivoltage and multithreshold-CMOS require special library cells. These cells include power management attributes to drive implementation tools during library cell selection. This chapter describes the power and ground (PG) pin syntax and provides modeling examples of the special cells, such as the level-shifter, isolation, switch, and retention cells.

Note:

To model very deep submicron (VDSM) logic libraries, you must use the PG pin syntax.

This chapter includes the following sections:

- [Power and Ground \(PG\) Pins](#)
- [PG Pin Power Mode Modeling](#)
- [Feedthrough Signal Pin Modeling](#)
- [Overdrive and Underdrive Voltage Modeling](#)
- [Internally Unconnected Pin Modeling](#)
- [Silicon-on-Insulator \(SOI\) Cell Modeling](#)
- [Level-Shifter Cells in a Multivoltage Design](#)
- [Isolation Cell Modeling](#)
- [Switch Cell Modeling](#)
- [Retention Cell Modeling](#)
- [Always-On Cell Modeling](#)
- [Macro Cell Modeling](#)
- [Modeling Antenna Diodes](#)

Power and Ground (PG) Pins

The syntax supports power and ground (PG) library pins. A power pin is a current source pin, and a ground pin is a current sink pin. This section provides an overview of PG pins.

Example Libraries With Multiple Power Supplies

[Example 88](#) shows a library with multiple power supplies defined at the library, cell, and pin levels.

Example 88 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 : VDD2;
    output_signal_level : VDD2;
}
}
}
```

Example 89 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 90 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 91 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 92 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 93 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 */
```

Partial PG Pin Cell Modeling

Partial PG pin cells are cells that have only power pins, only ground pins, or do not have power or ground PG pins.

Table 25 Categories of Partial PG Pin Cells

Partial PG Pin Cell	Description
Cells with one power pin and one ground pin	These cells are defined as having <ul style="list-style-type: none">• One or more primary power PG pins• One or more primary ground PG pins• Zero, or at least one, backup or internal power PG pins• Zero, or at least one, backup or internal ground PG pins• Zero, or at least one, nwell bias PG pins• Zero, or at least one, pwell bias PG pins
Cells with one power pin and no ground pins	These cells are defined as having <ul style="list-style-type: none">• One or more primary power PG pins• No primary ground PG pins• Zero, or at least one, backup or internal power PG pins• Zero, or at least one, backup or internal ground PG pins• Zero, or at least one, nwell bias PG pins• Zero, or at least one, pwell bias PG pins

Partial PG Pin Cell	Description
Cells with no power pins and one ground pin	These cells are defined as having <ul style="list-style-type: none">• No primary power PG pins• At least one primary ground PG pin• Zero, or at least one, backup or internal power PG pins• Zero, or at least one, backup or internal ground PG pins• Zero, or at least one, nwell bias PG pins• Zero, or at least one, pwell bias PG pins
Cells with no power pins or ground pins	These cells are defined as having <ul style="list-style-type: none">• No primary power PG pins• No primary ground PG pins• No backup or internal power PG pin• No backup or internal ground PG pin• Zero, or at least one, nwell bias PG pins• Zero, or at least one, pwell bias PG pins

Special Partial PG Pin Cells

The following types of partial PG pin cells are acceptable with certain conditions.

- ETM cells

ETM cells do not need a pair of PG pins. A cell is identified as an ETM cell if it has the `interface_timing` or `timing_model_type` attribute.

- Black box cells

A black box cell with no timing, noise, or power information does not need at least one power pin and one ground PG pin.

- Metal fills and antenna cells

Black box cells without functions do not need at least power pin and one ground pin.

- Cells without signal pins

Cells without signal pins do not need at least one power pin and one ground PG pin.

- Load cells

Cells without inout or output signal pins require either a power or a ground pin, but it is not necessary to have both.

- Tied-off cells and extensions

The following types of cells can be specified with one power pin and no ground PG pins:

- Cells with at least one inout or output pin with the `function` attribute set to `1`.
- Cells with a pull-up signal, for example with the `driver_type` attribute set to `pull_up` or `pull_up_function`.
- Cells with a resistive_1 signal, for example with the `driver_type` attribute set to `resistive_1` or `resistive_1_function`.

The following types of cells can be specified with no power pins and one ground PG pin:

- Cells with at least one inout or output pin with the `function` attribute set to `0`.
- Cells with a pull-down signal, for example with the `driver_type` attribute set to `pull_down` or `pull_down_function`.
- Cells with a resistive_0 signal, for example with the `driver_type` attribute set to `resistive_0` or `resistive_0_function`.

Supported Attributes

Cells with at least one power pin and no ground pins, cells with no power pins and at least one ground pin, and cells with no power pins or ground pins support the following attributes:

- To prevent a cell from being automatically inserted into the netlist, specify the `dont_touch` or the `dont_use` attribute. The `dont_touch` attribute set to `true` indicates that all instances of the cell must remain in the network. The `dont_use` attribute set to `true` indicates that a cell must not be added to a design during optimization.
- Cells with partial PG pins are reported using the following attributes:
 - `1p0g`

Reports cells with at least one power pin and no ground PG pins.

- 0p1g
 - Reports cells with no power pins and at least one ground PG pin.
- 0p0g
 - Reports cells with no power pins or ground PG pins.

For more information about library reports for partial PG pin cells, see the “Generating Library Reports” chapter in the *Library Quality Assurance System User Guide*.

Partial PG Pin Cell Example

[Example 94](#) shows a cell with only one primary ground pin.

Example 94 Partial PG Pin Cell With Only One Primary Ground Pin

```
cell (PULLDOWN) {
    pg_pin ( VSS ) {
        voltage_name : VSS;
        pg_type : primary_ground;
    }

    area : 1.0;
    dont_touch : true;
    dont_use : true;

    pin (X) {
        related_ground_pin : VSS;
        direction : output;
        function : "0";
        three_state : "!A";
        max_capacitance : 0.19;
        timing() {
            related_pin : "A";
            timing_type : three_state_enable;
            cell_rise(scalar) { values ( "0.1"); }
            rise_transition(scalar) { values ( "0.1"); }
            cell_fall(scalar) { values ( "0.1"); }
            fall_transition(scalar) { values ( "0.1"); }
        }
        timing() {
            related_pin : "A";
            timing_type : three_state_disable;
            cell_rise(scalar) { values ( "0.1"); }
            rise_transition(scalar) { values ( "0.1"); }
            cell_fall(scalar) { values ( "0.1"); }
            fall_transition(scalar) { values ( "0.1"); }
        }
    }
    pin (A) {
        related_ground_pin : VSS;
```

```
    direction : input;
    capacitance : 0.1;
    rise_capacitance : 0.1;
    rise_capacitance_range (0.1, 0.2);
    fall_capacitance : 0.1;
    fall_capacitance_range (0.1, 0.2);
}
}
```

PG Pin Syntax

The power and ground pin syntax for generic cells is as follows:

```
library(library_name) {
    ...
    voltage_map(voltage_name, voltage_value);
    voltage_map(voltage_name, voltage_value);
    ...
    operating_conditions(oc_name) {
        ...
        voltage : value;
        ...
    }
    ...
    default_operating_conditions : oc_name;
    cell(cell_name) {
        pg_pin(pg_pin_name_p1) {
            voltage_name : voltage_name_p1;
            pg_type : type_value;
        }
        pg_pin(pg_pin_name_g1) {
            voltage_name : voltage_name_g1;
            pg_type : type_value;
        }
        pg_pin(pg_pin_name_p2) {
            voltage_name : voltage_name_p2;
            pg_type : type_value;
        }
        pg_pin(pg_pin_name_g2) {
            voltage_name : voltage_name_g2;
            pg_type : type_value;
        }
        ...
        leakage_power() {
            related_pg_pin : pg_pin_name_p1;
            ...
        }
        ...
        pin(pin_name1) {
            direction : input | inout;
            related_power_pin : pg_pin_name_p1;
            related_ground_pin : pg_pin_name_g1;
        }
    }
}
```

```
    ...
}
...
pin (pin_name2) {
    direction : inout | output;
    power_down_function : (!pg_pin_name_p1 + !pg_pin_name_p2 + \
    !pg_pin_name_g1 + !pg_pin_name_g2) ;
    related_power_pin : pg_pin_namen_p2;
    related_ground_pin : pg_pin_namen_g2;
    output_signal_voltage_low : float;
    output_signal_voltage_high : float;
    timing () {
        ...
        output_signal_voltage_low : float;
        output_signal_voltage_high : float;
    }
    internal_power() {
        related_pg_pin : pg_pin_name_p2;
    ...
    } /* end internal_power group */
    ...
}/* end pin group*/
...
}/* end cell group*/
...
}/* end library group*/
```

Library-Level Attributes

This section describes library-level attributes.

voltage_map Complex Attribute

The `voltage_map` attribute associates the voltage name with the relative voltage values. These voltage names are referenced by the `pg_pin` groups defined at the cell level. When specified in a library, this attribute identifies the library as a power and ground pin library. At least one voltage map in the library should have a value of 0, which becomes the reference value to which other voltage map values relate.

default_operating_conditions Simple Attribute

The `default_operating_conditions` attribute specifies the name of the default `operating_conditions` group in the library, which helps to identify the operating condition process, voltage, and temperature (PVT) points that are used during library characterization.

Cell-Level Attributes

This section describes cell-level attributes for `pg_pin` groups.

pg_pin Group

The `pg_pin` groups are used to represent the power and ground pins of a cell. Library cells can have multiple power and ground pins. The `pg_pin` groups are mandatory for each cell using the power and ground pin syntax, and a cell must have at least one `primary_power` power pin and at least one `primary_ground` ground pin.

is_pad Simple Attribute

The `is_pad` attribute identifies a pad pin on any I/O cell. The valid values are `true` and `false`. You can also specify the `is_pad` attribute on a PG pin. If the cell-level `pad_cell` attribute is specified on an I/O cell, you must set the `is_pad` attribute to `true` in either a `pg_pin` group or on a signal pin for that cell.

voltage_name Simple Attribute

The `voltage_name` string attribute is mandatory in all `pg_pin` groups except for a level-shifter cell not powered by the switching power domains. The `voltage_name` attribute specifies the associated voltage name of the power and ground pin defined using the `voltage_map` complex attribute at the library level.

Note:

To allow implementation tools to include a level-shifter cell not powered by the switching domains in any other power domain, the voltage value of the PG pin with the `std_cell_main_rail` attribute is ignored. So for this cell, the `voltage_name` attribute is optional in the `pg_pin` group that has the `std_cell_main_rail` attribute and not connected to the signal pins.

pg_type Simple Attribute

The `pg_type` attribute, optional in `pg_pin` groups, specifies the type of power and ground pin. The `pg_type` attribute can have the following values: `primary_power`, `primary_ground`, `backup_power`, `backup_ground`, `internal_power`, `internal_ground`, `pwell`, `nwell`, `deepnwell` and `deeppwell`.

The `pg_type` attribute also supports substrate-bias modeling. *Substrate bias* is a technique in which a *bias voltage* is varied on the substrate terminal of a CMOS device. This increases the threshold voltage, the voltage required by the transistor to switch, which helps reduce transistor power leakage. The `pg_type` attribute provides the `pwell`, `nwell`, `deepnwell` and `deeppwell` values to support substrate-bias modeling. The `pwell` and `nwell` values specify regular wells, and `deeppwell` and `deepnwell` specify isolation wells.

[Table 26](#) describes the `pg_type` values.

Table 26 pg_type Values

Value	Description
primary_power	Specifies that pg_pin is a primary power source (the default). If the pg_type attribute is not specified, primary_power is the pg_type value.
primary_ground	Specifies that pg_pin is a primary ground source.
backup_power	Specifies that pg_pin is a backup (secondary) power source (for retention registers, always-on logic, and so on).
backup_ground	Specifies that pg_pin is a backup (secondary) ground source (for retention registers, always-on logic, and so on).
internal_power	Specifies that pg_pin is an internal power source for switch cells.
internal_ground	Specifies that pg_pin is an internal ground source for switch cells.
pwell	Specifies regular p-wells for substrate-bias modeling.
nwell	Specifies regular n-wells for substrate-bias modeling.
deepnwell	Specifies isolation n-wells for substrate-bias modeling.
deppwell	Specifies isolation p-wells for substrate-bias modeling.

physical_connection Simple Attribute

The physical_connection attribute can have the following values: device_layer and routing_pin. The device_layer value specifies that the bias connection is physically external to the cell. In this case, the library provides biasing tap cells that connect through the device layers. The routing_pin value specifies that the bias connection is inside a cell and is exported as a physical geometry and a routing pin. Macros with pin access generally use the routing_pin value if the cell has bias pins with geometry that is visible in the physical view.

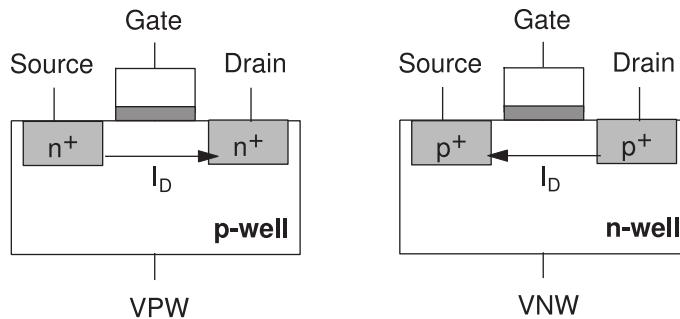
related_bias_pin Attribute

The related_bias_pin attribute defines all bias pins associated with a power or ground pin within a cell. The related_bias_pin attribute is required only when the attribute is declared in a pin group but it does not specify a complete relationship between the bias pin and power and ground pin for a library cell.

The related_bias_pin attribute also defines all bias pins associated with a signal pin. To associate substrate-bias pins to signal pins, use the related_bias_pin attribute

to specify one of the following pg_type values: pwell, nwell, deeppwell, deepnwell. Figure 55 shows transistors with p-well and n-well substrate-bias pins.

Figure 55 Transistors With p-well and n-well Substrate-Bias Pins



user_pg_type Simple Attribute

The user_pg_type optional attribute allows you to customize the type of power and ground pin. It accepts any string value. The following example shows a pg_pin library with the user_pg_type attribute specified.

```
pg_pin (pg_pin_name) {
    voltage_name : voltage_name;
    pg_type : primary_power | primary_ground |
    backup_power | backup_ground |
    internal_power | internal_ground;
    user_pg_type : user_pg_type_name;
}
```

Pin-Level Attributes

This section describes pin-level attributes.

direction Attribute

The direction attribute declares a pin as being an input, output, inout (bidirectional), or internal pin. The default is input.

power_down_function Attribute

The power_down_function string attribute specifies the Boolean condition under which the cell's output pin is switched off by the state of the power and ground pins (when the cell is in off mode due to the external power pin states).

You specify the `power_down_function` attribute for combinational and sequential cells. For simple and complex sequential cells, `power_down_function` also determines the condition of the cell's internal state.

For more information about using the `power_down_function` attribute for sequential cells, see [Chapter 5, Defining Sequential Cells](#).

related_power_pin and related_ground_pin Attributes

The `related_power_pin` and `related_ground_pin` attributes associate a predefined power and ground pin with the signal pin, in which they are defined. This behavior only applies to standard cells. For special cells, explicitly specify this relationship.

The `pg_pin` groups are mandatory for each cell. Because a cell must have at least one `primary_power` and at least one `primary_ground` pin, a default `related_power_pin` and `related_ground_pin` always exists in any cell.

output_signal_level_low and output_signal_level_high Attributes

You can define the `output_signal_level_low` and `output_signal_level_high` attributes for output and inout pins. The regular signal swings are derived for regular cells using the `related_power_pin` and `related_ground_pin` specifications.

You can also define the `output_signal_level_low` and `output_signal_level_high` attributes in timing arcs. See [output_signal_level_low and output_signal_level_high Attributes on page 374](#).

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).

Timing-Level Attributes

This section describes timing-level attributes.

output_signal_level_low and output_signal_level_high Attributes

The optional `output_signal_level_low` and `output_signal_level_high` attributes specify the actual output voltages of an output pin after a transition through a timing arc.

The `output_signal_level_low` attribute specifies the minimum voltage after a falling transition to state 0.

The `output_signal_level_high` attribute specifies the maximum voltage value after a rising transition to state 1.

Define the `output_signal_level_low` and `output_signal_level_high` attributes in the timing group when the following occur together:

- The cell output exhibits partial voltage swing (and not rail-to-rail swing).
- The voltages are different in different timing arcs.

Specify the voltages with respect to the `voltage_unit` defined in the library. The voltage values that you specify in the timing group override the voltages specified under the `pin` group. The attributes do not have a default.

Specifying Delay and Slew Attributes in Voltage Range

When partial voltage swing is defined in a `pin` or a `timing` group, the nonlinear delay model (NLDM) data in that group is for the partial swing. You must apply the following threshold and trip point attributes only in the voltage range from the `output_signal_level_low` value to the `output_signal_level_high` value:

- `input_threshold_pct_rise`
- `output_threshold_pct_rise`
- `input_threshold_pct_fall`
- `output_threshold_pct_fall`
- `slew_lower_threshold_pct_rise`
- `slew_upper_threshold_pct_rise`
- `slew_lower_threshold_pct_fall`
- `slew_upper_threshold_pct_fall`

The following example shows a library description with the `output_signal_level_low` and `output_signal_level_high` attributes defined in both the `pin` and the `timing` groups.

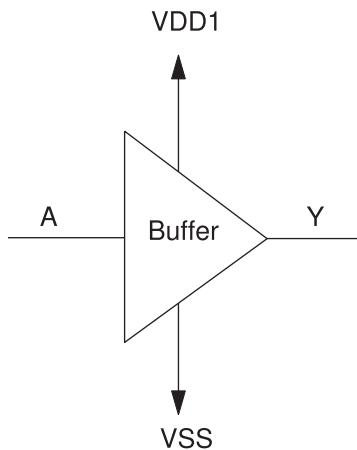
```
library (lib1) {  
    ...  
    cell (cell1) {
```

```
...
pin(pin1) {
    direction: output;
    output_signal_level_low: 0.15;
    output_signal_level_high: 0.75;
    timing() {
        ...
        related_pin : "CLKIN" ;
        timing_type : rising_edge ;
        output_signal_level_low: 0.1;
        output_signal_level_high: 0.7;
        ...
    } /* end of timing */
    ...
} /* end of pin */
...
} /* end of cell */
...
} /* end of library */
```

Standard Cell With One Power and Ground Pin Example

[Figure 56](#) shows a standard cell with a power and ground pin. The figure is followed by an example.

Figure 56 Standard Cell Buffer Schematic



```
library(standard_cell_library_example) {

    voltage_map(VDD, 1.0);
    voltage_map(VSS, 0.0);

    operating_conditions(XYZ) {
```

Chapter 9: Advanced Low-Power Modeling
Power and Ground (PG) Pins

```
voltage : 1.0;
...
}
default_operating_conditions : XYZ;

cell(BUF) {

    pg_pin(VDD) {
        voltage_name : VDD;
        pg_type : primary_power;
    }
    pg_pin(VSS) {
        voltage_name : VSS;
        pg_type : primary_ground;
    }

    leakage_power() {
        related_pg_pin : VDD;
        when : "!A";
        value : 1.5;
    }

    pin(A) {
        related_power_pin : VDD;
        related_ground_pin : VSS;
    }

    pin(Y) {
        direction : output;
        power_down_function : "!VDD + VSS";
        related_power_pin : VDD;
        related_ground_pin : VSS;

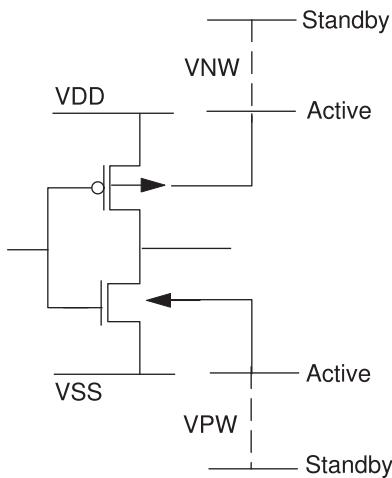
        timing() {
            related_pin : A;
            cell_rise(template) {
                ...
            }
            cell_fall(template) {
                ...
            }
            rise_transition(template) {
                ...
            }
            fall_transition(template) {
                ...
            }
        }
        internal_power() {
            related_pin : A;
            related_pg_pin : VDD;
            ...
        }
    }
}
```

```
 }/* end pin group*/  
 ...  
 }/*end cell group*/  
 ...  
 }/*end library group*/
```

Inverter With Substrate-Bias Pins Example

Figure 57 shows an inverter with substrate-bias pins. The figure is followed by an example.

Figure 57 Inverter With Substrate-Bias Pins



```
library(low_power_cells) {  
  
    delay_model : table_lookup;  
  
    /* unit attributes */  
    time_unit : "1ns";  
    voltage_unit : "1V";  
    current_unit : "1mA";  
    pulling_resistance_unit : "1kohm";  
    leakage_power_unit : "1pW";  
    capacitive_load_unit (1.0,pf);  
  
    voltage_map(VDD, 0.8); /* primary power */  
    voltage_map(VSS, 0.0); /* primary ground */  
    voltage_map(VNW, 0.8); /* bias power */  
    voltage_map(VPW, 0.0); /* bias ground */
```

Chapter 9: Advanced Low-Power Modeling
Power and Ground (PG) Pins

```
/* operation conditions */
operating_conditions(XYZ) {
    process: 1;
    temperature: 125;
    voltage: 0.8;
    tree_type: balanced_tree
}
default_operating_conditions : XYZ;

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

/* default attributes */
default_cell_leakage_power: 0.0;
default_fanout_load: 1.0;
default_output_pin_cap: 0.0;
default_inout_pin_cap: 0.1;
default_input_pin_cap: 0.1;
default_max_transition: 1.0;

cell(std_cell_inv) {
    cell_footprint : inv;
    area : 1.0;
    pg_pin(VDD) {
        voltage_name : VDD;
        pg_type : primary_power;
        related_bias_pin : "VNW";
    }
    pg_pin(VSS) {
        voltage_name : VSS;
        pg_type : primary_ground;
        related_bias_pin : "VPW";
    }
    pg_pin(VNW) {
        voltage_name : VNW;
        pg_type : nwell;
        physical_connection : device_layer;
    }
    pg_pin(VPW) {
        voltage_name : VPW;
        pg_type : pwell;
        physical_connection : device_layer;
    }
    pin(A) {
        direction : input;
```

```
    capacitance : 1.0;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    related_bias_pin : "VPW VNW";
}
pin(Y) {
    direction : output;
    function : "A'";
    related_power_pin : VDD;
    related_ground_pin : VSS;
    related_bias_pin : "VPW VNW";
    power_down_function : "!VDD + VSS + VNW' + VPW";
    internal_power() {
        related_pg_pin : VDD;
        related_pin : "A";
        rise_power(scalar) { values ( "1.0"); }
        fall_power(scalar) { values ( "1.0"); }
    }
    timing() {
        related_pin : "A";
        timing_sense : positive_unate;
        cell_rise(scalar) { values ( "0.1"); }
        rise_transition(scalar) { values ( "0.1"); }
        cell_fall(scalar) { values ( "0.1"); }
        fall_transition(scalar) { values ( "0.1"); }
    }
    max_capacitance : 0.1;
}
cell_leakage_power : 1.0;
leakage_power() {
    when :"!A";
    value : 1.5;
}
leakage_power() {
    when :"A";
    value : 0.5;
}
}
```

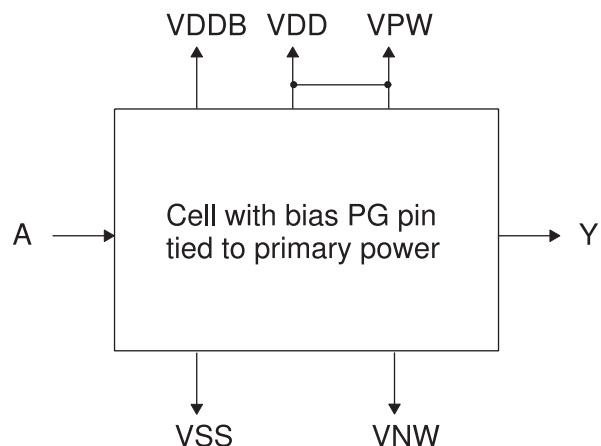
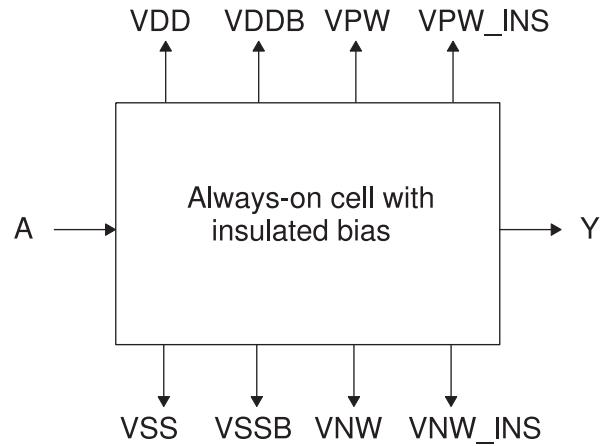
Insulated Bias Modeling

[Example 95](#) is a .lib description of a low-power library with the following two cells:

- An always-on cell with an insulated bias PG pin
- A cell where the bias PG pin is tied to the primary power supply

[Figure 58](#) shows the schematics of these cells.

Figure 58 An Always-On Cell With Insulated Bias and Cell Where the Bias PG pin is Tied to Primary Power



Example 95 A Low-Power Library Using the is_insulated and tied_to Attributes

```
library (mylib) {
    delay_model : table_lookup;

    /* library unit and slew attributes */
    voltage_map(VDD, 1.0);      /* Primary Power */
    voltage_map(VSS, 0.0);      /* Primary Ground */
    voltage_map(VDDB, 1.0);     /* Backup Power */
```

Chapter 9: Advanced Low-Power Modeling
Power and Ground (PG) Pins

```
voltage_map(VSSB, 0.0);      /* Backup Ground */
voltage_map(PW, 1.0);        /* nwell */
voltage_map(VNW, 0.0);       /* pwell */
voltage_map(PW_INS, 1.0);    /* insulated nwell */
voltage_map(VNW_INS, 0.0);   /* insulated pwell */

/* operation conditions */
nom_process      : 1.0;
nom_temperature  : 25;
nom_voltage      : 1.0;
operating_conditions(XYZ) {
    process      : 1.0;
    temperature  : 25;
    voltage      : 1.0;
    tree_type    : balanced_tree
}
default_operating_conditions : XYZ;

/* AO cell with insulated bias pg_pins */
cell(AO_with_insulated_bias) {
    area : 1.0;
    ...
    pg_pin(VDD) {
        voltage_name : VDD;
        pg_type : primary_power;
        related_bias_pin : PW;
    }
    pg_pin(VSS) {
        voltage_name : VSS;
        pg_type : primary_ground;
        related_bias_pin : VNW;
    }
    pg_pin(VDDB) {
        voltage_name : VDDB;
        pg_type : backup_power;
        related_bias_pin : PW_INS;
    }
    pg_pin(VSSB) {
        voltage_name : VSSB;
        pg_type : backup_ground;
        related_bias_pin : VNW_INS;
    }
    pg_pin(PW) {
        voltage_name : PW;
        pg_type : nwell;
        direction : inout;
        physical_connection : device_layer;
    }
    pg_pin(VNW) {
        voltage_name : VNW;
        pg_type : pwell;
        direction : inout;
        physical_connection : device_layer;
    }
}
```

Chapter 9: Advanced Low-Power Modeling
Power and Ground (PG) Pins

```
        }
    pg_pin(VNW_INS) {
        pg_type : pwell;
        voltage_name : VNW_INS;
        is_insulated : true;
        tied_to : VSSB;
        direction : internal;
    }
    pg_pin(VPW_INS) {
        pg_type : nwell;
        voltage_name : VPW_INS;
        is_insulated : true;
        tied_to : VDDB;
        direction : internal;
    }
    pin(A) {
        direction : input;
        related_power_pin : VDDB;
        related_ground_pin : VSSB;
        ...
    }
    pin(Y) {
        direction : output;
        function : "A";
        related_power_pin : VDDB;
        related_ground_pin : VSSB;
        power_down_function : "!VDDB + !VPW_INS + VSSB + VNW_INS";
        ...
    } /* end pin group */
} /* end cell group */

/* cell with bias pg_pin tied to Primary Power */
cell(cell_with_bias_pg_pin_tied_to_power) {
    area : 1.0;
    ...
    pg_pin(VDD) {
        voltage_name : VDD;
        pg_type : primary_power;
        related_bias_pin : VPW;
    }
    pg_pin(VSS) {
        voltage_name : VSS;
        pg_type : primary_ground;
        related_bias_pin : VNW;
    }
    pg_pin(VDDB) {
        voltage_name : VDDB;
        pg_type : backup_power;
        related_bias_pin : VPW_INS;
    }
    pg_pin(VPW) {
        voltage_name : VPW;
        pg_type : nwell;
```

```
    direction : inout;
    tied_to : VDD;
}
pg_pin(VNW) {
    voltage_name : VNW;
    pg_type : pwell;
    direction : inout;
    physical_connection : device_layer;
}
pin(A) {
    direction : input;
    related_power_pin : VDDB;
    related_ground_pin : VSSB;
    ...
}
pin(Y) {
    direction : output;
    function : "A";
    related_power_pin : VDDB;
    related_ground_pin : VSSB;
    power_down_function : "!VDDB + !VPW + VSSB + VNW";
    ...
} /* end pin group */
} /* end cell group */
} /* end library group */
```

Same PG Pin as Both Power Pin and Ground Pin

Typically in a cell, a supply rail provides either power or ground voltage. In the complex power scheme for a macro cell or a pad cell, the same supply might simultaneously act as a power rail for some section and as a ground rail for some other section.

To enable compiling cells containing such complex signal pin, `related_power_pin` and `related_ground_pin` associations, set the `lc_allow_same_pg_pin_as_related_power_and_ground` variable to `true`, as shown:

```
lc_allow_same_pg_pin_as_related_power_and_ground true
```

The default value is `false`.

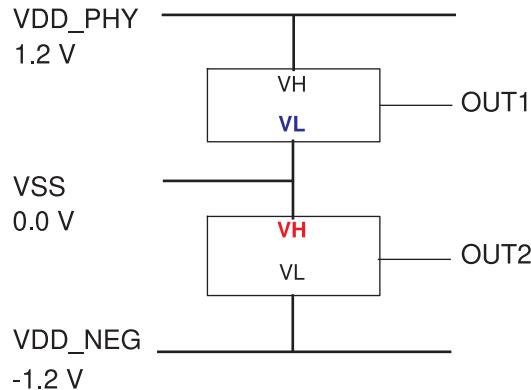
Some recommendations to model such a cell are as follows:

- Specify the PG pin with the highest voltage as a power pin.
- Specify the PG pin with the lowest voltage as a ground pin.
- For rest of the PG pins, specify the ones with voltage above 0.0 V as power pins, and the ones with voltage equal and less than 0.0 V as ground pins.
- The `power_down_function` Boolean expression of the related power pin must have negative logic.

- The `power_down_function` Boolean expression of the related ground pin must have positive logic.

[Figure 59](#) shows the schematic of a cell that uses the pin, VSS with voltage 0.0 V, as both the related power pin and the related ground pin. The subsequent Liberty snippet shows the PG pin model for the cell.

Figure 59 A Cell With Same Pin as Both Power Pin and Ground Pin



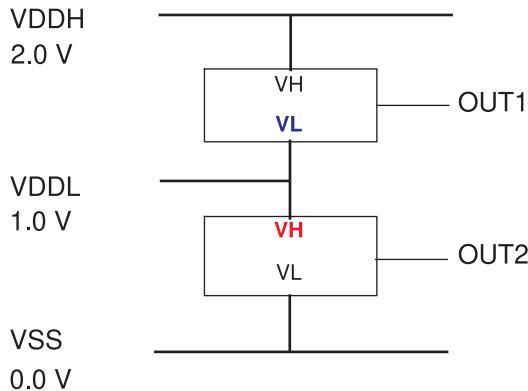
```

cell(same_pin_both_pg1) {
    pg_pin(VDD_PHY) {
        pg_type: primary_power ;
    }
    pg_pin(VSS) {
        pg_type: primary_ground ;
    }
    pg_pins(VDD_NEG) {
        pg_type: primary_ground ;
    }
    pin(OUT1) {
        related_power_pin: "VDD_PHY" ;
        related_ground_pin: "VSS" ;
        power_down_function: "!VDD_PHY + VSS" ;
    }
    pin(OUT2) {
        related_power_pin: "VSS" ;
        related_ground_pin: "VDD_NEG";
        power_down_function: "!VSS + VDD_NEG" ;
    }
}

```

Figure 60 shows the schematic of a cell that uses the pin, VDDL with a positive voltage value, as both the power pin and the ground pin. The following Liberty snippet shows the PG pin model for the cell.

Figure 60 A Cell With a Positive Voltage Pin as Both Power Pin and Ground Pin



```

cell(same_pin_both_pg2) {
    pg_pin(VDDH) {
        pg_type: primary_power ;
    }
    pg_pin(VDDL) {
        pg_type: primary_power ;
    }
    pg_pins(VSS) {
        pg_type: primary_ground ;
    }
    pin(OUT1) {
        related_power_pin: "VDDH" ;
        related_ground_pin: "VDDL" ;
        power_down_function: "!VDDH + VDDL" ;
    }
    pin(OUT2) {
        related_power_pin: "VDDL" ;
        related_ground_pin: "VSS" ;
        power_down_function: "!VDDL + VSS" ;
    }
}
  
```

PG Pin Power Mode Modeling

In complex library cells such as retention and macro cells, power pins can have more power states (or PG modes) apart from the regular ON and OFF states. The following syntax allows you to model such power states of PG pins and the relationship between these states, so that complex power states or modes can be captured.

The syntax enables you to

- Extend power supply state and voltage range definitions
- Define system power states as function of power supplies
- Define transitions between system power states
- Associate a PG mode to each conditional timing, power, and noise group

Syntax

```
library (library_name) {
    voltage_map (voltage_name, voltage_value);

    voltage_state_range_list (voltage_name) {
        voltage_state_range(pg_state, nom_voltage);
        voltage_state_range(pg_state, min_voltage, max_voltage);
        voltage_state_range(pg_state, min_voltage,
                            nom_voltage, max_voltage);
    ...
} /* end voltage_state_range_list group */
cell (cell_name) {
    pg_pin (pg_name) {
        voltage_name : volt_name;
    ...
} /* end pg_pin group */

    pg_setting_definition (psd_name) {

        pg_setting_value(psv_name) {
            pg_pin_active_state (pg_pin, pg_state);
            pg_pin_condition : Boolean expression of PG and signal pins;
            pg_setting_active_state (psd_name2, psv_name2);
            pg_setting_condition : Boolean expression of
                                    psd_name2 and signal pin;
        } /* end pg_setting_value group */

        default_pg_setting : psv_name;

        pg_setting_transition (pst_name) {
            is_illegal : true | false;
            start_setting : psv_name;
            end_setting : psv_name;
        } /* end pg_setting_transition group */

        illegal_transition_if_undefined : true | false;
    } /* end pg_setting_definition group */

    mode_definition (mode_def) {
        mode_value (mode_name) {

```

```
    pg_setting (psd_name, psv_name);
} /* end mode_value group */
} /* end mode_definition group */

pin (pin_name) {

    minimum_period([minimum_period_name]) {
        mode(mode_def_name, mode_name);
    } /* end minimum_period group */
    min_pulse_width([min_pulse_width_name]) {

        mode(mode_def_name, mode_name);
    } /* end min_pulse_width group */
} /* end pin group */
} /* end cell group */
} /* end library group */
```

Library-Level Groups and Attributes

Use the following syntax to model power and ground (PG) modes other than the regular on and off states.

voltage_state_range_list Group

The `voltage_state_range_list` group defines the voltage range of all the states for a specified power rail. It only applies to a power rail with multiple states. The name of the group (`voltage_name`) is a power rail name, which must also be defined in the same library using the `voltage_map` attribute. Each `voltage_name` can have only one corresponding `voltage_state_range_list` group.

voltage_state_range Attribute

The `voltage_state_range` attribute defines the corner-independent operating voltages for the state, `pg_state`, of the power rail. Specify this attribute in the `voltage_state_range_list` group.

Syntax

```
voltage_state_range_list (voltage_name) {
    voltage_state_range(pg_state, nom_voltage);
    voltage_state_range(pg_state, min_voltage, max_voltage);
    voltage_state_range(pg_state, min_voltage, nom_voltage, max_voltage);
```

- `pg_state` is the state name of the power rail
- The value can be a nominal voltage, a pair specifying the minimum and maximum voltages, or a triplet specifying the minimum, nominal and maximum voltages.
 - `min_voltage` and `max_voltage` are floating-point numbers for the minimum and maximum operating voltage of the state.

- `nom_voltage` is a floating-point number and the default operating voltage that applies to all designs. The instance-specific nominal voltage specification at the top-design level overrides the default nominal voltage.
- These values represent the nominal voltage range and do not include the power supply uncertainty or tolerance.
- For each `pg_state` defined in the `voltage_state_range_list` group:
 - `min_voltage` must be less than or equal to `max_voltage` or `nom_voltage`.
 - `nom_voltage` must be less than or equal to `max_voltage`.
 - Different `pg_state` can have overlapped operating voltage range.

Example

```
voltage_map (VDD, 0.8);
voltage_state_range_list(VDD) {
    voltage_state_range(normal, 0.81, 0.9, 0.99);
    voltage_state_range(under_drive, 0.665, 0.77);
    voltage_state_range(on, 0.7);
}
```

Cell-Level Groups and Attributes

Use the following syntax to define the system power states as a function of the power supply states, and the legality of transition between these power states.

pg_setting_definition Group

The `pg_setting_definition` group defines the following:

- The system power states (`pg_setting_value` group).
- The legality of transitions (`pg_setting_transition` group) between the system power states.
- The default power state of the cell.
- The legality of undefined transitions.

`psd_name` is the name of the `pg_setting_definition` group. The system power states must be mutually exclusive.

Syntax

```
pg_setting_definition(psd_name) {
    ...
}
```

psd_name is the name of the `pg_setting_definition` group. The system power states must be mutually exclusive.

Example

```
pg_setting_definition(psd) {  
    ...  
}
```

pg_setting_value Group

The `pg_setting_value` group defines a system power state named *psv_name*. Specify this group in the `pg_setting_definition` group.

- You must define all the legal power states. Undefined power states are considered illegal.
- A legal power state is a state that is expected to occur during normal operation.
- An illegal power state is a state that is not expected to occur during normal operation.

Syntax

```
pg_setting_value (psv_name) {  
    pg_pin_active_state (pg_pin, pg_state);  
    pg_pin_condition : Boolean expression of pg_pin and pin;  
    pg_setting_active_state (psd_name2, psv_name2);  
    pg_setting_condition : Boolean expression of psd_name2 and pin;  
}
```

Example

```
pg_setting_definition(psd) {  
    pg_setting_value (psv1) {  
        ...  
    }  
}
```

The `pg_setting_value` group contains the following attributes.

pg_pin_active_state and pg_pin_condition Attributes

These attributes define a system power state as the function of the PG pins and the signal pins in the `pg_setting_value` group.

The `pg_pin_active_state` complex attribute defines the active power supply state, `pg_state`, for a specified PG pin. The attribute only applies to a multi-state power supply. For a PG pin with a single state, the pin is active when it is on.

The `pg_pin_condition` attribute specifies the logic condition when the system is in the power state named *psv_name*. For a single-state PG pin, this condition evaluates to true when the PG pin is in the on state. For a multi-state PG pin, this condition evaluates to

true whenever the PG pin is in the active state specified by the `pg_pin_active_state` attribute.

Syntax

```
pg_setting_value (psv_name) {
    pg_pin_active_state(pg_pin, pg_state);
    pg_pin_condition : Boolean expression of pg_pin and signal pin;
}
```

Example

```
pg_setting_definition (psd) {
    pg_setting_value (on) {
        pg_pin_active_state(VDD, normal);
        pg_pin_condition : "VDD * !VSS" ;
    }
    pg_setting_value (sleep) {
        pg_pin_active_state(VDD, under_drive);
        pg_pin_condition : "VDD * !VSS * sig_sleep" ;
    }
}
```

pg_setting_active_state and pg_setting_condition Attributes

These attributes define a system power state as a function of other system power states and signal pins in the `pg_setting_value` group. Because they define the system power states hierarchically, you do not need to repeat the complete list of PG pin conditions and states for a large block.

The `pg_setting_active_state` complex attribute specifies the active system power state, `psv_name2`, of another `pg_setting_definition` group (named `psd_name2`) in the system power state, `psv_name`.

The `pg_setting_condition` attribute specifies the logic condition when the system is in the power state, `psv_name`. This condition evaluates to true whenever `psd_name2` is in the active state specified by the `pg_setting_active_state` attribute.

Syntax

```
pg_setting_value (psv_name) {
    pg_setting_active_state (psd_name2, psv_name2);
    pg_setting_condition : Boolean expression of psd_name2 & signal pin;
}
```

Example

```
pg_setting_definition(psd) {
    pg_setting_value (psv1) {...}
    pg_setting_value (psv2) {...}
}
pg_setting_definition(psd2) {
```

```
    pg_setting_value (psv1) { ... }
    pg_setting_value (psv2) { ... }
}
pg_setting_definition(top_psd) {
    pg_setting_value (psv1) {
        pg_setting_active_state(psd, psv1);
        pg_setting_active_state(psd2, psv2);
        pg_setting_condition : "psd * psd2" ;
    }
    pg_setting_value (psv2) {
        pg_setting_active_state(psd, psv2);
        pg_setting_active_state(psd2, psv1);
        pg_setting_condition : "psd * psd2" ;
    }
}
```

default_pg_setting Attribute

The `default_pg_setting` attribute specifies a default power state to match when the explicitly defined power states do not completely cover the state space of the `pg_setting_definition` group. The default power state also acts as the initial power state of the `pg_setting_definition` group.

`psv_name` is the name of a `pg_setting_value` group specified in the same `pg_setting_definition` group.

Syntax

```
pg_setting_definition (psd_name) {
    ...
    default_pg_setting : psv_name ;
}
```

Example

```
pg_setting_definition (psd) {
    default_pg_setting : "psv1" ;
}
```

pg_setting_transition Group

The `pg_setting_transition` group specifies the legal and illegal transitions between the PG modes defined in the `pg_setting_definition` group.

`pst_name` is the name you assign to the transition.

Syntax

```
pg_setting_transition (pst_name) {
    is_illegal : [ true | false ];
    start_setting : psv_name1;
    end_setting : psv_name2;
```

}

pst_name is the name you assign to the transition.

Example

```
pg_setting_transition (sleep) {  
    start_setting : on;  
    end_setting : sleep;  
}
```

The `pg_setting_transition` group contains the following attributes.

is_illegal Attribute

The `is_illegal` attribute specifies if the transition, *pst_name*, is illegal. The default is `false`.

start_setting and end_setting Attributes

The `start_setting` attribute specifies the power state from where the transition, *pst_name*, starts. The `end_setting` attribute specifies the power state where the transition, *pst_name*, ends. Both the power states must be defined in the same `pg_setting_definition` group.

If you do not specify the `start_setting` attribute, it means that all the power states in the same `pg_setting_definition` group can be the start states for the transition, *pst_name*.

If you do not specify the `end_setting` attribute, it means that all the power states in the same `pg_setting_definition` group can be the end states for the transition, *pst_name*.

illegal_transition_if_undefined Attribute

The `illegal_transition_if_undefined` attribute, if set to `true`, means that all the undefined transitions in the `pg_setting_definition` group are illegal or invalid. By default, all undefined transitions are valid.

Syntax

```
pg_setting_definition(psd_name) {  
    illegal_transition_if_undefined : true | false ;  
}
```

Specifying Power States in Timing, Power, and Noise Models

Use the following syntax to make the conditional timing, power, and noise models power-state aware and to address frequency-scaled power states.

pg_setting Attribute in mode_value Group

The `pg_setting` complex attribute specifies the power state of a cell mode by referencing a power state (`psv_name`) defined in the `pg_setting_definition` group of the cell.

You can define multiple `pg_setting` attributes in a `mode_value` group provided each of the `pg_setting` attributes references a power state from a different `pg_setting_definition` group.

Syntax

```
cell (cell_name) {
    mode_definition (mode_def) {
        mode_value (mode_name) {
            pg_setting (psd_name, psv_name);
        }
    }
}
```

mode Attribute in minimum_period and min_pulse_width Groups

The `mode` complex attribute enables conditional data modeling in the frequency-scaled groups, that is, the `minimum_period` and `min_pulse_width` groups.

The `mode` attribute uses the `mode_name` to reference a `mode_value` group defined in the cell. When you specify the `pg_setting` attribute in the referenced `mode_value` group, the frequency-scaled group where the mode is referenced also becomes power-state aware.

Syntax

```
pin (pin_name) {
    minimum_period(minimum_period_name) {
        mode(mode_def_name, mode_name);
    }
    min_pulse_width(min_pulse_width_name) {
        mode(mode_def_name, mode_name);
    }
}
```

Defining PG Modes in Macro Cells

To include the PG mode syntax in your current design flow, remember the following:

- PG mode syntax is corner-independent.

You must define the operating voltage ranges for all the PG states, if multi-state.

To maintain consistency between different corners, you must define all the PG modes for a cell.

- For a power rail with only on and off states, you do not need to specify a voltage range. The PG modes are active in all corners.
- The default nominal voltage of a PG state (*nom_voltage* in the *voltage_state_range* attribute definition) is the default voltage that applies to all the designs of your intellectual property (IP).

PG Mode Examples

The following examples illustrate the use of the PG mode syntax in modeling different cells.

Cell With Different Power States

The following is an example of a cell with multiple power states. The cell has two power pins, VDD and VDDS and one ground pin, VSS. The following table lists the voltage ranges of the PG pins.

PG pin	Voltage range in state1	Voltage range in state 2
VDD	1.08	0.92~1.00 (nominal: 0.96)
VDDS	1.08	X
VSS	0	X

```
library (mylib) {
    voltage_map ("VM1", 1.08);
    voltage_map ("VM2", 1.08);
    voltage_map ("VG", 0);
    voltage_state_range_list(VM1) {
        /* state1 */
        voltage_state_range(HV, 1.08);
        /* state2 */
        voltage_state_range(HV_L, 0.92, 0.96, 1.00);
        /* state not used in cell(mycell) */
        voltage_state_range(HV_H, 0.98, 1.08);
    }
}

cell (mycell) {
    pg_pin (VDD) {
        pg_type : primary_power;
        voltage_name: "VM1";
    }
    pg_pin(VDDS) {
```

```

        pg_type : primary_power;
        voltage_name: "VM2";
    }
    pg_pin(VSS) {
        pg_type : primary_ground;
        voltage_name: "VG";
    }

    pg_setting_definition (pst) {
        pg_setting_value(ps_1) {
            pg_pin_condition : "VDD * VDDS * !VSS";
            pg_pin_active_state(VDD, HV);
        }
        pg_setting_value(ps_2) {
            pg_pin_condition : "VDD * !VDDS * !VSS";
            pg_pin_active_state (VDD, HV_L);
        }
        pg_setting_value(off) {
            pg_pin_condition : "!VDD * !VDDS + VSS";
        }
        default_pg_setting : ps_1 ;
    }
}/* end cell group */
}/* end library group */

```

Power State Transition

The following is an example of a cell with transition between the power states. The cell has two blocks: one powered by VDD, and the other powered by VDDS. The table lists the voltage ranges of the PG pins.

PG pin	Voltage range in state1	Voltage range in state 2
VDD	0.7~0.9	0.6
VDDS	0.8~1.0	X
VSS1	0	X

```

library (mylib) {

    voltage_map("VM1", 0.8);
    voltage_map("VM2", 0.9);
    voltage_map("VG", 0);
    voltage_state_range_list(VM1) {
        voltage_state_range(on, 0.7, 0.9);
        voltage_state_range(pon, 0.6);
    }
    voltage_state_range_list(VM2) {
        voltage_state_range(on, 0.8, 1.0);
    }
}

```

Chapter 9: Advanced Low-Power Modeling
PG Pin Power Mode Modeling

```
/* this state not used in cell(mycell) */
voltage_state_range(pon, 0.6);
}

cell (mycell) {
    pg_pin (VDD) {
        pg_type : primary_power;
        voltage_name: "VM1";
    }
    pg_pin(VDDS) {
        pg_type : primary_power;
        voltage_name: "VM2";
    }
    pg_pin(VSS) {
        pg_type : primary_ground;
        voltage_name: "VG";
    }

/* power state of block 1 */
pg_setting_definition(PD_SSAH) {

    /* state1 */
    pg_setting_value(GO) {
        pg_pin_condition : "VDD * !VSS";
        pg_pin_active_state(VDD, on);
    }
    /* state2 */
    pg_setting_value(SLEEP) {
        pg_pin_condition: "VDD * !VSS * sp_on";
        pg_pin_active_state(VDD, pon);
    }
    /* state off */
    pg_setting_value(OFF) {
        pg_pin_condition: "!VDD + VSS ";
    }
    /* transition */
    pg_setting_transition(turn_sleep) {
        start_setting : GO;
        end_setting : SLEEP;
    }
    illegal_transition_if_undefined : true ;
} /* end of PD_SSAH */

/* power state of block 2 */
pg_setting_definition (PD_SSBH) {
    pg_setting_value (ON) {
        pg_pin_condition : "VDDS * !VSS";
        pg_pin_active_state(VDDS, on);
    }
    pg_setting_value (OFF) {
        pg_pin_condition : "!VDDS + VSS";
    }
}/* end of PD_SSBH */
```

```

/* power state of the cell */
pg_setting_definition (PD) {
    pg_setting_value (S1) {
        pg_setting_condition : "PD_SSAH * PD_SSBH" ;
        pg_setting_active_state(PD_SSAH, GO);
        pg_setting_active_state(PD_SSBH, ON);
    }
    pg_setting_value (S2) {
        pg_setting_condition : "PD_SSAH * PD_SSBH" ;
        pg_setting_active_state(PD_SSAH, SLEEP);
        pg_setting_active_state(PD_SSBH, OFF);
    }
    pg_setting_value (OFF) {
        pg_setting_condition : "PD_SSAH * PD_SSBH" ;
        pg_setting_active_state(PD_SSAH, OFF);
        pg_setting_active_state(PD_SSBH, OFF);
    }
}
}/* end cell group */
}/* end library group */

```

Macro Cell With PG Modes

The following example shows a Liberty model of a macro cell that contains two blocks (block1 and block2). block1 has primary power pin, VDD1, and backup power pin, VDD3. block2 has primary power pin, VDD2.

PG pin	Voltage range in state1	Voltage range in state 2
VDD1	0.81~0.99 (nominal: 0.9)	0.9~1.05 (nominal: 1.0)
VDD2	0.72~0.88	0.9~1.15
VDD3	0.9	X
VSS1/VSS2	0	X

```

library (mylib) {
    voltage_map("VM1", 0.9);
    voltage_map("VM2", 0.8);
    voltage_map("VM3", 0.9);
    voltage_map("VG", 0);
    voltage_state_range_list(VM1) {
        voltage_state_range(v1_l, 0.81, 0.9, 0.99);
        voltage_state_range(v1_h, 0.9, 1.0, 1.05 );
    }
    voltage_state_range_list(VM2) {

```

Chapter 9: Advanced Low-Power Modeling
PG Pin Power Mode Modeling

```
voltage_state_range(v2_l, 0.72, 0.88);
voltage_state_range(v2_h, 0.9, 1.15);
}
cell (my_macro_cell) {
    pg_pin(VDD1) {
        pg_type : primary_power;
        voltage_name : "VM1";
    }
    pg_pin(VDD2) {
        pg_type : primary_power;
        voltage_name : "VM2";
    }
    pg_pin(VDD3) {
        pg_type : backup_power;
        voltage_name : "VM3";
    }
    pg_pin(VSS1) {
        pg_type : primary_ground;
        voltage_name : "VG";
    }
    pg_pin(VSS2) {
        pg_type : primary_ground;
        voltage_name : "VG";
    }
/* Block1 power states: VDD1/VDD3/VSS1 */
pg_setting_definition(P1) {
    pg_setting_value(sleep) {
        pg_pin_condition : "!VDD1 * VDD3 * !VSS1";
    }
    pg_setting_value(normal) {
        pg_pin_condition : "VDD1 * VDD3 * !VSS1";
        pg_pin_active_state(VDD1, v1_l);
    }
    pg_setting_value(power_mode) {
        pg_pin_condition : "VDD1 * VDD3 * !VSS1";
        pg_pin_active_state(VDD1, v1_h);
    }
    pg_setting_value(off) {
        pg_pin_condition : "!VDD1 * !VDD3 + VSS";
    }
    default_pg_setting : sleep ;
    pg_setting_transition(turn_on) {
        end_setting : normal ;
    }
    pg_setting_transition(no_way) {
        is_illegal : true ;
        start_setting : power_mode ;
        end_setting : sleep ;
    }
    pg_setting_transition(power_save) {
        start_setting : normal ;
        end_setting : sleep ;
    }
}
```

```
    illegal_transition_if_undefined : true ;  
}  
  
/* Block2 power states: VDD2/VSS2 */  
pg_setting_definition(P2) {  
    pg_setting_value(low_speed) {  
        pg_pin_condition : "VDD2 * !VSS2";  
        pg_pin_active_state(VDD2, v2_l);  
    }  
    pg_setting_value(high_speed) {  
        pg_pin_condition : "VDD2 * !VSS2";  
        pg_pin_active_state(VDD2, v2_h);  
    }  
    pg_setting_value(off) {  
        pg_pin_condition : "!VDD2 + VSS";  
    }  
    pg_setting_transition(power_mode) {  
        start_setting : low_speed;  
        end_setting : high_speed;  
    }  
    pg_setting_transition(power_save) {  
        start_setting : high_speed;  
        end_setting : low_speed;  
    }  
    default_pg_setting: low_speed ;  
}  
  
/* Cell power states: block1 & block2 */  
pg_setting_definition(TOP) {  
    pg_setting_value(power_save) {  
        pg_setting_condition : "P1 * P2";  
        pg_setting_active_state (P1, sleep);  
        pg_setting_active_state (P2, low_speed);  
    }  
    pg_setting_value(normal_mode) {  
        pg_setting_condition : "P1 * P2";  
        pg_setting_active_state (P1, normal);  
        pg_setting_active_state (P2, low_speed);  
    }  
    pg_setting_value(power_mode) {  
        pg_setting_condition : "P1 * P2";  
        pg_setting_active_state (P1, high_speed);  
        pg_setting_active_state (P2, high_speed);  
    }  
    pg_setting_value(off) {  
        pg_setting_condition : "P1 * P2";  
        pg_setting_active_state (P1, off);  
        pg_setting_active_state (P2, off);  
    }  
    default_pg_setting : normal_mode ;  
}  
  
mode_definition(block1) {
```

```

mutually_exclusive_mode_values : true;
initial_mode : "Idle" ;
mode_value(Idle) {
    when : "!Enable";
    pg_setting(p1, sleep);
}
mode_value(Active) {
    when : "Enable & CS1";
    pg_setting(p1, normal);
}
mode_value(Standby) {
    when : "Enable & !CS1";
    pg_setting(p1, normal);
}
...
} /* end cell group */
} /* end library group */

```

Retention Cell Leakage Power in Different Power Modes

Retention Cell With Two Modes

The cell operates in the normal mode when VDD is on, and in the retention mode when VDD is off. VDD is the primary power supply and VDDG is the always-on (backup) power supply. RETN is the save or restore pin. The data is saved when the pin is high and restored when the pin is low.

Pin	Working voltage (V)	pg_setting_value	
		Normal	Retention
RETN	-	1/0/r/f	0
CK	-	1/0/r/f	X
D	-	1/0	X
VDD	0.63	1	OFF
VDDG	0.63	1	1
VSS/VSSG	0/0	0	0

To model different leakage power values when VDD is on or off, specify the PG pin and signal pin states for the different modes of operation of the cell.

```

library(mylib) {
    voltage_map (VDD, 0.63); /* Switchable Power */
    voltage_map (VDDG, 0.63); /* Backup Power */

```

Chapter 9: Advanced Low-Power Modeling
PG Pin Power Mode Modeling

```
voltage_map (VSS, 0);      /* Primary Ground */
voltage_map (VSSG, 0);    /* Backup Ground */
...
cell (my_retention_cell) {
...
    pg_pin (VDD) {
        pg_type : primary_power;
        voltage_name : "VDD";
    }
    pg_pin (VDDG) {
        pg_type : backup_power;
        voltage_name : "VDDG";
    }
    pg_pin (VSS) {
        pg_type : primary_ground;
        voltage_name : "VSS";
    }
    pg_pin (VSSG) {
        pg_type : backup_ground;
        voltage_name : "VSSG";
    }
}

pg_setting_definition (PD) {
    pg_setting_value (normal) {
        pg_pin_condition : "VDDG & VDD & !VSS & !VSSG";
    }
    pg_setting_value (retention) {
        pg_pin_condition : "VDDG & !VDD & !VSS & !VSSG";
    }
    pg_setting_value (off) {
        pg_pin_condition : "!VDDG & !VDD + VSS + VSSG";
    }
}
mode_definition (power_state){
/* signal: CK/D/RETN could be 1/0 */
    mode_value (active) {
        pg_setting (PD, normal);
    }
/* signal: CK/D are X, RETN is 0 */
    mode_value (retained) {
        when : "!RETN" ;
        pg_setting (PD, retention);
    }
}
/* pin definitions are omitted */
/* only 2 leakage power groups possible in retention mode */
leakage_power () {
    mode (power_state, retained);
    when : "(!RETN)";
    value : 0;
    related_pg_pin : VDD;
}
leakage_power () {
```

```

        mode (power_state, retained);
        when : "(!RETN)";
        value : 1.31337e-06;
        related_pg_pin : VDDG;
    }
/* 16 leakage power groups with different CK/D/RETN signal pin states */
leakage_power () {
    mode (power_state, active);
    value : 6.87551e-06;
    when : "(CK&D&RETN)";
    related_pg_pin : VDD;
}
leakage_power () {
    mode (power_state, active);
    value : 9.57963e-07;
    when : "(CK&D&RETN)";
    related_pg_pin : VDDG;
}
leakage_power () {
    mode (power_state, active);
    value : 7.02861e-06;
    when : "(CK&D&!RETN)";
    related_pg_pin : VDD;
}
leakage_power () {
    mode (power_state, active);
    value : 1.21337e-06;
    when : "(CK&D&!RETN)";
    related_pg_pin : VDDG;
}
.....
}/* end cell group */
}/* end library group */

```

Retention Cell With Three Modes

Compared to the retention cell with two modes, the retention cell in this example has an extra mode to save more power in the deep retention mode. The cell has three modes: `normal`, `retention`, and `retention_low`.

`VDDG` is the always-on (backup) power supply that works at two voltage values:

- A high voltage of 0.63 V (operating range is 0.56 ~ 0.70)
- A low voltage of 0.45 V (operating range is 0.41 ~ 0.49) to save more power

Pin	pg_setting_value	
	Normal	Retention
RETN	1/0/r/f	0

Pin	pg_setting_value	
	Normal	Retention
CK	1/0/r/f	X
D	1/0	X
VDD	1	OFF
VDDG	1	1/1
VSS/VSSG	0	0

To model this retention cell, the following two corner libraries are needed:

- PVT1.lib: VDD = 0.63 V; VDDG = 0.63 V
- PVT2.lib: VDD = 0.58 V; VDDG = 0.45 V

PVT1.lib

```
library(mylib) {
    voltage_map (VDD, 0.63);
    voltage_map (VDDG, 0.63); /* characterization voltage */
    voltage_map (VSS, 0);
    voltage_map (VSSG, 0);
    voltage_state_range_list(VDDG) {
        voltage_state_range(high, 0.56, 0.63, 0.70); /* min/nom/max */
        voltage_state_range(low, 0.41, 0.45, 0.49); /* min/nom/max */
    }
    cell (my_retention_cell) {
        pg_setting_definition (PD) {
            pg_setting_value (normal) {
                pg_pin_condition : " VDDG & VDD & !VSS & !VSSG";
                pg_active_state(VDDG, high);
            }
            pg_setting_value (retention) {
                pg_pin_condition : " VDDG & !VDD & !VSS & !VSSG";
                pg_active_state(VDDG, high);
            }
            pg_setting_value (retention_low) {
                pg_pin_condition : " VDDG & !VDD & !VSS & !VSSG";
                pg_active_state(VDDG, low);
            }
            pg_setting_value (off) {
                pg_pin_condition : " !VDDG & !VDD + VSS + VSSG";
            }
            /* PG mode cannot directly move from
               active to retention_low mode or vice-versa */
            pg_setting_transition(normal_to_low) {

```

```
        is_illegal : true;
        start_setting : normal;
        end_setting : retention_low;
    }
    pg_setting_transition(low_to_normal) {
        is_illegal : true;
        start_setting : retention_low;
        end_setting : normal;
    }
}
mode_definition (power_state){
/* signal: CK/D/RETN could be 1/0 */
    mode_value (active) {
        pg_setting (PD, normal);
    }
/* signal: CK/D are X, RETN is 0 */
    mode_value (retained) {
        when : "!RETN" ;
        pg_setting (PD, retention);
    }
/* signal: CK/D are X, RETN is 0 */
    mode_value (retained_low) {
        when : "!RETN" ;
        pg_setting (PD, retention_low);
    }
}
/* retained mode works in this corner */
leakage_power () {
    mode (power_state, retained);
    when : "(!RETN)";
    value : 0;
    related_pg_pin : VDD;
}
leakage_power () {
    mode (power_state, retained);
    when : "(!RETN)";
    value : 1.31337e-06;
    related_pg_pin : VDDG;
}
/* active mode is same as example 12.1.4.1 */
}/* end cell group */
}/* end library group */
```

PVT2.lib

```
library(mylib) {
    voltage_map (VDD, 0.58);
    voltage_map (VDDG, 0.45); /* characterization voltage */
    voltage_map (VSS, 0);
    voltage_map (VSSG, 0);
    voltage_state_range_list(VDDG) {
        voltage_state_range(high, 0.56, 0.63, 0.70); /* min/nom/max */
        voltage_state_range(low, 0.41, 0.45, 0.49); /* min/nom/max */
    }
}
```

```
}

cell (my_retention_cell) {
    pg_setting_definition (PD) {
        pg_setting_value (normal) {
            pg_pin_condition : " VDDG & VDD & !VSS & !VSSG";
            pg_active_state(VDDG, high);
        }
        pg_setting_value (retention) {
            pg_pin_condition : " VDDG & !VDD & !VSS & !VSSG";
            pg_active_state(VDDG, high);
        }
        pg_setting_value (retention_low) {
            pg_pin_condition : " VDDG & !VDD & !VSS & !VSSG";
            pg_active_state(VDDG, low);
        }
        pg_setting_value (off) {
            pg_pin_condition : "!VDDG & !VDD + VSS + VSSG";
        }
        /* PG mode cannot directly move from
           active to retention_low mode and vice-versa */
        pg_setting_transition(normal_to_low) {
            is_illegal : true;
            start_setting : normal;
            end_setting : retention_low;
        }
        pg_setting_transition(low_to_normal) {
            is_illegal : true;
            start_setting : retention_low;
            end_setting : normal;
        }
    }
    mode_definition (power_state){
        /* signal: CK/D/RETN could be 1/0 */
        mode_value (active) {
            pg_setting (PD, normal);
        }
        /* signal: CK/D are X, RETN is 0 */
        mode_value (retained) {
            when : "!RETN" ;
            pg_setting (PD, retention);
        }
        /* signal: CK/D are X, RETN is 0 */
        mode_value (retained_low) {
            when : "!RETN" ;
            pg_setting (PD, retention_low);
        }
    }
    /* retained_low mode works in this corner */
    leakage_power () {
        mode (power_state, retained_low) ;
        when : "(!RETN)" ;
        value : 0;
        related_pg_pin : VDD;
```

```
        }
leakage_power () {
    mode (power_state, retained_low) ;
    when : "(!RETN)" ;
    value : 1.31337e-07;
    related_pg_pin : VDDG;
}
/* No leakage power group in active mode as VDDG is in low state */
}/* end cell group */
}/* end library group */
```

Feedthrough Signal Pin Modeling

A feedthrough is created when two or more pins of a cell share the same physical or unbuffered logical net. Some feedthroughs have no electrical connections with the cell power pins and do not affect cell behavior.

Feedthrough pins can be of two types:

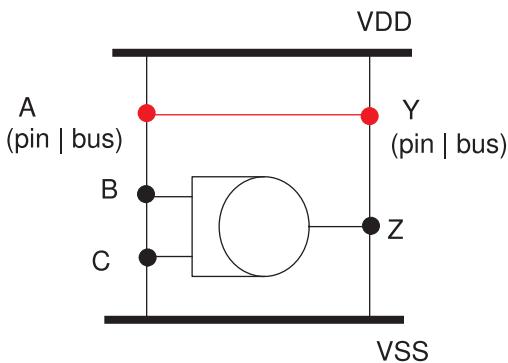
- Multipin, also known as logical feedthroughs or shorted pins
- Single-pin, also known as a pass-through, physical feedthrough, or floating pin

Feedthrough pins can also be part of buses.

Multipin Feedthroughs

[Figure 61](#) is a schematic of a two-pin feedthrough cell with feedthrough pins, A and Y. Pins A and Y appear in all the model views (such as Verilog) as two separate external pins.

Figure 61 A Two-Pin Feedthrough



Multipin Feedthrough Syntax

```
library (library_name) {
```

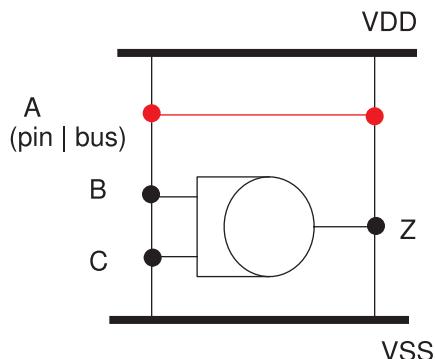
```
cell(cell_name) {
    ...
    short(pin_name, pin_name, ...);
    ...
    pin(pin_name) {
        direction : inout;
        ...
    } /* end pin group */
    ...
} /* end cell group */
} /* end library group */
```

To model multipin feedthroughs as shown in [Figure 61](#), use the `short` attribute in the `cell` group.

Single-Pin Feedthrough

[Figure 62](#) is a schematic of a single-pin feedthrough cell, with feedthrough pin A. In all the model views, only one external pin, bus, or bundle, A, is declared even though the signal can establish one or more physical connections with other cells.

Figure 62 Single-Pin Feedthrough



Single-Pin Feedthrough Syntax

```
library (library_name) {
    cell(cell_name) {
        ...
        pin(single_pin_feedthrough_name) {
            direction : inout;
        } /* end pin group */
        ...
    } /* end cell group */
} /* end library group */
```

To model a single-pin feedthrough, as shown in [Figure 62](#):

- Specify the feedthrough pin as a floating or unused signal pin without parasitics.
- Do not specify any `related_power_pin` and `related_ground_pin` associations for these pins.

[Example 96](#) is a library model of the cell shown in [Figure 62](#).

Example 96 A Single-Pin Feedthrough Model

```
library(mylib) {
    cell (mycell) {
        ...
        pg_pin (VDD) {
            voltage_name : "VDD";
            pg_type : "primary_power";
        }
        pg_pin (VSS) {
            voltage_name : "VSS";
            pg_type : "primary_ground";
        }
        pin (B C) {
            related_power_pin : "VDD";
            related_ground_pin : "VSS";
            direction : "input";
            ...
        }
        pin (A) { /* Do not specify related_power_pin or related_ground_pin */
            /* set pin direction based on actual cell characteristics */
            direction : "inout";
            ...
        }
        pin (Z) {
            related_power_pin : "VDD";
            related_ground_pin : "VSS";
            direction : "output";=
            function : "B * C";
            timing () {
                related_pin : "B C";
                ...
            }
            ...
        }
    } /* end pin group */
} /* end cell group*/
} /* end library group */
```

Overdrive and Underdrive Voltage Modeling

The voltage of an input signal may slightly differ from the supply voltage to the cell.

The optional `max_input_delta_overdrive_high` and `max_input_delta_underdrive_high` attributes specify the allowed overdrive and underdrive delta voltage values of the signal pin with respect to the `related_power_pin` voltage. Together, these attributes establish the legal voltage range of operation for a driver pin.

The voltage values of these two attributes must be either a constant or a linear function of the `related_power_pin` voltage. The attribute value units are taken from the library-level `voltage_unit` attribute.

You can specify this attribute only for input and inout signal pins of macro cells, memory cell, switching cells, level-shifter cells, and pad cells.

Syntax

```
pin(pin_name) {
    direction : input | inout ;
    max_input_delta_overdrive_high : "float" ;
    max_input_delta_underdrive_high : "float" ;
    related_power_pin : pg_pin ;
} /* end pin group */
```

The following example shows a library corner where the overdrive and underdrive voltage values are within both the `input_voltage_range` and `output_voltage_range` values. The input pin, D, of the cell, `mycell`, is allowed an overdrive of 0.3 V and an underdrive of 0.2 V.

```
library (mylib) {
    voltage_map(VDD, 1.0);
    voltage_map(VDD1, 1.2);
    voltage_map(VSS, 0.0);
    input_voltage (OD_and_UD) {
        vil : VSS;
        vih : VDD - 0.2;
        vimin : VSS;
        vimax : VDD + 0.3;
    }
    ...
    cell (mycell) {
        pg_pin (PWR) {
            voltage_name : VDD;
            pg_type : primary_power;
        }
        pg_pin (PWR1) {
            voltage_name : VDD1;
            pg_type : primary_power;
        }
        pg_pin (VSS) {
            voltage_name : VSS;
            pg_type : primary_ground;
        }
    }
}
```

Chapter 9: Advanced Low-Power Modeling Overdrive and Underdrive Voltage Modeling

```
pin (D) {
    direction : input;
    related_power_pin : PWR;
    related_ground_pin : VSS;
    level_shifter_data_pin : true;
    max_input_delta_overdrive_high : 0.3;
    max_input_delta_underdrive_high : 0.2;
    input_voltage : OD_and_UD;
    input_voltage_range (0.5, 1.5);
}
pin (Y) {
    direction : output;
    function : "D";
    related_power_pin : PWR1;
    related_ground_pin : VSS;
    output_voltage_range (0.5, 1.5);
}
} /* end cell */
} /* end library */
```

The following example shows a library corner where the overdrive voltage exceeds the maximum allowable voltage (1.5 V) for pin D. The underdrive voltage is within the `input_voltage_range` and `output_voltage_range` values.

```
library (mylib) {
    voltage_map(VDD, 1.3);
    voltage_map(VDD1, 1.4);
    voltage_map(VSS, 0.0);
    input_voltage (OD_and_UD) {
        vil : VSS;
        vih : VDD - 0.2;
        vimin : VSS;
        vimax : VDD + 0.3;
    }
    ...
    cell (mycell) {
        pg_pin (PWR) {
            voltage_name : VDD;
            pg_type : primary_power;
        }
        pg_pin (PWR1) {
            voltage_name : VDD1;
            pg_type : primary_power;
        }
        pg_pin (VSS) {
            voltage_name : VSS;
            pg_type : primary_ground;
        }
        pin (D) {
            direction : input;
            related_power_pin : PWR;
            related_ground_pin : VSS;
            level_shifter_data_pin : true;
```

```
max_input_delta_overdrive_high : 0.3;
max_input_delta_underdrive_high : 0.2;
input_voltage : OD_and_UD;
input_voltage_range (0.5, 1.5);
}
pin (Y) {
    direction : output;
    related_power_pin : PWR1;
    related_ground_pin : VSS;
    output_voltage_range (0.5, 1.5);
}
/* end cell */
} /* end library */
```

When the values of the `max_input_delta_overdrive_high` or the `max_input_delta_underdrive_high` attribute exceed the `input_voltage_range` values:

- Reduce the allowable overdrive voltage to the high value in the input voltage range.
- Reduce the allowable underdrive voltage to the low value in the input voltage range.

Internally Unconnected Pin Modeling

By default, a pin is considered to be internally not connected when:

- The pin has no related power or related ground pins.
- The pin is not part of a feedthrough path that is marked with the `short` attribute.

To explicitly mark the pin that meets both these conditions as not connected, use the `is_unconnected` attribute in the `pin` group. You can also define this attribute in the `bus` or `bundle` group. Valid value is `true` or `false`.

Note:

It is optional to specify the `is_unconnected` attribute for signal pins of a PG pin library.

Syntax

```
library (library_name) {
    cell(cell_name) {
        ...
        pin (pin_name) {
            is_unconnected : true | false ;
        ...
    }/* End pin group */
    ...
}/* End cell group */
}/* End library group */
```

The following example shows how to mark an internally unconnected pin with the `is_unconnected` attribute.

```
library(short) {
    cell(BLOCK) {
        ...
        pin(y) {
            is_unconnected : true;
        }
    }
}
```

Silicon-on-Insulator (SOI) Cell Modeling

Silicon-on-insulator (SOI) devices are fabricated on a silicon layer that rests upon an insulating layer on the substrate. The presence of the insulating layer makes the drain and source junctions shallow, with small capacitances. Therefore, the SOI devices have better electrical characteristics resulting in improved switching speed and reduced power dissipation.

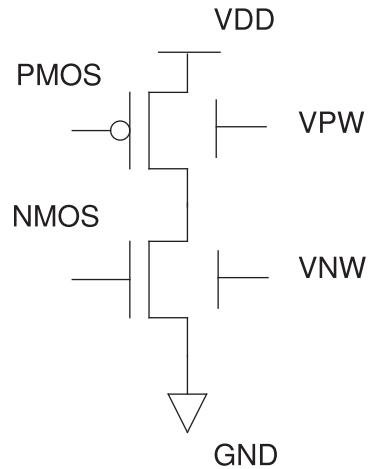
The isolation of the silicon layer from the substrate improves the operating speed, reduces leakage to the substrate, and reduces the number of latch-ups. It is easy to closely pack these inherently insulated structures for high device densities. Further, the SOI fabrication process requires only minor changes to the bulk-CMOS process flow.

Depending on the thickness of the silicon layer, an SOI cell is of two types:

- Fully depleted SOI (FDSOI) cell, where the channel depletion width is greater than the thickness of the silicon layer. Therefore, the channel is fully depleted.
- Partially depleted SOI (PDSOI) cell, where the channel depletion width is less than the thickness of the silicon layer. Therefore, the channel is partially depleted.

[Figure 63](#) shows a typical SOI cell schematic.

Figure 63 A Typical SOI Cell Schematic



The modeling syntax of the SOI cell is:

```
library(library_name) {
    ...
    is_soi: true | false;
    ...
    cell(cell_name) {
        ...
        is_soi: true | false;
        ...
    } /* End cell group */
} /* End Library group */
```

[Table 27](#) shows the differences in modeling substrate-bias pins between a bulk-CMOS and the SOI cell.

Table 27 Differences in Substrate-Bias Pin Modeling Between Bulk-CMOS and SOI Cell

Bulk-CMOS	SOI
For a substrate-bias pin associated with a primary power pin, the <code>pg_type</code> attribute must be <code>nwell</code> .	For a substrate-bias pin associated with a primary power pin, the <code>pg_type</code> attribute can be <code>pwell</code> or <code>nwell</code> .
For a substrate-bias pin associated with a primary ground pin, the <code>pg_type</code> attribute must be <code>pwell</code> .	For a substrate-bias pin associated with a primary ground pin, the <code>pg_type</code> attribute can be <code>pwell</code> or <code>nwell</code> .

[Example 97](#) shows a typical SOI library and cell model.

Example 97 An SOI Library and Cell Description

```
library(SOI) {
    delay_model : table_lookup;
    /* unit attributes */

    ...
    is_soi : true;
    ...
    /* operation conditions */
    ...
    /* threshold definitions */
    ...
    /* default attributes */
    ...
    cell(std_cell_inv) {
        is_soi : true;
        cell_footprint : inv;
        area : 1.0;
        pg_pin(vdd) {
            voltage_name : vdd;
            pg_type : primary_power;
            related_bias_pin : "vpw";
        }
    }
}
```

Library-Level Attribute

This section describes a library-level attribute of SOI cells.

is_soi Attribute

The `is_soi` attribute specifies that all the cells in a library are SOI cells. The default is `false`, which means that the library cells are bulk-CMOS cells.

If the `is_soi` attribute is specified at both the library and cell levels, the cell-level value overrides the library-level value.

Cell-Level Attribute

This section describes a cell-level attribute of SOI cells.

is_soi Attribute

The `is_soi` attribute specifies that the cell is an SOI cell. The default is `false`, which means that the cell is a bulk-CMOS cell.

Level-Shifter Cells in a Multivoltage Design

Level-shifter cells (or buffer-type level-shifter cells) and isolation cells are used to connect the netlist in different power domains, to meet design constraints. In multivoltage and shutdown designs, both level shifting and isolation are required.

Implementation tools need the following information about level-shifter cells from the cell library:

- Which power and ground pin of the level-shifter cell is used for voltage boundary hookup during its insertion. This information allows the optimization tools to determine on which side of the voltage boundary a particular level-shifter cell is allowed.
 - Which voltage conversions the particular level-shifter cell can handle. Specifically, does the level-shifter cell work for conversion from high voltage to low voltage (HL), from low voltage to high voltage (LH), or both (HL_LH)?
 - What the input and output voltage ranges are for a level-shifter cell under all operating conditions.
-

Operating Voltages

In a multivoltage design, each design instance can operate at its specified operating voltage. Therefore, each voltage must correspond to one or more logical hierarchies. All cells in a hierarchy operate at the same voltage except for level shifters.

The operating voltages are annotated on the top design, design instances, or hierarchical ports through PVT operating conditions.

Level Shifter Functionality

A level shifter functions like a buffer, except that the input pin and output pin voltages are different. These cells are necessary in a multivoltage design because the nets connecting pins at two different operating voltages can cause a design violation. Level shifters provide these nets with the needed voltage adjustments.

A level shifter has two components:

- Two power supplies
- Specified input and output voltages

The functionality of level shifters includes

- Identifying nets in the design that need voltage adjustments
- Analyzing the target library for the availability of level shifters

- Ripping the net and instantiating level shifters where appropriate

Basic Level-Shifter Syntax

The basic syntax for level-shifter cells is as follows:

```
cell(level_shifter) {
    is_level_shifter : true ;
    level_shifter_type : HL | LH | HL_LH ;
    input_voltage_range ("float, float");
    output_voltage_range ("float, float");
    ...
    pg_pin(pg_pin_name_P) {
        pg_type : primary_power;
        std_cell_main_rail : true;
        ...
    }
    pg_pin(pg_pin_name_G) {
        pg_type : primary_ground;
        ...
    }
    pin (data) {
        direction : input;
        input_signal_level : "voltage_rail_name";
        input_voltage_range ("float , float");
        level_shifter_data_pin : true ;
        ...
    }/* End pin group */

    pin (enable) {
        direction : input;
        input_voltage_range ("float , float");
        level_shifter_enable_pin : true ;
        ...
    }/* End pin group */

    pin (output) {
        direction : output;
        output_voltage_range ("float , float");
        power_down_function : (!pg_pin_name_P + pg_pin_name_G);
        ...
    }/* End pin group */

    ...
}/* End Cell group */
```

Cell-Level Attributes

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

is_level_shifter Attribute

The `is_level_shifter` simple attribute identifies a cell as a level shifter. The valid values of this attribute are `true` or `false`. If not specified, the default is `false`, meaning that the cell is a standard cell.

level_shifter_type Attribute

The `level_shifter_type` complex attribute specifies the supported voltage conversion type. The valid values are

LH

Low to high

HL

High to low

HL_LH

High to low and low to high

The `level_shifter_type` attribute is optional. The default is `HL_LH`.

input_voltage_range Attribute

The `input_voltage_range` attribute specifies the allowed voltage range of the level-shifter input pin and the voltage range for all input pins of the cell under all possible operating conditions (defined across multiple libraries). The attribute defines two floating values: the first is the lower bound, and the second is the upper bound.

The `input_voltage_range` syntax differs from the pin-level `input_signal_level_low` and `input_signal_level_high` syntax in the following ways:

- The `input_signal_level_low` and `input_signal_level_high` attributes are defined on the input pins under one operating condition (the default operating condition of the library).
- The `input_signal_level_low` and `input_signal_level_high` attributes specify the partial voltage swing of an input pin. Use these attributes to specify partial swings rather than the full rail-to-rail swing. The `input_voltage_range` attribute is not related to the voltage swing.

Note:

The `input_voltage_range` and `output_voltage_range` attributes must be defined together

output_voltage_range Attribute

The `output_voltage_range` attribute is similar to the `input_voltage_range` attribute except that it specifies the allowed voltage range of the level shifter for the output pin instead of the input pin.

The `output_voltage_range` syntax differs from the pin-level `output_signal_level_low` and `output_signal_level_high` syntax in the following ways:

- The `output_signal_level_low` and `output_signal_level_high` attributes are defined on the output pins under one operating condition (the default operating condition of the library).
- The `output_signal_level_low` and `output_signal_level_high` attributes specify the partial voltage swing of an output pin. Use these attributes to specify partial swings rather than the full rail-to-rail swing. The `output_voltage_range` attribute is not related to the voltage swing.

Note:

The `input_voltage_range` and `output_voltage_range` attributes must be defined together.

ground_input_voltage_range Attribute

The `ground_input_voltage_range` attribute specifies the ground voltage range of all the input pins of a level-shifter cell, under all possible operating conditions.

ground_output_voltage_range Attribute

The `ground_output_voltage_range` attribute specifies the ground voltage range of all the output pins of a level-shifter cell, under all possible operating conditions.

Note:

To specify the ground voltage range using the `ground_input_voltage_range` or `ground_output_voltage_range` attributes, define a lower bound and an upper bound of the ground voltage. Specify the lower bound before the upper bound and ensure that the lower bound is less than or equal to the upper bound.

Pin-Level Attributes

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

std_cell_main_rail Attribute

The `std_cell_main_rail` attribute is defined in a `primary_power` power pin. When the attribute is set to `true`, the `pg_pin` is used to determine which power pin is the main rail in the cell.

level_shifter_data_pin Attribute

The `level_shifter_data_pin` attribute identifies a data pin of a level-shifter cell. The default is `false`, meaning that the pin is a regular signal pin.

level_shifter_enable_pin Attribute

The `level_shifter_enable_pin` attribute identifies an enable pin of a level-shifter cell. The default is `false`, meaning that the pin is a regular signal pin.

input_voltage_range and output_voltage_range Attributes

The `input_voltage_range` and `output_voltage_range` attributes specify the allowed voltage ranges of the input or an output pin of the level-shifter cell. The attributes define two floating values where the first value is the lower bound and the second value is the upper bound.

Note:

The pin-level attribute specifications always override the cell-level specifications.

ground_input_voltage_range Attribute

The `ground_input_voltage_range` attribute specifies the ground voltage range of an input pin of a level-shifter cell.

ground_output_voltage_range Attribute

The `ground_output_voltage_range` attribute specifies the ground voltage range of an output pin of a level-shifter cell.

Note:

To specify the ground voltage range using the `ground_input_voltage_range` or `ground_output_voltage_range` attributes, define a lower bound and an upper bound of the ground voltage. Specify the lower bound before the upper bound and ensure that the lower bound is less than or equal to the upper bound.

input_signal_level Attribute

The `input_signal_level` attribute is defined at the pin level and is used to specify which signal is driving the input pin. The attribute defines special overdrive cells that do not have a physical relationship with the power and ground on input pins.

If the `input_signal_level`, `related_power_pin`, and `related_ground_pin` attributes are defined on any input pin, the full voltage swing derived from the `input_signal_level` attribute takes precedence over the voltage swing derived from the `related_power_pin` and `related_ground_pin` attributes during timing calculations.

power_down_function Attribute

The `power_down_function` attribute identifies the Boolean condition under which the cell's signal output pin is switched off (when the cell is in the off mode due to the external power pin states).

alive_during_power_up Attribute

The optional `alive_during_power_up` attribute specifies an active data pin that is powered by a more always-on supply. The default is `false`. If set to `true`, it indicates that the data pin is active while its related power pin is active.

You can specify this attribute only in a `pin` group where the `isolation_cell_data_pin` or the `level_shifter_data_pin` attribute is set to `true`.

Enable Level-Shifter Cell

An enable level-shifter cell generates a voltage shift between the input and output. An additional enable input controls the output.

Differential Level-Shifter Cell

A differential level-shifter cell has a pair of complementary data input pins to generate a voltage difference or shift between the input and output levels, with only a single supply voltage. This voltage shift is significantly greater than generated by other level-shifter cells.

The syntax for modeling the differential level-shifter cell is as follows:

```
cell(cell_name) {
    is_level_shifter : true ;
    pin_opposite ("pin_name", "pin_name");
    contention_condition : "boolean_expression";
    ...
    pin (pin_name) {
        direction : input;
        level_shifter_data_pin : true ;
        input_signal_level : voltage_rail_name;
        ...
    }
}
```

```
 }/* End pin group */
pin (pin_name) {
    direction : input;
    level_shifter_data_pin : true ;
    input_signal_level : voltage_rail_name;
    timing()/* optional */
        related_pin : "pin_name";
        timing_type : non_seq_setup_rising | non_seq_setup_falling |
            non_seq_hold_rising | non_seq_hold_falling;
        rise_constraint(template_name) {
            index_1 ("float, ..., float");
            index_2 ("float, ..., float");
            values("float, ..., float", ..., "float, ..., float");
        }
        fall_constraint(template_name) {
            index_1 ("float, ..., float");
            index_2 ("float, ..., float");
            values("float, ..., float", ..., "float, ..., float");
        }
    }
...
}/* End pin group */
pin (pin_name) {
    direction : input;
    level_shifter_enable_pin : true ;
    ...
}/* End pin group */
pin (pin_name) {
    direction : output;
    function : "boolean_expression";
    ...
}/* End pin group */
...
pg_pin (pg_pin_name) {
    pg_type : primary_power;
    ...
}/* End pg_pin group */
pg_pin (pg_pin_name) {
    pg_type : primary_ground;
    ...
}/* End pg_pin group */
}/* End Cell group */
```

Note:

This syntax is applicable to all types of differential level-shifter cells including low-to-high and high-to-low differential level-shifter cells, and differential level-shifter cells with or without enable inputs.

Cell-Level Attributes

This section describes a cell-level attribute for differential level-shifter cells.

is_differential_level_shifter Attribute

The `is_differential_level_shifter` attribute identifies a level-shifter cell as a differential level-shifter cell. The `is_differential_level_shifter` attribute is automatically set on a level-shifter cell that has pins with the `pin_opposite` and `contention_condition` attributes.

Pin-Level Attributes

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

pin_opposite Attribute

The `pin_opposite` attribute specifies the logical inverse pin groups of a differential level-shifter cell. All the input pins used in the `pin_opposite` attribute must be data pins with the `level_shifter_data_pin` attribute set to `true`.

contention_condition Attribute

The `contention_condition` attribute specifies the Boolean expression of the invalid condition for a differential level-shifter cell. All the input pins used in the `contention_condition` attribute must be data pins with the `level_shifter_data_pin` attribute set to `true`.

Note:

A cell that has pins with the `pin_opposite` and `contention_condition` attributes is identified as a differential level-shifter cell and the `is_differential_level_shifter` and `dont_use` attributes are automatically set on this cell.

Asynchronous Timing Constraints

Use the `timing` group to specify the nonsequential setup and hold constraints between the two complementary input signals on the arrival of data. To set the asynchronous setup and hold constraints at the input pins, model the timing arcs with the following values of the `timing_type` attribute:

- `non_seq_setup_rising`: Checks the setup constraint of the rising edge of the related pin to the signal pin.
- `non_seq_setup_falling`: Checks the setup constraint of the falling edge of the related pin to the signal pin.
- `non_seq_hold_rising`: Checks the hold constraint for the rising edge of the related pin to the signal pin.
- `non_seq_hold_falling`: Checks the hold constraint for the falling edge of the related pin to the signal pin.

Note:

The related pin is an input pin without the `level_shifter_enable_pin` attribute.

Clamping Enable Level-Shifter Cell Outputs

In enable level-shifter (ELS) cells with multiple enable pins, clamping the outputs might be required to maintain them at particular logic levels, such as 0, 1, z, or a previously-latched value.

The clamp function modeling syntax for an enable level-shifter cell and its pins is as follows:

```
cell (cell_name) {
    is_level_shifter : true;
    ...
    pin(input_pin) {
        direction : input;
        level_shifter_data_pin : true;
        ...
    }
    pin(input_pin) {
        direction : input;
        level_shifter_enable_pin : true;
        ...
    }
    pin(input_pin) {
        direction : input;
        level_shifter_enable_pin : true;
        ...
    }
    pin(output_pin_name) {
        direction : output;
        clamp_0_function : "Boolean expression" ;
        clamp_1_function : "Boolean expression" ;
        clamp_z_function : "Boolean expression" ;
        clamp_latch_function : "Boolean expression" ;
        illegal_clamp_condition : "Boolean expression" ;
        ...
    }
    ...
}
```

Pin-Level Attributes

This section describes the pin-level clamp function attributes to clamp the output pins of an enable level-shifter cell.

clamp_0_function Attribute

The `clamp_0_function` attribute specifies the input condition for the enable pins of an enable level-shifter cell when the output clamps to 0.

clamp_1_function Attribute

The `clamp_1_function` attribute specifies the input condition for the enable pins of an enable level-shifter cell when the output clamps to 1.

clamp_z_function Attribute

The `clamp_z_function` attribute specifies the input condition for the enable pins of an enable level-shifter cell when the output clamps to z.

clamp_latch_function Attribute

The `clamp_latch_function` attribute specifies the input condition for the enable pins of an enable level-shifter cell when the output clamps to the previously latched value.

illegal_clamp_condition Attribute

The `illegal_clamp_condition` attribute specifies the invalid condition for the enable pins of an enable level-shifter cell. If the `illegal_clamp_condition` attribute is not specified, the invalid condition does not exist.

Note:

All the input pins used in the Boolean expressions of the `clamp_0_function`, `clamp_1_function`, `clamp_z_function`, and `clamp_latch_function` attributes must be enable pins with the `level_shifter_enable_pin` attribute set to `true`. The Boolean expressions of the `clamp_0_function`, `clamp_1_function`, `clamp_z_function`, `clamp_latch_function`, and `illegal_clamp_condition` attributes must be mutually exclusive.

Level Shifter Modeling Examples

The following sections provide examples to model various types of level-shifter cells.

Level-Shifter Cell Without Enable Pin

A single-bit level-shifter cell without an enable pin has one data input and one output pin. An N-bit multibit level-shifter cell without enable pins, has N data input and N output pins.

The output pin Liberty model of such cells must have a clamp function attribute. During compilation, the tool automatically sets the cell-level `is_no_enable` attribute to `true` for such cells.

The following example shows the Liberty model of a level-shifter cell without any enable pin. D and Z are the input and output pins, respectively. VDD is the related power pin of

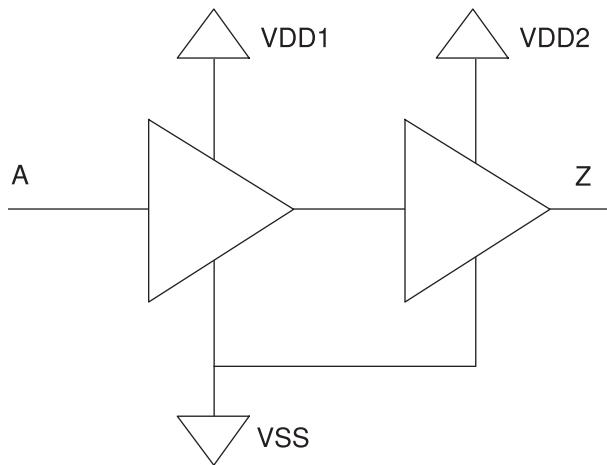
the input, and VDDO is the related power pin of the output. The output clamps to latch condition when the input circuit is powered off and the output circuit remains powered on.

```
library (mylib) {
...
  voltage_map(VDD, 0.80);
  voltage_map(VDDO, 1.20);
...
  cell(No_Enable_LS) {
    is_level_shifter : true;
    level_shifter_type : HL_LH;
    pg_pin(VDD) {
      voltage_name : VDD;
      pg_type : primary_power;
    }
    pg_pin(VDDO) {
      voltage_name : VDDO;
      pg_type : primary_power;
    }
    pg_pin(VSS) {
      voltage_name : VSS;
      pg_type : primary_ground;
    }
    pin(D) {
      direction : input;
      related_power_pin : VDD;
      related_ground_pin : VSS;
      input_voltage_range (0.7, 0.9);
      level_shifter_data_pin : true;
    }
    pin(Z) {
      direction : output;
      related_power_pin : VDDO;
      related_ground_pin : VSS;
      function : "D";
      clamp_latch_function : "!VDD";
      power_down_function : "!VDDO + VSS";
      timing() {
        related_pin : "D";
        ...
      } /* end timing group */
    }/* end pin group*/
  }/*end cell group*/
}/*end library group*/
```

Simple Buffer Type Low-to-High Level Shifter

Figure 64 shows a simple buffer type low-to-high level-shifter cell modeled using the power and ground pin syntax and level-shifter attributes. The figure is followed by an example.

Figure 64 Buffer Type Low-to-High Level-Shifter Cell



```
library(level_shifter_cell_library_example) {
    voltage_map(VDD1, 0.8);
    voltage_map(VDD2, 1.2);
    voltage_map(VSS, 0.0);
    operating_conditions(XYZ) {
        process : 1.0;
        voltage : 3.0;
        temperature : 25.0;
    }
    default_operating_conditions : XYZ;

    cell(Buffer_Type_LH_Level_shifter) {
        is_level_shifter : true;
        level_shifter_type : LH ;

        pg_pin(VDD1) {
            voltage_name : VDD1;
            pg_type : primary_power;
            std_cell_main_rail : true;
        }
        pg_pin(VDD2) {
            voltage_name : VDD2;
            pg_type : primary_power;
        }
        pg_pin(VSS) {
            voltage_name : VSS;
            pg_type : primary_ground;
        }

        leakage_power() {
            when : "!A";
            value : 1.5;
        }
    }
}
```

Chapter 9: Advanced Low-Power Modeling Level-Shifter Cells in a Multivoltage Design

```
    related_pg_pin : VDD1;
}
leakage_power() {
    when : "!A";
    value : 2.7;
    related_pg_pin : VDD2;
}

pin(A) {
    direction : input;
    related_power_pin : VDD1;
    related_ground_pin : VSS;
    input_voltage_range ( 0.7 , 0.9);
}

pin(Z) {
    direction : output;
    related_power_pin : VDD2;
    related_ground_pin : VSS;
    function : "A";
    power_down_function : "!VDD1 + !VDD2 + VSS";
    output_voltage_range (1.1 , 1.3);

    timing() {
        related_pin : A;
        cell_rise(template) {
            ...
        }
        cell_fall(template) {
            ...
        }
        rise_transition(template) {
            ...
        }
        fall_transition(template) {
            ...
        }
    }

    internal_power() {
        related_pin : A;
        related_pg_pin : VDD1;
        ...
    }
    internal_power() {
        related_pin : A;
        related_pg_pin : VDD2;
        ...
    }
}

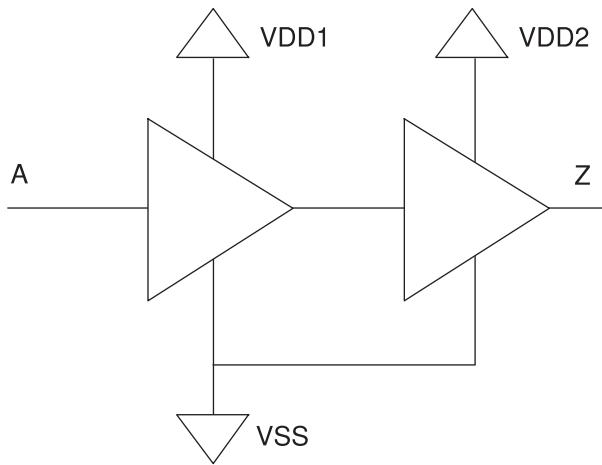
} /* end pin group*/
...
} /*end cell group*/
```

```
    ...
} /*end library group*/
```

Simple Buffer Type High-to-Low Level Shifter

Figure 65 shows a simple buffer type high-to-low level-shifter cell. The cell is modeled using the power and ground pin syntax and level-shifter attributes. Shifting the signal level from high to low voltage can be useful for timing accuracy. When you do this, the level-shifter cell receives a higher voltage signal as its input, which is characterized in the delay tables of the cell description.

Figure 65 Buffer Type High-to-Low Level-Shifter Cell



```
library(level_shifter_cell_library_example) {
  voltage_map(VDD1, 1.2);
  voltage_map(VDD2, 0.8);
  voltage_map(VSS, 0.0);
  operating_conditions(XYZ) {
    process : 1.0;
    voltage : 3.0;
    temperature : 25.0;
  }
  default_operating_conditions : XYZ;

  cell(Buffer_Type_HL_Level_shifter) {
    is_level_shifter : true;
    level_shifter_type : HL ;

    pg_pin(VDD1) {
      voltage_name : VDD1;
      pg_type : primary_power;
      std_cell_main_rail : true;
    }
  }
}
```

Chapter 9: Advanced Low-Power Modeling Level-Shifter Cells in a Multivoltage Design

```
pg_pin(VDD2) {
    voltage_name : VDD2;
    pg_type : primary_power;
}
pg_pin(VSS) {
    voltage_name : VSS;
    pg_type : primary_ground;
}

leakage_power() {
    when : "!A";
    value : 1.5;
    related_pg_pin : VDD1;
}
leakage_power() {
    when : "!A";
    value : 2.7;
    related_pg_pin : VDD2;
}

pin(A) {
    direction : input;
    related_power_pin : VDD1;
    related_ground_pin : VSS;
    input_voltage_range ( 0.7 , 0.9);
}

pin(Z) {
    direction : output;
    related_power_pin : VDD2;
    related_ground_pin : VSS;
    function : "A";
    power_down_function : "!VDD1 + !VDD2 + VSS";
    output_voltage_range (1.1 , 1.3);

    timing() {
        related_pin : A;
        cell_rise(template) {
            ...
        }
        cell_fall(template) {
            ...
        }
        rise_transition(template) {
            ...
        }
        fall_transition(template) {
            ...
        }
    }
}

internal_power() {
    related_pin : A;
```

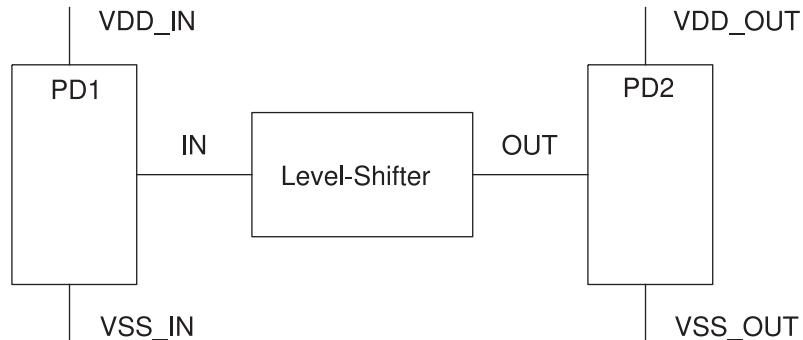
```
related_pg_pin : VDD1;
...
}
internal_power() {
    related_pin : A;
    related_pg_pin : VDD2;
...
}

} /* end pin group*/
...
} /*end cell group*/
...
} /*end library group*/
```

Power-and-Ground Level-Shifter Cell

Figure 66 shows the block diagram of a power-and-ground level-shifter cell. The cell connects the PD1 and PD2 power domains. PD1 and PD2 have power supply voltages, VDD_IN and VDD_OUT, and ground voltages, VSS_IN and VSS_OUT, respectively.

Figure 66 A Simple Power-and-Ground Level-Shifter Cell



Level-shifter cell insertion is supported for the following conditions:

- $VDD_IN \geq VDD_OUT; VSS_IN \geq VSS_OUT$
- $VDD_IN \geq VDD_OUT; VSS_IN \leq VSS_OUT$
- $VDD_IN \leq VDD_OUT; VSS_IN \geq VSS_OUT$
- $VDD_IN \leq VDD_OUT; VSS_IN \leq VSS_OUT$

Example 98 shows the Liberty model of a simple power-and-ground level-shifter cell.

Example 98 Library Model of a Power-and-Ground Level-Shifter Cell

```
library(mylib) {
```

Chapter 9: Advanced Low-Power Modeling Level-Shifter Cells in a Multivoltage Design

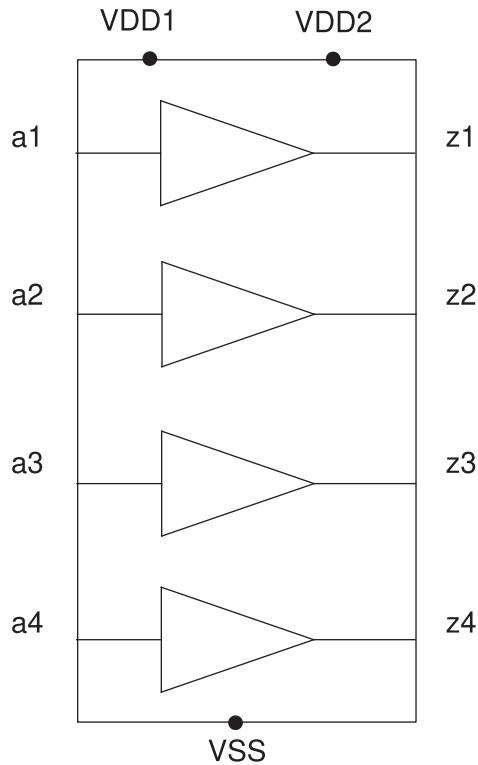
```
delay_model : table_lookup ;
...
voltage_map(VDD_IN, 0.8) ; /* primary power */
voltage_map(VDD_OUT, 1.2) ; /* primary power */
voltage_map(VSS_IN, 0.5) ; /* primary ground */
voltage_map(VSS_OUT, 0.0) ; /* primary ground */
/* operation conditions */
operating_conditions(XYZ) {
    process      : 1 ;
    temperature : 125 ;
    voltage     : 0.8 ;
    tree_type   : balanced_tree ;
}
default_operating_conditions : XYZ;
...
cell(up_shift ){
    area : 1.0 ;
    is_level_shifter : true ;
    level_shifter_type : LH ;
    pg_pin(VDD_IN) {
        voltage_name : VDD_IN ;
        pg_type : primary_power ;
        std_cell_main_rail : true ;
    }
    pg_pin(VDD_OUT) {
        voltage_name : VDD_OUT ;
        pg_type : primary_power ;
    }
    pg_pin(VSS_IN) {
        voltage_name : VSS_IN ;
        pg_type : primary_ground ;
        std_cell_main_rail : true ;
    }
    pg_pin(VSS_OUT) {
        voltage_name : VSS_OUT ;
        pg_type : primary_ground ;
    }
    pin(IN) {
        direction : input ;
        capacitance : 1.0 ;
        related_power_pin : VDD_IN ;
        related_ground_pin : VSS_IN ;
        input_voltage_range (0.8, 1.0) ;
        ground_input_voltage_range (0.4, 0.5) ;
    }
    pin(OUT) {
        direction : output ;
        related_power_pin : VDD_OUT ;
        related_ground_pin : VSS_OUT ;
        function : "IN" ;
        power_down_function : "!VDD_IN + !VDD_OUT + \
                           VSS_IN + VSS_OUT" ;
        output_voltage_range (1.0, 1.2) ;
    }
}
```

```
ground_output_voltage_range (0.0 , 0.1) ;
timing() {
    related_pin : IN ;
    cell_rise(scalar) {
        values ("1.0") ;
    }
    rise_transition (scalar) {
        values ("1.0") ;
    }
    cell_fall(scalar) {
        values ("1.0") ;
    }
    fall_transition (scalar) {
        values ("1.0") ;
    }
}
}
} /* end cell group */
} /* end library group */
```

Multibit Level-Shifter Cell

[Figure 67](#) shows a multibit buffer-type level-shifter cell with four data input and four data output pins. Each of the outputs is a linear function of the respective input. The input data pins are powered by the PG pin, VDD1, and the output data pins are powered by the PG pin, VDD2. The example following the figure is a model of the level-shifter cell.

Figure 67 4-bit Level-Shifter Cell



```
cell (4_bit_ls) {
    is_level_shifter : true;
    pg_pin (VDD1) {
        pg_type : primary_power;
        voltage_name : VDD1;
    }
    pg_pin (VDD2) {
        pg_type : primary_power;
        voltage_name : VDD2;
    }
    pg_pin (VSS) {
        pg_type : primary_ground;
        voltage_name : VSS;
    }
    ...
    pin (a1) {
        level_shifter_data_pin : true;
        related_power_pin : VDD1;
        related_ground_pin : VSS;
        ...
    }
    pin (a2) {
```

Chapter 9: Advanced Low-Power Modeling Level-Shifter Cells in a Multivoltage Design

```
level_shifter_data_pin : true;
related_power_pin : VDD1;
related_ground_pin : VSS;
...
}
pin (a3) {
    level_shifter_data_pin : true;
    related_power_pin : VDD1;
    related_ground_pin : VSS;
    ...
}
pin (a4) {
    level_shifter_data_pin : true;
    related_power_pin : VDD1;
    related_ground_pin : VSS;
    ...
}
pin (z1) {
    direction : output;
    related_power_pin : VDD2;
    related_ground_pin : VSS;
    power_down_function : !VDD1+!VDD2+VSS;
    function : a1;
    ...
}
pin (z2) {
    direction : output;
    related_power_pin : VDD2;
    related_ground_pin : VSS;
    power_down_function : !VDD1+!VDD2+VSS;
    function : a2;
    ...
}
pin (z3) {
    direction : output;
    related_power_pin : VDD2;
    related_ground_pin : VSS;
    power_down_function : !VDD1+!VDD2+VSS;
    function : a3;
    ...
}
pin (z4) {
    direction : output;
    related_power_pin : VDD2;
    related_ground_pin : VSS;
    power_down_function : !VDD1+!VDD2+VSS;
    function : a4;
}
} /* end cell */
```

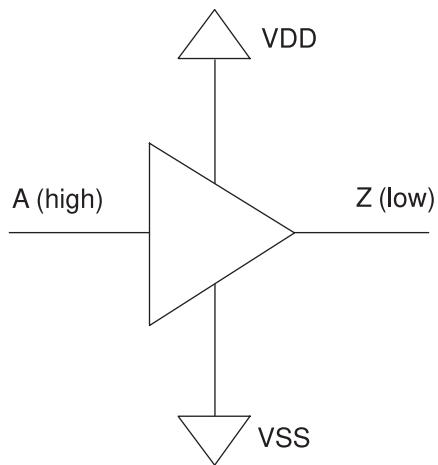
Overdrive Level-Shifter Cell

The cell in [Figure 68](#) is known as an overdrive level-shifter cell. To model an overdrive level-shifter cell, specify the `related_ground_pin` attribute and the `input_signal_level` attribute, as shown in the example that follows the figure.

Note:

The area of a multiple-rail level-shifter cell (as shown in the figure in [Simple Buffer Type High-to-Low Level Shifter](#)) is larger than that of an overdrive level-shifter cell (as shown in [Figure 68](#)). Keep the area in mind when you design your cell.

Figure 68 Overdrive Level-Shifter Cell



```
library(level_shifter_cell_library_example) {
    voltage_map(VDD1, 1.2);
    voltage_map(VDD, 0.8);
    voltage_map(VSS, 0.0);
    operating_conditions(XYZ) {
        process : 1.0;
        voltage : 3.0;
        temperature : 25.0;
    }
    default_operating_conditions : XYZ;

    cell(Buffer_Type_HL_Level_shifter) {
        is_level_shifter : true;
        level_shifter_type : HL ;

        pg_pin(VDD) {
            voltage_name : VDD;
            pg_type : primary_power;
```

Chapter 9: Advanced Low-Power Modeling Level-Shifter Cells in a Multivoltage Design

```
        std_cell_main_rail : true;
    }
    pg_pin(VSS) {
        voltage_name : VSS;
        pg_type : primary_ground;
    }
    leakage_power() {
        when : "!A";
        value : 1.5;
        related_pg_pin : VDD;
    }

    pin(A) {
        direction : input;
    /* Defining the input_signal_level attribute identifies an Overdrive
    Level Shifter cell */
        input_signal_level : VDD1;
        related_power_pin : VDD;
        related_ground_pin : VSS;
        input_voltage_range ( 1.1 , 1.3);
    }

    pin(Z) {
        direction : output;
        related_power_pin : VDD;
        related_ground_pin : VSS;
        function : "A";
        power_down_function : "!VDD + VSS";
        output_voltage_range (0.6 , 0.9);

        timing() {
            related_pin : A;
            cell_rise(template) {
                ...
            }
            cell_fall(template) {
                ...
            }
            rise_transition(template) {
                ...
            }
            fall_transition(template) {
                ...
            }
        }
        internal_power() {
            related_pin : A;
            related_pg_pin : VDD;
            ...
        }
    } /* end pin group*/
...
} /*end cell group*/
```

```
    ...
} /*end library group */
```

Level-Shifter Cell With Virtual Bias Pins

This section provides an example of a level-shifter cell with virtual bias pins and two n-well regular wells for substrate-bias modeling.

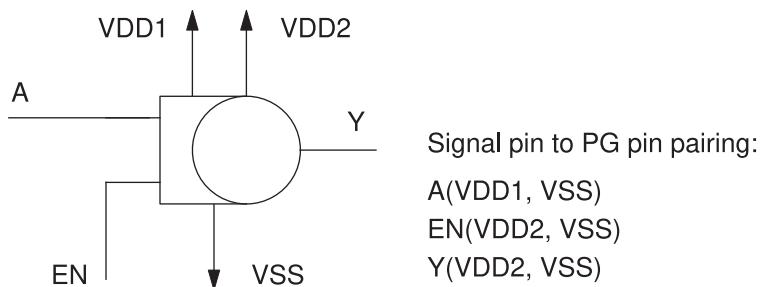
```
library (example_multi_rail_with_bias_pins) {
...
  cell ( level_shifter ) {
    pg_pin ( vdd1 ) {
      pg_type : primary_power ;
      ...
    }
    pg_pin ( vdd2 ) {
      pg_type : primary_power ;
      ...
    }
    pg_pin ( vss ) {
      pg_type : primary_ground ;
      ...
    }
    pg_pin ( vpw ) {
      pg_type : pwell ;
      ...
    }
    pg_pin ( vnwl ) {
      pg_type : nwell ;
      ...
    }
    pg_pin ( vnw2 ) {
      pg_type : nwell ;
      ...
    }
    pin ( I ) {
      direction : input;
      related_power_pin : vdd1
      related_ground_pin : vss
      related_bias_pin : "vnwl vpw"
      ...
    }
    pin ( Z ) {
      direction : output;
      function : "I";
      related_power_pin : "vdd2" ;
      related_ground_pin : "vss" ;
      related_bias_pin : "vnw2 vpw";
      power_down_function : "!vdd1 + !vdd2 + vss + !vnwl + !vnw2 + vpw";
      ...
    }
  } /* End of cell group */
} /* End of library group */
```

Simple Enable Level-Shifter Cell

Figure 69 shows the schematic of a basic enable level-shifter cell. The A signal pin is linked to the VDD1 and VSS power and ground pin pair, and the EN signal pin and the Y signal pin are linked to the VDD2 and VSS power and ground pin pair. The enable pin is linked to VDD2.

The example that follows the figure shows a library containing an enable level-shifter cell.

Figure 69 *Enable Level-Shifter Cell Schematic*



```
library(enable_level_shifter_library_example) {  
  
    voltage_map(VDD1, 0.8);  
    voltage_map(VDD2, 1.2);  
    voltage_map(VSS, 0.0);  
  
    operating_conditions(XYZ) {  
        voltage : 3.0;  
        ...  
    }  
    default_operating_conditions : XYZ;  
  
    cell(Enable_Level_Shifter) {  
        is_level_shifter : true;  
        level_shifter_type : LH ;  
  
        pg_pin(VDD1) {  
            voltage_name : VDD1;  
            pg_type : primary_power;  
            std_cell_main_rail : true;  
        }  
        pg_pin(VSS) {  
            voltage_name : VSS;  
            pg_type : primary_ground;  
        }  
        pg_pin(VDD2) {  
            voltage_name : VDD2;  
        }  
    }  
}
```

Chapter 9: Advanced Low-Power Modeling Level-Shifter Cells in a Multivoltage Design

```
    pg_type : primary_power;
}

leakage_power() {
    when : "!A";
    value : 1.5;
    related_pg_pin : VDD1;
}
leakage_power() {
    when : "!A";
    value : 1.5;
    related_pg_pin : VDD2;
}

pin(A) {
    direction : input;
    related_power_pin : VDD1;
    related_ground_pin : VSS;
    input_voltage_range ( 0.7 , 0.9 );
    level_shifter_data_pin : true;
}

pin(EN) {
    direction : input;
    related_power_pin : VDD2;
    related_ground_pin : VSS;
    input_voltage_range ( 1.1 , 1.3 );
    level_shifter_enable_pin : true;
}

pin(Y) {
    direction : output;
    related_power_pin : VDD2;
    related_ground_pin : VSS;
    function : "A * EN";
    power_down_function : "!VDD2 + VSS";
    output_voltage_range ( 1.1 , 1.3 );
    timing() {
        related_pin : "A EN";
        cell_rise(template) {
            ...
        }
        cell_fall(template) {
            ...
        }
        rise_transition(template) {
            ...
        }
        fall_transition(template) {
            ...
        }
    }
}
```

```
internal_power() {
    related_pin : "A EN";
    related_pg_pin : VDD1;
    ...
}
internal_power() {
    related_pin : "A EN";
    related_pg_pin : VDD2;
    ...
}
/* end pin group*/
...
/*end cell group*/
...
/*end library group*/
```

Enable Level-Shifter With Isolation

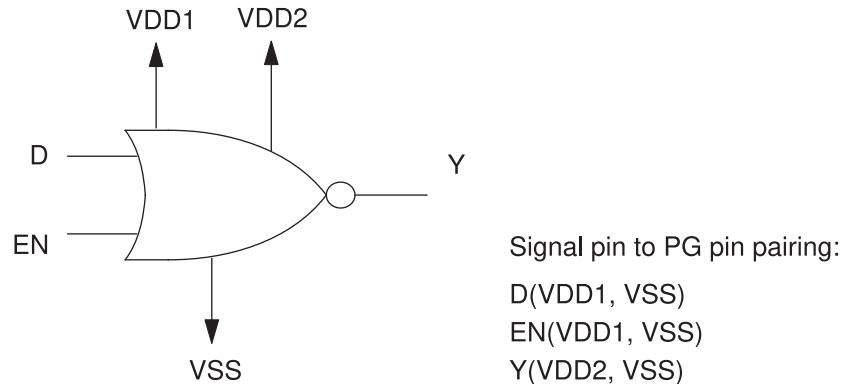
[Figure 70](#) shows the schematic of a NOR-type enable level-shifter cell with isolation capabilities. The example that follows the figure is a Liberty description of the cell. At the cell level, the Liberty model has both the `is_isolation_cell` and the `is_level_shifter` attributes set to `true`.

At the pin level, only the `isolation_cell_enable_pin` and the `isolation_cell_data_pin` isolation attributes (and not the level-shifter attributes) are specified. For more information about isolation attributes, see [Isolation Cell Modeling on page 446](#).

During compilation, the tool automatically derives the corresponding pin-based level-shifter attributes. Therefore, the .db file includes the following derived attributes:

- `level_shifter_data_pin` set to `true` on pin D
- `level_shifter_enable_pin` set to `true` on pin EN

Figure 70 NOR-Type Enable Level-Shifter Cell With Isolation



```
cell (ELS) {
    is_level_shifter : true;
    is_isolation_cell : true;
    pg_pin (VDD1) {
        voltage_name : "VDD1";
        pg_type : "primary_power";
        std_cell_main_rail : true;
        permit_power_down : true;
    }
    pg_pin (VDD2) {
        voltage_name : "VDD2";
        pg_type : "primary_power";
    }
    pg_pin (VSS) {
        voltage_name : "VSS";
        pg_type : "primary_ground";
    }
    pin (D) {
        related_power_pin : "VDD1";
        related_ground_pin : "VSS";

        isolation_cell_data_pin : true;
        direction : "input";
        ...
    }
    pin (EN) {
        related_power_pin : "VDD1";
        related_ground_pin : "VSS";
        isolation_cell_enable_pin : true;
        alive_during_partial_power_down : true;
        direction : "input";
    }
    pin (Y) {
        related_power_pin : "VDD2";
        related_ground_pin : "VSS";
        direction : "output";
    }
}
```

```

        alive_during_partial_power_down : true;
        power_down_function : "(!VDD1*!EN)+!VDD2+VSS";
        function : "! (D + EN)";
    }
} /* end cell */

```

Clamping in Latch-Based Enable Level-Shifter Cell

Table 28 shows the truth table for a latch-based enable level-shifter cell with two enable pins, EN1 and EN2. The example that follows the table shows the enable level-shifter cell modeled with the clamp function attributes.

Table 28 Truth Table for a Latch-Based Enable Level-Shifter Cell With Two Enable Pins

A	EN1	EN2	Y	Function
1/0	1	0	1/0	Level shifter
-	0	1	1	Clamp to 1
-	0	0	Qn-1	Latch
-	1	1	?	Forbidden

```

cell(clamp_1_latch_ELS) {
    is_level_shifter : true ;
    ...
    pg_pin(VDD1) {
        pg_type : primary_power;
        ...
    }
    pg_pin(VSS1) {
        pg_type : primary_ground;
        ...
    }
    pg_pin(VDD2) {
        pg_type : primary_power;
        std_cell_main_rail : true;
        ...
    }
    pg_pin(VSS2) {
        pg_type : primary_ground;
        ...
    }
    latch (IQ, IQN) {
        data_in : A;
        enable : EN1;
        preset : EN2;
    }
}

```

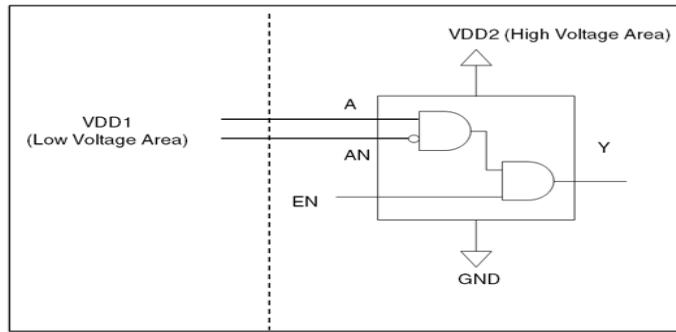
```
pin (A) {
    related_power_pin : VDD1;
    related_ground_pin : VSS1;
    direction : input;
    level_shifter_data_pin : true ;
    ...
}
pin (EN1) {
    related_power_pin : VDD2;
    related_ground_pin : VSS2;
    direction : input;
    level_shifter_enable_pin : true ;
    ...
}
pin (EN2) {
    related_power_pin : VDD2;
    related_ground_pin : VSS2;
    direction : input;
    level_shifter_enable_pin : true ;
    ...
}
pin (Y) {
    related_power_pin : VDD2;
    related_ground_pin : VSS2;
    direction : output;
    clamp_1_function: "!EN1 * EN2";
    clamp_latch_function: "!EN1 * !EN2";
    illegal_clamp_function: "EN1 * EN2";
    function : "IQ";
    ...
}
...
}
```

Differential Level-Shifter Cell

Figure 71 shows a schematic of a low-to-high differential level-shifter cell with the complementary input pins, A and AN, and an enable pin, EN. The differential level-shifter cell connects the VDD1 and VDD2 power domains. The differential level-shifter cell uses the supply voltage, VDD2. The supply voltage, VDD1, drives the inputs on the pins, A and AN. The example that follows the figure shows a typical model of the differential level-shifter cell.

Chapter 9: Advanced Low-Power Modeling
Level-Shifter Cells in a Multivoltage Design

Figure 71 Differential Level-Shifter Schematic



```
cell(DLS) {
    is_level_shifter : true ;
    pin_opposite ("A", "AN");
    contention_condition : "! (A^AN)*EN";
    ...
    pg_pin (VDD2) {
        pg_type : primary_power;
        std_cell_main-rail : true;
        ...
    }
    pg_pin (GND) {
        pg_type : primary_ground;
        ...
    }
    pin (A) {
        related_power_pin : VDD2 ;
        related_ground_pin : GND ;
        direction : input;
        level_shifter_data_pin : true ;
        input_signal_level : VDD1 ;
        ...
    }
    pin (AN) {
        related_power_pin : VDD2 ;
        related_ground_pin : GND ;
```

Chapter 9: Advanced Low-Power Modeling
Level-Shifter Cells in a Multivoltage Design

```
direction : input;
level_shifter_data_pin : true ;
input_signal_level : VDD2;

timing(){
    related_pin : "A";
    timing_type : non_seq_setup_rising ;
    rise_constraint(DLS_temp) {
        ...
    }
    fall_constraint(template_name) {
        ...
    }
}

timing(){
    related_pin : "A";
    timing_type : non_seq_setup_falling ;
    rise_constraint(DLS_temp) {
        ...
    }
    fall_constraint(template_name) {
        ...
    }
}

timing(){
    related_pin : "A";
    timing_type : non_seq_hold_rising ;
    rise_constraint(DLS_temp) {
        ...
    }
    fall_constraint(template_name) {
        ...
    }
}

timing(){
    related_pin : "A";
    timing_type : non_seq_hold_falling ;
    rise_constraint(DLS_temp) {
        ...
    }
    fall_constraint(template_name) {
        ...
    }
}

pin (EN) {
    related_power_pin : VDD2;
    related_ground_pin : GND;
    direction : input;
```

```
    level_shifter_enable_pin : true ;
    ...
}
pin (Y) {
    related_power_pin : VDD2;
    related_ground_pin : GND;
    direction : output;
    function : "(A*!AN)*EN";
    ...
}
...
}
```

Isolation Cell Modeling

In partition-based designs with multiple power supplies and voltages, the outputs from a shut-down partition to an active partition must be maintained at predictable signal levels. An isolation cell isolates the shut-down partition from the active partition. The isolation logic ensures that all the inputs to the active partition are clamped to a fixed value.

[Example 99](#) shows the syntax for modeling the isolation cell, and its pins.

Example 99 Isolation Cell Modeling Syntax

```
cell(isolation_cell) {
    is_isolation_cell : true ;
    ...
    pg_pin(pg_pin_name_P) {
        pg_type : primary_power ;
        permit_power_down : true ;
        ...
    }
    pg_pin(pg_pin_name_G) {
        pg_type : primary_ground;
        ...
    }

    pin (data) {
        direction : input;
        isolation_cell_data_pin : true ;
        ...
    }

    pin (enable) {
        direction : input;
        isolation_cell_enable_pin : true ;
        alive_during_partial_power_down : true ;
        ...
    }
    ...
    pin (output) {
```

```
direction : output;
alive_during_partial_power_down : true ;
function : "Boolean Expression";
power_down_function : "Boolean Expression";
...
}/* End pin group */

}/* End Cell group */
```

Cell-Level Attribute

This section describes a cell-level attribute of the isolation cells.

is_isolation_cell Attribute

The `is_isolation_cell` attribute identifies a cell as an isolation cell. The valid values of this attribute are `true` or `false`. If not specified, the default is `false`, meaning that the cell is an ordinary standard cell.

Pin-Level Attributes

This section describes the pin-level attributes for the isolation cells.

isolation_cell_data_pin Attribute

The `isolation_cell_data_pin` attribute identifies the data pin of an isolation cell. The valid values are `true` or `false`. If not specified, all the input pins of the isolation cell are considered to be data pins.

isolation_cell_enable_pin Attribute

The `isolation_cell_enable_pin` attribute identifies an enable or control pin of an isolation cell including a clock isolation cell. The valid values are `true` or `false`. The default is `false`.

power_down_function Attribute

The `power_down_function` attribute identifies the Boolean condition to switch off the output pin of the cell (when the cell is inactive due to the external power-pin states).

permit_power_down Attribute

The `permit_power_down` attribute indicates that the power pin can be powered down while in the isolation mode. The valid values are `true` or `false`. The default is `false`.

alive_during_partial_power_down Attribute

The `alive_during_partial_power_down` attribute indicates that the pin with this attribute is active while the isolation cell is partially powered down, and the corresponding power or the ground rails are not considered as the power reference. The valid values are `true` and `false`. The default is `false`, and in the default setting, the UPF isolation supply set is the power reference.

alive_during_power_up Attribute

The optional `alive_during_power_up` attribute specifies an active data pin that is powered by a more always-on supply. The default is `false`. If set to `true`, it indicates that the data pin is active while its related power pin is active.

You can specify this attribute only in a `pin` group where the `isolation_cell_data_pin` or the `level_shifter_data_pin` attribute is set to `true`.

Attribute Dependencies

The isolation cell attributes have the following dependencies:

- When the control pin of an isolation cell is activated, the output becomes a constant.
- The control pin of an isolation cell, that permits partial power down must be included in the Boolean expression of the `power_down_function` attribute for the same output. The control pin blocks the terms that use the powered-down rail, to set to `true`. To generate the output in the isolation mode, the active power rail is used.

Therefore, an isolation cell cannot be partially powered down if the `power_down_function` expression of its power pin does not include the `isolation_cell_enable_pin` attribute set.

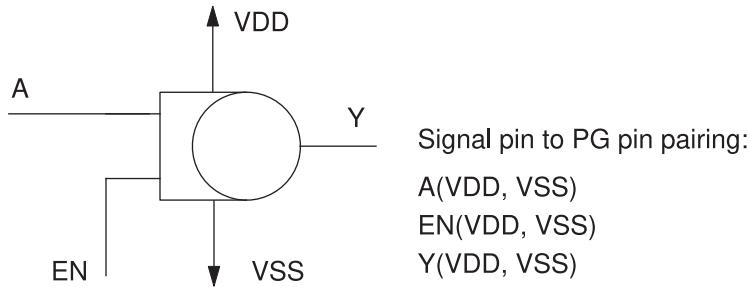
Isolation Cell Examples

Unlike level-shifter cells, isolation cells cannot shift voltage levels. All other characteristics are the same between a level-shifter cell and an isolation cell. The following sections show Liberty models of isolation cells.

Simple NAND-Type Isolation Cell

[Figure 72](#) shows a schematic and a library description of a simple NAND-type isolation cell. The library describes only the portion related to the power and ground pin syntax. In the figure, the A, EN, and Y signal pins are associated to the VDD and VSS power and ground pin pair. The example shows a library with an isolation cell.

Figure 72 Simple NAND-Type Isolation Cell



```
library(isolation_cell_library_example) {

    voltage_map(VDD, 1.0);
    voltage_map(VSS, 0.0);

    operating_conditions(XYZ) {
        voltage : 1.0;
        ...
    }
    default_operating_conditions : XYZ;

    cell(Isolation_Cell) {
        is_isolation_cell : true;
        dont_touch : true;
        dont_use : true;

        pg_pin(VDD) {
            voltage_name : VDD;
            pg_type : primary_power;
        }

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

        leakage_power() {
            when : "!A";
            value : 1.5;
            related_pg_pin : VDD;
        }
    }

    pin(A) {
        direction : input;
        related_power_pin : VDD;
        related_ground_pin : VSS;
        isolation_cell_data_pin : true;
    }
}
```

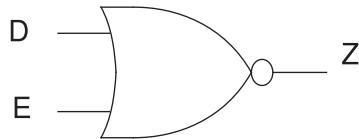
```
}

pin(EN) {
    direction : input;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    isolation_cell_enable_pin : true;
}
pin(Y) {
    direction : output;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    function : "A * EN";
    power_down_function : "!VDD + VSS";
    timing() {
        related_pin : "A EN";
        cell_rise(template) {
            ...
        }
        cell_fall(template) {
            ...
        }
        rise_transition(template) {
            ...
        }
        fall_transition(template) {
            ...
        }
    }
    internal_power() {
        related_pin : A;
        related_pg_pin : VDD;
        ...
    }
}
/* end pin group*/
...
}/*end cell group*/
...
}/*end library group*/
```

Simple NOR-Type Isolation Cell

Figure 73 shows a schematic of a simple NOR-type isolation cell with data input, D, and enable input, E. The example that follows the figure is a library description of the cell.

Figure 73 Simple NOR-Type Isolation Cell

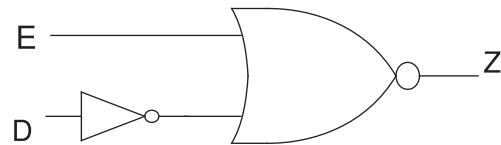


```
cell (NOR_cellA) {
    is_isolation_cell : true;
    pg_pin (VDD) {
        voltage_name : "VDD";
        pg_type : "primary_power";
        permit_power_down : true;
    }
    pg_pin (VSS) {
        voltage_name : "VSS";
        pg_type : "primary_ground";
    }
    pin (D) {
        related_power_pin : "VDD";
        related_ground_pin : "VSS";
        isolation_cell_data_pin : true;
        direction : "input";
        ...
    }
    pin (E) {
        related_power_pin : "VDD";
        related_ground_pin : "VSS";
        isolation_cell_enable_pin : true;
        alive_during_partial_power_down : true;
        direction : "input";
        ...
    }
    pin (Z) {
        related_power_pin : "VDD";
        related_ground_pin : "VSS";
        alive_during_partial_power_down : true;
        direction : "output";
        power_down_function : "(!VDD*!E) + VSS";
        function : "!(D + E)";
        ...
    }
} /* end cell */
```

NOR-Type Isolation Cell With Inverted Input

Figure 74 shows a schematic of a NOR-type isolation cell with an inverter at the data input, D. For the isolation cell to function correctly, the high signal voltage on D must be greater than or equal to the cell supply voltage, VDD. The example that follows the figure is a library description of the isolation cell.

Figure 74 NOR-Type Isolation Cell With Inverted Input



```
cell (NOR_cellB) {
    is_isolation_cell : true;
    pg_pin (VDD) {
        voltage_name : "VDD";
        pg_type : "primary_power";
        permit_power_down : true;
    }
    pg_pin (VSS) {
        voltage_name : "VSS";
        pg_type : "primary_ground";
    }
    pin (D) {
        related_power_pin : "VDD";
        related_ground_pin : "VSS";
        isolation_cell_data_pin : true;
        alive_during_power_up : true;

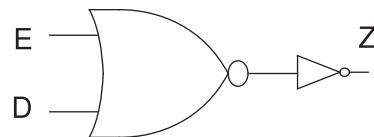
        /* ISO data pin must stay alive when its RPP is alive
         * 1. default value is false
         * 2. alive_during_power_up is only valid
         * when "isolation_cell_data_pin : true" */
        direction : "input";
        ...
    }
    pin (E) {
        related_power_pin : "VDD";
        related_ground_pin : "VSS";
        isolation_cell_enable_pin : true;
        alive_during_partial_power_down : true;
        direction : "input";
        ...
    }
    pin (Z) {
        related_power_pin : "VDD";
        related_ground_pin : "VSS";
        alive_during_partial_power_down : true;
        direction : "output";
        power_down_function : "(!VDD * !E) + VSS";
        function : "!(!D + E)";
        ...
    }
}
```

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

NOR-Type Isolation Cell With Inverted Output

Figure 75 shows a schematic of a NOR-type isolation cell with an inverter at the output, Z. For the isolation cell to function correctly, the high signal voltage on D must be less than or equal to the cell supply voltage, VDD. The example that follows the figure is a library description of the cell.

Figure 75 NOR-Type Isolation Cell With Inverted Output



```
cell (NOR_cellC) {
    is_isolation_cell : true;
    pg_pin (VDD) {
        voltage_name : "VDD";
        pg_type : "primary_power";
    }
    pg_pin (VSS) {
        voltage_name : "VSS";
        pg_type : "primary_ground";
    }
    pin (D) {
        related_power_pin : "VDD";
        related_ground_pin : "VSS";
        isolation_cell_data_pin : true;
        direction : "input";
        ...
    }
    pin (E) {
        related_power_pin : "VDD";
        related_ground_pin : "VSS";
        isolation_cell_enable_pin : true;
        direction : "input";
        ...
    }
    pin (Z) {
        related_power_pin : "VDD";
        related_ground_pin : "VSS";
        direction : "output";
        power_down_function : "!VDD + VSS";
        function : "(D + E)";
        ...
    }
} /* end cell */
```

Isolation Cell Without Enable Pin

A single-bit isolation cell without an enable pin has one data input and one output pin. An N-bit multibit isolation cell without enable pins, has N data input and N output pins.

The output pin Liberty model of such cells must have a clamp function attribute. During compilation, the tool automatically sets the cell-level `is_no_enable` attribute to `true` for such cells.

[Table 29](#) shows the truth table for an isolation cell without any enable pin. D and Z are the input and output pins, respectively. VDD is the related power pin of the input, and VDDO is the related power pin of the output. The output clamps to latch condition when the input circuit is powered off and the output circuit remains powered on. The example that follows the table shows the Liberty model of the isolation cell.

Table 29 Truth Table for a No-Enable-Pin Isolation Cell

A (Input)	VDD	VDDO	Z (Output)
Any	ON	ON	D
Any	OFF	ON	Latch
Any	OFF	OFF	X
Any	ON	OFF	X

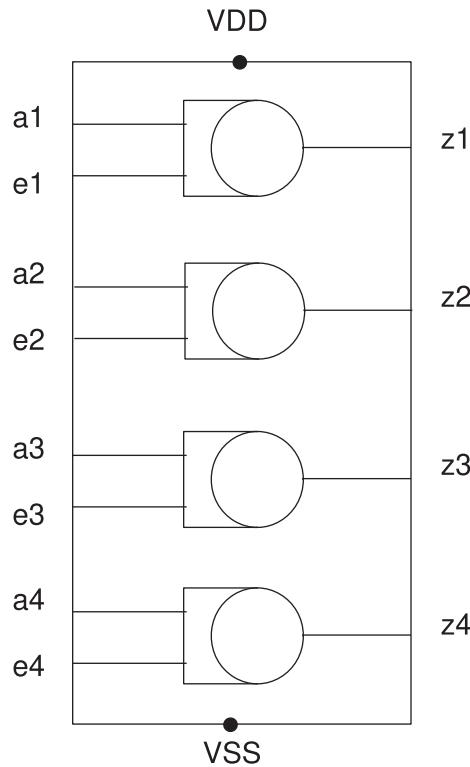
```
library (mylib) {
    ...
    voltage_map(VDD, 0.80);
    voltage_map(VDDO, 1.20);
    ...
    cell(No_Enable_Iso) {
        is_isolation_cell : true;
        pg_pin(VDD) {
            voltage_name : VDD;
            pg_type : primary_power;
        }
        pg_pin(VDDO) {
            voltage_name : VDDO;
            pg_type : backup_power;
        }
        pg_pin(VSS) {
            voltage_name : VSS;
            pg_type : primary_ground;
        }
        pin(D) {
            direction : input;
            related_power_pin : VDD;
            related_ground_pin : VSS;
        }
    }
}
```

```
    isolation_cell_data_pin : true;
}
pin(Z) {
    direction : output;
    related_power_pin : VDDO;
    related_ground_pin : VSS;
    function : "D";
    clamp_latch_function : "!VDD";
    power_down_function : "!VDDO + VSS";
    timing() {
        related_pin : "D";
        ...
    } /* end timing group */
}/* end pin group*/
}/*end cell group*/
}/*end library group*/
```

Multibit Isolation Cell

Figure 76 shows a multibit isolation cell with four data input pins, four enable pins for each of the data input pins, and four data output pins. Each of the data outputs is a linear function of the respective data input. The example following the figure is a model of the isolation cell.

Figure 76 4-bit Isolation Cell



```
cell (4_bit_iso) {
    is_isolation_cell : true;
    pg_pin (VDD) {
        pg_type : primary_power;
        voltage_name : VDD;
    }
    pg_pin (VSS) {
        pg_type : primary_ground;
        voltage_name : VSS;
    }
    ...
    pin (a1) {
        isolation_cell_data_pin : true;
        related_power_pin : VDD;
        related_ground_pin : VSS;
    ...
}
pin (a2) {
    isolation_cell_data_pin : true;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    ...
}
```

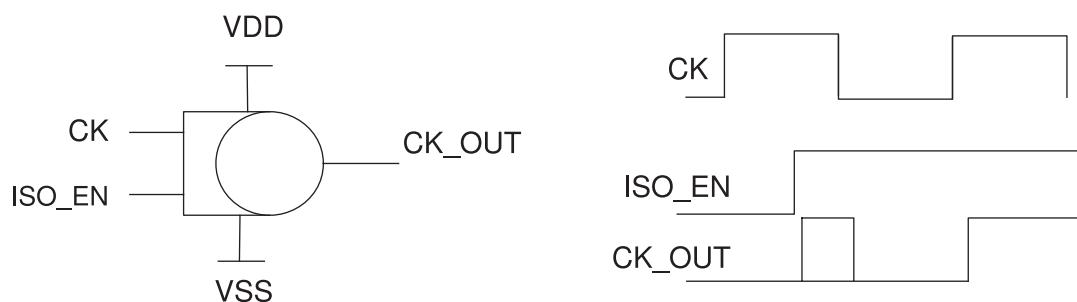
```
    }
pin (a3) {
    isolation_cell_data_pin : true;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    ...
}
pin (a4) {
    isolation_cell_data_pin : true;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    ...
}
pin (e1) {
    isolation_cell_enable_pin : true;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    ...
}
pin (e2) {
    isolation_cell_enable_pin : true;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    ...
}
pin (e3) {
    isolation_cell_enable_pin : true;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    ...
}
pin (e4) {
    isolation_cell_enable_pin : true;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    ...
}
pin (z1) {
    related_power_pin : VDD;
    related_ground_pin : VSS;
    power_down_function : !VDD + VSS;
    function : a1 * e1;
    clam_0_function : !e1;
    ...
}
pin (z2) {
    related_power_pin : VDD;
    related_ground_pin : VSS;
    power_down_function : !VDD + VSS;
    function : a2 * e2;
    clam_0_function : !e2;
```

```
...
}
pin (z3) {
    related_power_pin : VDD;
    related_ground_pin : VSS;
    power_down_function : !VDD + VSS;
    function : a3 * e3;
    clam_0_function : !e3;
...
}
pin (z4) {
    related_power_pin : VDD;
    related_ground_pin : VSS;
    power_down_function : !VDD + VSS;
    function : a4 * e4;
    clam_0_function : !e4;
...
}
}
```

Clock-Isolation Cell Modeling

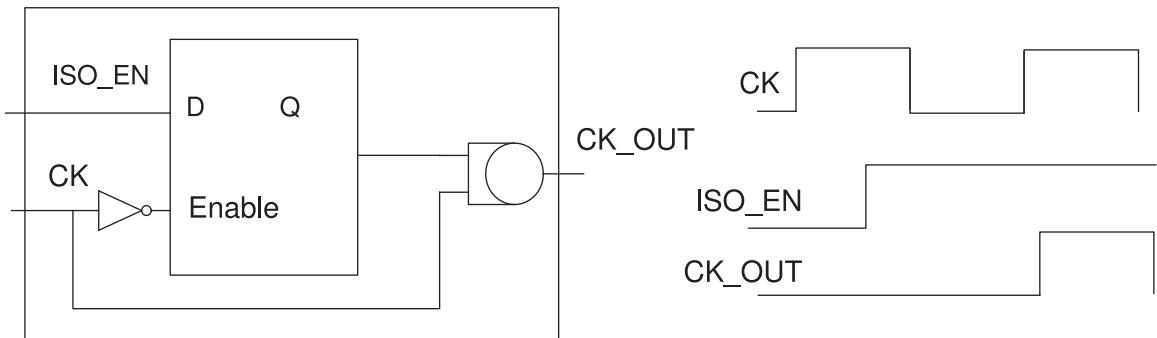
Using combinational isolation cells to isolate clock signals might result in phase-clipped outputs. [Figure 77](#) shows an AND isolation cell. The phase of the output clock signal, CK_OUT, becomes clipped, depending on the arrival time of the isolation-enable signal, ISO_EN, with respect to the input clock signal, CK.

Figure 77 Clock Isolation With AND Isolation Cell



To avoid phase-clipping, use clock-isolation cells for clock isolation. [Figure 78](#) shows a schematic and a typical output of a clock-isolation cell.

Figure 78 Clock Isolation Cell Schematic



Note:

The phase of the output clock signal, CK_OUT, is not clipped.

The modeling syntax of the clock-isolation cell and its pins is as follows:

```
cell (cell_name) {
    is_clock_isolation_cell : true;
    pin(input_pin) {
        clock_isolation_cell_clock_pin : true;
        direction : input;
        ...
    }
    pin(input_pin) {
        isolation_cell_enable_pin : true;
        direction : input;
        ...
    }
    pin(output_pin_name) {
        direction : output;
        ...
    }
    ...
}
```

Cell-Level Attribute

This section describes the cell-level attribute of clock-isolation cells.

is_clock_isolation_cell Attribute

The `is_clock_isolation_cell` attribute identifies a cell as a clock-isolation cell. The default is `false`, meaning that the cell is a standard cell.

Pin-Level Attributes

This section describes the pin-level attributes for clock-isolation cells.

isolation_cell_enable_pin Attribute

The `isolation_cell_enable_pin` attribute identifies an enable or control pin of a clock-isolation cell. The default is `false`.

clock_isolation_cell_clock_pin Attribute

The `clock_isolation_cell_clock_pin` attribute identifies an input clock pin of a clock-isolation cell. The default is `false`.

Clock Isolation Cell Examples

[Example 100](#) shows a library description of the clock-isolation cell in [Figure 78](#) on page 459.

Example 100 Library Description of the Clock-Isolation Cell

```
library (my_lib) {
    ...
    cell(ck_iso_cell) {
        is_clock_isolation_cell : true;
        ...
        pg_pin (VDD) {
            pg_type : primary_power;
            voltage_name : VDD;
        }
        pg_pin (VSS) {
            pg_type : primary_ground;
            voltage_name : VSS;
        }
        statetable(" CK ISO_EN ", "Q ") {
            table : " L L : - : L , \
                      L H : - : H , \
                      H - : - : N ";
        }
        pin(CK) {
            clock_isolation_cell_clock_pin : true;
            direction : input;
            related_ground_pin : VSS;
            related_power_pin : VDD;
            ...
        }
        pin(ISO_EN) {
            isolation_cell_enable_pin : true;
            direction : input;
            related_ground_pin : VSS;
            related_power_pin : VDD;
            ...
        }
        pin(CK_OUT) {
            direction : output;
            power_down_function : "!VDD + VSS";
            related_ground_pin : VSS;
        }
    }
}
```

```
related_power_pin : VDD;
state_function : "CK * Q";
timing() {
    related_pin : "CK";
    timing_sense : "positive_unate";
    ...
}
...
}
}
```

[Example 101](#) shows the clock-isolation cell also modeled as a clock-gating cell. To model a cell as both a clock-isolation cell and a clock-gating cell, use both the clock-isolation and clock-gating attributes.

Example 101 Example of a Cell Modeled as Both Clock-Isolation and Clock-Gating Cell

```
cell(ICG_CK_ISO_CELL) {
    is_clock_isolation_cell : true;
    clock_gating_integrated_cell : "latch_posedge";
    ...
    pg_pin (VDD) {
        pg_type : primary_power;
        voltage_name : VDD;
    }
    pg_pin (VSS) {
        pg_type : primary_ground;
        voltage_name : VSS;
    }
    statetable(" CK EN ", "Q ") {
        table : " L L : - : L , \
                  L H : - : H , \
                  H - : - : N ";
    }
    pin(CK) {
        clock_isolation_cell_clock_pin : true;
        clock_gate_clock_pin : true;
        direction : input;
        related_ground_pin : VSS;
        related_power_pin : VDD;
        ...
    }
    pin(EN) {
        isolation_cell_enable_pin : true;
        clock_gate_enable_pin : true;
        direction : input;
        related_ground_pin : VSS;
        related_power_pin : VDD;
        ...
    }
    pin(CK_OUT) {
```

```
direction : output;
clock_gate_output_pin : true;
power_down_function : "!VDD + VSS";
related_ground_pin : VSS;
related_power_pin : VDD;
state_function : "CK * Q";
timing() {
    related_pin : "CK";
    timing_sense : "positive_unate";
    ...
}
...
}
```

Clamping Isolation Cell Output Pins

In isolation cells with multiple enable pins, clamping the outputs might be required to maintain them at particular logic levels, such as 0,1, z, or a previously-latched value.

The clamp function modeling syntax for the isolation cell and its pins is as follows:

```
cell (cell_name) {
    is_isolation_cell : true;
    ...
    pin(input_pin) {
        direction : input;
        isolation_cell_data_pin : true;
        ...
    }
    pin(input_pin) {
        direction : input;
        isolation_cell_enable_pin : true;
        ...
    }
    pin(input_pin) {
        direction : input;
        isolation_cell_enable_pin : true;
        ...
    }
    pin(output_pin_name) {
        direction : output;
        clamp_0_function : "Boolean expression" ;
        clamp_1_function : "Boolean expression" ;
        clamp_z_function : "Boolean expression" ;
        clamp_latch_function : "Boolean expression" ;
        illegal_clamp_condition : "Boolean expression" ;
        ...
    }
    ...
}
```

Pin-Level Attributes

This section describes the pin-level clamp function attributes to clamp the isolation cell output pins.

clamp_0_function Attribute

The `clamp_0_function` attribute specifies the input condition for the enable pins of an isolation cell when the output clamps to 0.

clamp_1_function Attribute

The `clamp_1_function` attribute specifies the input condition for the enable pins of an isolation cell when the output clamps to 1.

clamp_z_function Attribute

The `clamp_z_function` attribute specifies the input condition for the enable pins of an isolation cell when the output clamps to z.

clamp_latch_function Attribute

The `clamp_latch_function` attribute specifies the input condition for the enable pins of an isolation cell when the output clamps to the previously latched value.

illegal_clamp_condition Attribute

The `illegal_clamp_condition` attribute specifies the invalid condition for the enable pins of an isolation cell. If the `illegal_clamp_condition` attribute is not specified, the invalid condition does not exist.

Note:

All the input pins used in the Boolean expressions of the `clamp_0_function`, `clamp_1_function`, `clamp_z_function`, and `clamp_latch_function` attributes must be enable pins with the `isolation_cell_enable_pin` attribute set to `true`. The Boolean expressions of the `clamp_0_function`, `clamp_1_function`, `clamp_z_function`, `clamp_latch_function`, and `illegal_clamp_condition` attributes must be mutually exclusive.

Example of Clamping in Isolation Cell

The table in [Clamping in Latch-Based Enable Level-Shifter Cell](#) shows the truth table for an isolation cell with two enable pins, EN1 and EN2. [Example 102](#) shows the isolation cell modeled with a clamp function attribute.

Table 30 Truth Table for an Isolation Cell With Two Enable Pins

A	EN1	EN2	Y	Function
0/1	1	1	0/1	AND
-	0	1	0	Clamp to 0
-	1	0	0	Clamp to 0
-	0	0	0	Clamp to 0

Example 102 Using a Clamp Function Attribute to Model an Isolation Cell

```
cell(ISO_clamp_0) {
    is_isolation_cell : true;
    ...
    pin(A) {
        direction : input;
        capacitance : 1.0;
        isolation_cell_data_pin : true;
        ...
    }
    pin(EN1) {
        direction : input;
        capacitance : 1.0;
        isolation_cell_enable_pin : true;
        ...
    }
    pin(EN2) {

        direction : input;
        capacitance : 1.0;
        isolation_cell_enable_pin : true;
        ...
    }
    pin(Y) {
        direction : output;
        function : "A*EN1*EN2";
        clamp_0_function : "! (EN1*EN2)";
        ...
    }
}
```

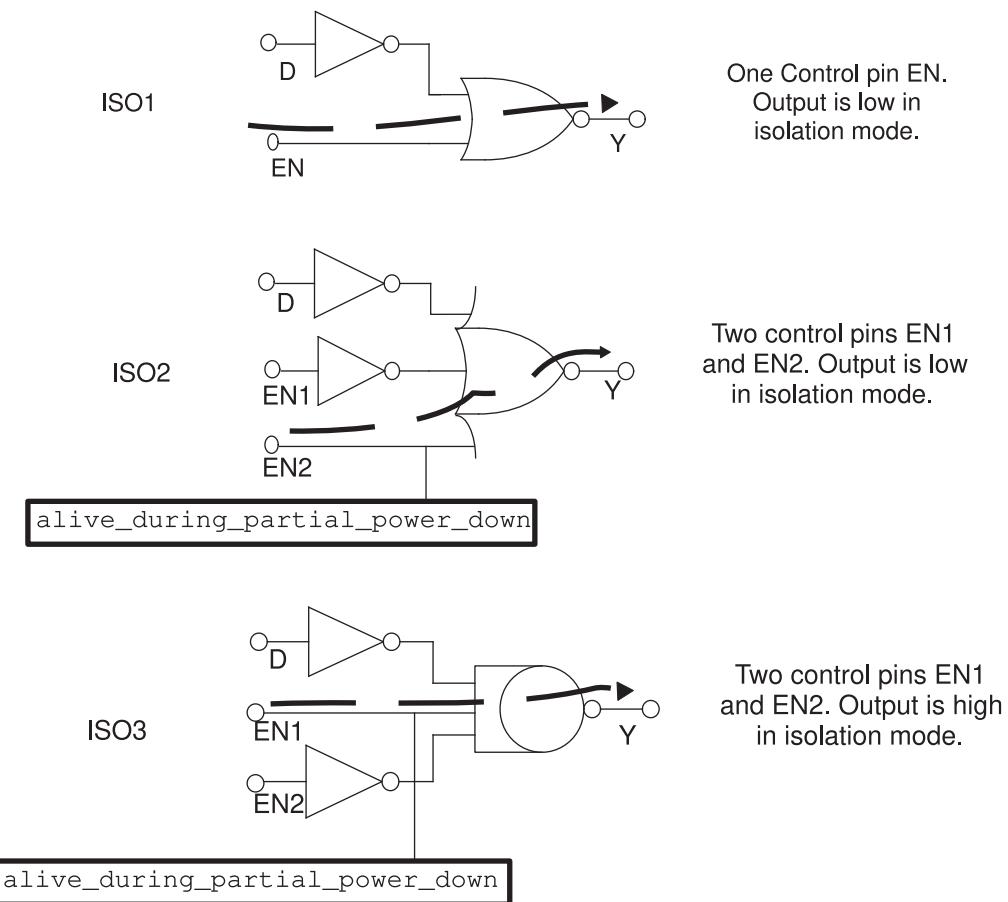
Isolation Cells With Multiple Control Pins

In partition-based designs, output pins of an isolation-cell need to be maintained at predefined levels for a significant period of time. In such designs, using isolation cells with multiple control or enable pins reduces the leakage current. Multiple control pins allow

an isolation cell to be partially powered down. For example, in an isolation cell with two control pins, one control pin controls the output level. The other control pin is used to partially power-down the isolation cell while the output level is maintained.

[Figure 79](#) shows the gate-level diagrams of isolation cells with multiple control pins. In each of the three diagrams, the dotted-line arrow traces the path from the isolation-control pin to the output pin.

Figure 79 Gate-Level Diagrams of Isolation Cells With Multiple Control Pins



The isolation control pins permit the power pins (not shown in [Figure 79](#)) to power down. The `alive_during_partial_power_down` attribute on the isolation-control pins indicate that these pins remain active even when the corresponding power pins are powered down. [Example 103](#) shows the partially powered down model of the cell, ISO1, depicted in [Figure 79](#). [Table 31](#) shows the Boolean expressions for the `function` and `power_down_function` attributes of the isolation cells depicted in [Figure 79](#).

Example 103 Partially Powered Down Model of the ISO1 Cell

```

cell ( ISO1 ) {
    is_isolation_cell : true;
    pg_pin ( VDD ) {
        pg_type : primary_power;
        permit_power_down : true
    }
    pg_pin ( VSS ) {
        pg_type : primary_ground ;
    }
    pin ( D ) {
        isolation_cell_data_pin : true ;
    }
    pin ( EN ) {
        direction : input
        isolation_cell_enable_pin : true ;
        alive_during_partial_power_down : true ;
    }
    pin ( Y ) {
        direction : output ;
        alive_during_partial_power_down : true ;
        function : D * !EN ;
        power_down_function : (!VDD * !EN) + VSS ;
    }
}

```

Note:

By default, the `related_power_pin` and `related_ground_pin` attributes are mapped to VDD and VSS, respectively. The `related_power_pin` and `related_ground_pin` attributes are not specified for the input and output pins in [Example 103](#).

Table 31 Boolean Expressions for the function and power_down_function Attributes of the Isolation Cells in [Figure 79](#)

Cell	function (Output Y)	power_down_function (Output Y)
ISO1	D * !EN	(!VDD * !EN) + VSS
ISO2	D * EN1 * !EN2	(!VDD * !EN2) + VSS
ISO3	D + !EN1 + EN2	!VDD + (VSS + EN1)

Pins with the `alive_during_partial_power_down` attribute do not require the power pins to be active. For example, for the cell ISO2, the isolation-control pin, EN2, does not require VDD to be active. This pin controls the power-down of VDD through the Boolean expression of the `power_down_function` attribute as shown in [Table 31](#). The other

control pin, EN1, without the `alive_during_partial_power_down` attribute requires both VDD and VSS to be active, and is not included in the Boolean expression of the `power_down_function` attribute.

In each of the isolation cells in [Figure 79](#), the `permit_power_down` attribute is set on a power pin when there is at least one pin with the `isolation_cell_enable_pin` attribute. For example, for the cell ISO2, the `permit_power_pin` attribute is set on the power pin, VDD, given the `isolation_cell_enable_pin` attribute is set on the pin, EN2.

Switch Cell Modeling

Switch cells, also known as multithreshold complementary metal oxide semiconductor cells (power-switch cells), are used to reduce power. They are divided into the following two classes:

- Coarse grain

There are two types of coarse-grain switch cells, header switch cells, and footer switch cells. The header switch cells control the power nets based on a PMOS transistor, and the footer switch cells control the ground nets based on an NMOS transistor. The coarse grain cell is a switch that drives the power to other logic cells. It is used as a big switch to the supply rails and to turn off design partitions when the relative logic is inactive. Therefore, coarse-grain switch cells can reduce the leakage of the inactive logic.

In addition, coarse-grain switch cells can also have the properties of a fine-grain switch cell. For example, they can act as a switch and have output pins that might or might not be shut off by the internal switch pins.

- Fine grain

To reduce the leakage power, the fine-grain switch cell includes an embedded switch pin that is used to turn off the cell when it is inactive.

The syntax supports fine-grain switch cells only for macro cells.

Coarse-Grain Switch Cells

Coarse-grain switch cells must have the following properties:

- They must be able to model the condition under which the cell turns off the controlled design partition or the cells connected in the output power pin's fanout logical cone. This is modeled with a switching function based on special switch signal pins as well as a separate but related power-down function based on power pin inputs.
- They must be able to model the “acknowledge” output pins (output pins whose signal is used to propagate the switch signal to the next-switch cell or to a power

controller input's logic cloud). Timing is also propagated from the input switch pin to the acknowledge output pins.

- They must have at least one virtual output power and ground pin (virtual VDD or virtual VSS), one regular input power and ground pin (VDD or VSS), and one switch input pin. There is no limit on the number of pins a coarse-grain cell can have.

The following describes a simple coarse-grain switch header cell and a simple coarse-grain switch footer cell:

- Header cell: one switch input pin, one VDD (power) input power and ground pin, and one virtual VDD (internal power) output power and ground pin
- Footer cell: one switch input pin, one virtual VSS (internal ground) output power and ground pin, and one VSS (ground) input power and ground pin
- They can have multiple switch pins and multiple acknowledge output pins.
- They must have the steady state current (I/V) information to determine the resistance value when the switch is on.
- The timing information can be specified for coarse-grain switch cells on the output pins, and it can be state dependent for switch pins.

The power output pins in a coarse-grain switch cell can have the following two states:

- Awake or On

In this state, the input power level is transmitted through the cell to either a 1 or 0 on the output power pin, depending on other prior switch cells in series settings.

- Off

In this state, the sleep pin deactivates the pass-through function, and the output power pin is set to X.

Coarse-Grain Switch Cell Syntax

The following syntax is a portion of the coarse-grain switch cell syntax:

```
library(coarse_grain_library_name) {
...
lu_table_template (template_name)
    variable_1 : input_voltage;
    variable_2 : output_voltage;
    index_1 ( float, ... );
    index_2 ( float, ... );
}
...
cell(cell_name) {
    switch_cell_type : coarse_grain;
```

```
...
pg_pin (VDD/VSS pin name) {
    pg_type : primary_power | primary_ground;
    direction : input ;
    ...
}
/* Virtual power and ground pins use "switch_function" to describe the
logic to shut off the attached design partition */

pg_pin (virtual VDD/VSS pin name) {
    pg_type : internal_power | internal_ground;
    direction: output;
    ...
    switch_function : "function_string";
    pg_function : "function_string";

}
dc_current (dc_current_name) {
    related_switch_pin : input_pin_name;
    related_pg_pin : VDD pin name;
    related_internal_pg_pin : Virtual VDD;

    values("float, ...");
}
pin (input_pin_name) {
    direction : input;
    switch_pin : true;
    ...
}
...
/* The acknowledge output pin uses "function" to represent the
propagated
switching
signal
*/
pin(acknowledge_output_pin_name) {
    ...
    function : "function_string";
    power_down_function : "function_string";
    direction : output;
    ...
} /* end pin group */
} /* end cell group */
```

You can use the following syntax for intrinsic parasitic models.

```
switch_cell_type : coarse_grain;
dc_current (template_name) {
    related_switch_pin : pin_name;
    related_pg_pin : pg_pin_name;
    related_internal_pg_pin : pg_pin_name;
    index_1 ( "float, ..." );
```

```
    index_2 ( "float, ..." );
    values ( "float, ..." );
}
```

Library-Level Group

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

lu_table_template Group

The library-level `lu_table_template` group models the templates for the steady state current information that is used within the `dc_current` group. The `input_voltage` value specifies input voltage values for the switch pin. The `output_voltage` value specifies the output voltage values of the switch pin. The `input_voltage` and `output_voltage` values are the absolute gate voltage and absolute drain voltage, respectively, when a CMOS transistor is used to model a power-switch cell.

The `dc_current` group, which is used for steady state current modeling, must be defined at the cell level. It is defined by two indexes: `index_1` and `index_2`. The `index_1` attribute represents a string that includes a comma-separated set of N values, where N represents the table rows. The values define the voltage levels measured at either the input voltage or the output voltage. When referring to the input voltage, the voltage level is measured at the related switch pin, and the set of N values must have at least two values for N. When referring to the output voltage, the voltage level is measured at the related internal PG pin, and the set of N values must have at least three values for N. This voltage level is related to a common reference ground for the cell. The set of N `index_1` values must increase monotonically.

The `index_2` attribute represents a string that includes a comma-separated set of M values, where M represents the table columns. The values define the voltage levels that are specified at either the input voltage or the output voltage. When referring to the input voltage, the voltage level is measured at the related switch pin, and the set of M values must have at least two values for M. When referring to the output voltage, the voltage level is measured at the related internal PG pin, and the set of M values must have at least three values for M. This voltage level is related to a common reference ground for the cell. The set of M `index_2` values must increase monotonically.

Cell-Level Attribute and Group

The following cell-level attribute and group apply to coarse-grain switch cells.

switch_cell_type Attribute

The `switch_cell_type` attribute specifies the switch cell type so that it is not indirectly derived from the cell modeling description. Valid values are `coarse_grain` and `fine_grain`.

dc_current Group

The cell-level `dc_current` group models the steady state current information, similar to the `lu_table_template` group. The table is used to specify the DC current through the cell's output pin (generally the `related_internal_pg_pin`) in the current units specified at the library level using the `current_unit` attribute.

The `dc_current` group includes the `related_switch_pin`, `related_pg_pin`, and `related_internal_pg_pin` attributes, which are described in the following sections.

related_switch_pin Attribute

The `related_switch_pin` string attribute specifies the name of the related switch pin for the coarse-grain switch cell.

related_pg_pin Attribute

The `related_pg_pin` string attribute is used to specify the name of the power and ground pin that represents the VDD or VSS power source.

related_internal_pg_pin Attribute

The `related_internal_pg_pin` string attribute is used to specify the name of the power and ground pin that represents the virtual VDD or virtual VSS power source.

Pin-Level Attributes

The following attributes are pin-level attributes for coarse-grain switch cells.

switch_function Attribute

The `switch_function` string attribute identifies the condition when the attached design partition is turned off by the input `switch_pin`.

For a coarse-grain switch cell, the `switch_function` attribute can be defined at both controlled power and ground pins (virtual VDD and virtual VSS for `pg_pin`) and the output pins. It identifies the signal pins that can turn the power pin on.

When the `switch_function` attribute is defined in the controlled power and ground pin, it is used to specify the Boolean condition under which the cell switches off (or drives an X to) the controlled design partitions, including the traditional signal input pins only with no related power pins to this output.

switch_pin Attribute

The `switch_pin` attribute is a pin-level Boolean attribute. When it is set to true, it is used to identify the pin as the switch pin of a coarse-grain switch cell.

function Attribute

The `function` attribute in a pin group defines the value of an output pin or inout pin in terms of the input pins or inout pins in the cell group or model group. The `function` attribute describes the Boolean function of only nonsleep input signal pins.

pg_function Attribute

The `pg_function` attribute models the logical function of a virtual or derived power and ground (PG) pin as a Boolean expression involving the cells input signal pins, internal signal pins, and PG pins. The Boolean expression attribute is checked during library compile to ensure that only one `pg_pin` is always active at this virtual or derived PG pin. If more than one `pg_pin` is found to be active at the virtual or the derived `pg_pin` output, the `read_lib` command generates an error.

```
pg_pin (VOUT) {  
    voltage_name : "VOUT";  
    pg_type : "internal_power";  
    direction : "output";  
    pg_function : "VDD1 * !(en1 & en2 & sleep) + VDD2 * (en1 & en2 &  
    sleep)";  
}
```

power_down_function Attribute

The `power_down_function` string attribute is used to identify the condition under which the cell's signal output pin is switched off (when the cell is in off mode due to the external power pin states).

pg_pin Group

The cell-level `pg_pin` group is used to model the VDD and VSS pins and virtual VDD and VSS pins of a coarse-grain switch cell. The syntax is based on the Y-2006.06 power and ground pin syntax.

Fine-Grained Switch Support for Macro Cells

A macro cell with a fine-grained switch is a cell that contains a special switch transistor with a control pin that can turn off the power supply of the cell when it is idle. This significantly lowers the power consumption of a design.

With the growing popularity of low-power designs, macro cells with fine-grained switches play an important role. The syntax identifies a cell as a macro cell and specifies the correct power pin that supplies power to each signal pin.

Macro Cell With Fine-Grained Switch Syntax

```
cell(cell_name) {  
    is_macro_cell : true;  
    switch_cell_type : coarse_grain | fine_grain;
```

```
pg_pin (power/ground pin name) {
    pg_type : primary_power | primary_ground | backup_power |
    backup_ground;
    direction: input | inout | output;
    ...
}

/* This is a special pg pin that uses "switch_function" to describe the
logic to shut
off the attached design partition */
pg_pin (internal power/ground pin name) {
    direction: internal | input | output | inout;
    pg_type : internal_power | internal_ground;
    switch_function : "function_string";
    pg_function : "function_string";
    ...
}

pin (input_pin_name) {
    direction : input | inout;
    switch_pin : true | false;
    ...
}
...
pin(output_pin_name) {
    direction : output | inout;
    power_down_function : "function_string";
    ...
} /* end pin group */
} /* end cell group */
```

Cell-Level Attributes

The following attributes are cell-level attributes for macro cells with fine-grained switches.

is_macro_cell Attribute

The `is_macro_cell` simple Boolean attribute identifies whether a cell is a macro cell. If the attribute is set to `true`, the cell is a macro cell. If it is set to `false`, the cell is not a macro cell.

switch_cell_type Attribute

The `switch_cell_type` attribute is enhanced to support macro cells with internal switches. The valid enumerated values for this attribute are `coarse_grain` and `fine_grain`.

pg_pin Group

The following attribute can be specified under the `pg_pin` group for macro cells with fine-grained switches.

direction Attribute

The `direction` attribute supports `internal` as a valid value for macros when the internal power and ground is not visible or accessible at the cell boundary.

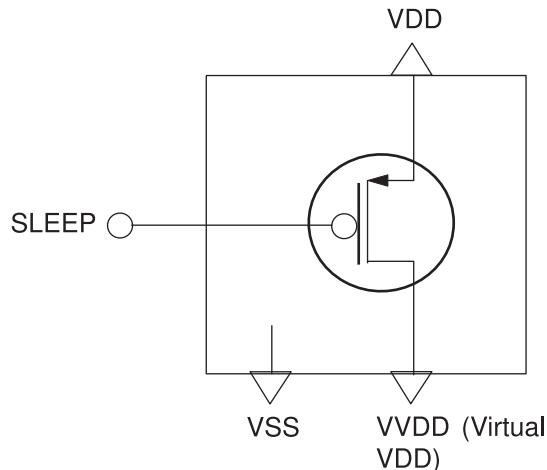
Switch-Cell Modeling Examples

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

Simple Coarse-Grain Header Switch Cell

[Figure 80](#) and the example that follows it show a simple coarse-grain header switch cell.

Figure 80 Simple Coarse-Grain Header Switch Cell



```
library (simple_coarse_grain_lib) {

    ...
    current_unit : 1mA;
    ...

    voltage_map(VDD, 1.0);
    voltage_map(VVDD, 0.8);
    voltage_map(VSS, 0.0);

    operating_conditions(XYZ) {
        process : 1.0;
        voltage : 1.0;
        temperature : 25.0;
    }
    default_operating_conditions : XYZ;
}
```

```
lu_table_template ( ivt1 ) {
    variable_1 : input_voltage;
    variable_2 : output_voltage;
    index_1 ( "0.1, 0.2, 0.4, 0.8, 1.0" );
    index_2 ( "0.1, 0.2, 0.4, 0.8, 1.0" );
}
...
cell ( Simple_CG_Switch ) {
    ...
    switch_cell_type : coarse_grain;

    pg_pin ( VDD ) {
        pg_type : primary_power;
        direction : input;
        voltage_name : VDD;
    }

    pg_pin ( VVDD ) {
        pg_type : internal_power;
        voltage_name : VVDD;
        direction : output ;
        switch_function : "SLEEP" ;
        pg_function : "VDD" ;
    }

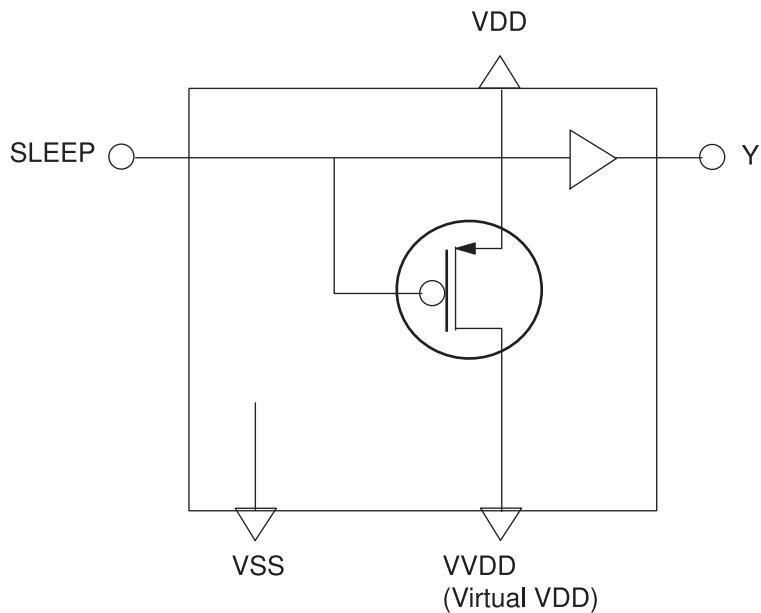
    pg_pin ( VSS ) {
        pg_type : primary_ground;
        direction : input;
        voltage_name : VSS;
    }

/* I/V curve information */
dc_current ( ivt1 ) {
    related_switch_pin : SLEEP;      /* control pin */
    related_pg_pin : VDD;           /* source */
    related_internal_pg_pin : VVDD; /* drain */
    values( "0.010, 0.020, 0.030, 0.030, 0.030", \
            "0.011, 0.021, 0.031, 0.041, 0.051", \
            "0.012, 0.022, 0.032, 0.042, 0.052", \
            "0.013, 0.023, 0.033, 0.043, 0.053", \
            "0.014, 0.024, 0.034, 0.044, 0.054");
}
    ...
}
pin ( SLEEP ) {
    switch_pin : true;
    capacitance: 1.0;
    related_power_pin : VDD;
    related_ground_pin : VSS;
} /* end pin */
} /* end cell */
} /* end library */
```

Complex Coarse-Grain Header Switch Cell

Figure 81 and the example that follows it show a complex coarse-grain header switch cell.

Figure 81 Complex Coarse-Grain Header Switch Cell



```
library (complex_coarse_grain_lib) {
    ...
    current_unit : 1mA;
    ...
    voltage_map(VDD, 1.0);
    voltage_map(VVDD, 0.8);
    voltage_map(VSS, 0.0);

    operating_conditions(XYZ) {
        process : 1.0;
        voltage : 1.0;
        temperature : 25.0;
    }
    default_operating_conditions : XYZ;

    lu_table_template ( ivt1 ) {
        variable_1 : input_voltage;
        variable_2 : output_voltage;
        index_1 ( "0.1, 0.2, 0.4, 0.8, 1.0" );
        index_2 ( "0.1, 0.2, 0.4, 0.8, 1.0" );
    }
    ...
}
```

Chapter 9: Advanced Low-Power Modeling
Switch Cell Modeling

```
cell ( Complex(CG_Switch) ) {
    ...
    switch_cell_type : coarse_grain;
    pg_pin ( VDD ) {
        pg_type : primary_power;
        voltage_name : VDD;
        direction: input ;
    }
    pg_pin ( VVDD ) {
        pg_type : internal_power;
        direction : output ;
        voltage_name : VVDD;
        switch_function : "SLEEP";
        pg_function : "VDD" ;
    }
    pg_pin ( VSS ) {
        pg_type : primary_ground;
        voltage_name : VSS;
        direction : input ;
    }

/* I/V curve information */
dc_current ( ivt1 ) {
    related_switch_pin : SLEEP;      /* control pin */
    related_pg_pin : VDD;           /* source power pin */
    related_internal_pg_pin : VVDD; /* drain internal power pin*/
    values(    "0.010, 0.020, 0.030, 0.040, 0.050", \
              "0.011, 0.021, 0.031, 0.041, 0.051", \
              "0.012, 0.022, 0.032, 0.042, 0.052", \
              "0.013, 0.023, 0.033, 0.043, 0.053", \
              "0.014, 0.024, 0.034, 0.044, 0.054");
}

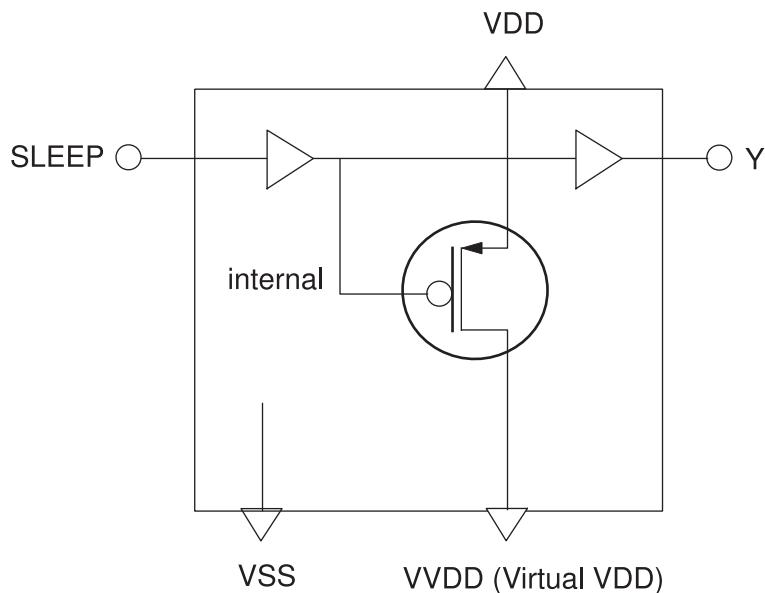
pin ( SLEEP ) {
    direction : input;
    switch_pin : true;
    capacitance: 1.0;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    ...
}
...
pin ( Y ) {
    direction : output;
    function : "SLEEP";
    related_power_pin : VDD;
    related_ground_pin : VSS;
    power_down_function : "!VDD + VSS";
    timing() {
        related_pin : SLEEP;
        ...
    }
} /* end pin group */
```

```
    } /* end cell group*/  
} /* end library group*/
```

Complex Coarse-Grain Switch Cell With an Internal Switch Pin

[Figure 82](#) and the example that follows it show a complex coarse-grain switch cell with an internal switch pin.

Figure 82 Complex Coarse-Grain Switch Cell With an Internal Switch Pin



```
library (Complex(CG_lib) {  
    ...  
    current_unit : 1mA;  
    ...  
  
    voltage_map(VDD, 1.0);  
    voltage_map(VVDD, 0.8);  
    voltage_map(VSS, 0.0);  
  
    operating_conditions(XYZ) {  
        process : 1.0;  
        voltage : 1.0;  
        temperature : 25.0;  
    }  
    default_operating_conditions : XYZ;  
  
    lu_table_template ( ivt1 ) {  
        variable_1 : input_voltage;  
        variable_2 : output_voltage;
```

Chapter 9: Advanced Low-Power Modeling
Switch Cell Modeling

```
index_1 ( "0.1, 0.2, 0.4, 0.8, 1.0" );
index_2 ( "0.1, 0.2, 0.4, 0.8, 1.0" );
}
...

cell (COMPLEX_HEADER_WITH_INTERNAL_SWITCH_PIN) {
    cell_footprint : complex_mtpmos;
    area : 1.0;
    switch_cell_type : coarse_grain;
pg_pin(VDD) {
    voltage_name : VDD;
    pg_type : primary_power;
    direction : input;
}
pg_pin(VVDD) {
    voltage_name : VVDD;
    pg_type : internal_power;
    direction : output;
    switch_function : "SLEEP" ;
}
pg_pin(VSS) {
    voltage_name : VSS;
    pg_type : primary_ground;
    direction : input;
}

dc_current(ivt1) {
    related_switch_pin : internal;
    related_pg_pin : VDD;
    related_internal_pg_pin : VVDD;
    values(    "0.010, 0.020, 0.030, 0.040, 0.050", \
              "0.011, 0.021, 0.031, 0.041, 0.051", \
              "0.012, 0.022, 0.032, 0.042, 0.052", \
              "0.013, 0.023, 0.033, 0.043, 0.053", \
              "0.014, 0.024, 0.034, 0.044, 0.054");
}

pin(SLEEP) {
    switch_pin : true;
    direction : input;
    capacitance : 1.0;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    ...
}

pin(Y) {
    direction : output;
    function : "SLEEP";
    related_power_pin : VDD;
    related_ground_pin : VSS;
    power_down_function : "!VDD + VSS";
```

```

        timing() {
            related_pin : "SLEEP"
            ...
        }
    } /* end pin group */

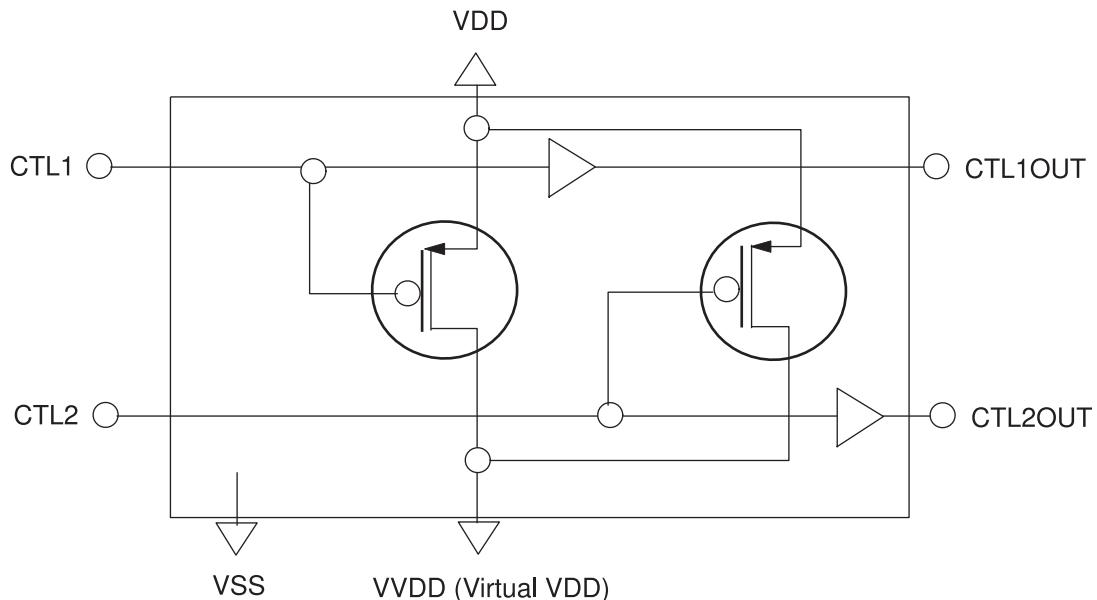
pin(internal) {
    direction : internal;
    timing() {
        related_pin : "SLEEP"
        ...
    }
} /* end pin group */
} /* end cell group */
} /* end library group */

```

Complex Coarse-Grain Switch Cell With Parallel Switches

[Figure 83](#) and the example that follows it show a complex coarse-grain switch cell with two parallel switches.

Figure 83 Complex Coarse-Grain Switch Cell With Two Parallel Switches



```
library (Complex_CG_lib) {  
    ...  
    current_unit : 1mA;  
    ...  
  
    voltage_map(VDD, 1.0);
```

Chapter 9: Advanced Low-Power Modeling
Switch Cell Modeling

```
voltage_map (VVDD, 0.8);
voltage_map (VSS, 0.0);

operating_conditions (XYZ) {
    process : 1.0;
    voltage : 1.0;
    temperature : 25.0;
}
default_operating_conditions : XYZ;

lu_table_template ( ivt1 ) {
variable_1 : input_voltage;
variable_2 : output_voltage;
index_1 ( "0.1, 0.2, 0.4, 0.8, 1.0" );
index_2 ( "0.1, 0.2, 0.4, 0.8, 1.0" );
}
...
cell (COMPLEX_HEADER_WITH_TWO_PARALLEL_SWITCHES) {
    cell_footprint : complex_mtpmos;
    area : 1.0;
    switch_cell_type : coarse_grain;

pg_pin(VDD) {
    voltage_name : VDD;
    pg_type : primary_power;
    direction : input;
}
pg_pin(VVDD) {
    voltage_name : VVDD;
    pg_type : internal_power;
    direction : output;
    switch_function : "CTL1 & CTL2" ;
}
pg_pin(VSS) {
    voltage_name : VSS;
    pg_type : primary_ground;
    direction : input;
}

dc_current(ivt1) {
    related_switch_pin : CTL1;
    related_pg_pin : VDD;
    related_internal_pg_pin : VVDD;
    values( "0.010, 0.020, 0.030, 0.040, 0.050", \
            "0.011, 0.021, 0.031, 0.041, 0.051", \
            "0.012, 0.022, 0.032, 0.042, 0.052", \
            "0.013, 0.023, 0.033, 0.043, 0.053", \
            "0.014, 0.024, 0.034, 0.044, 0.054");
}
dc_current(ivt1) {
    related_switch_pin : CTL2;
}
```

Chapter 9: Advanced Low-Power Modeling
Switch Cell Modeling

```
related_pg_pin : VDD;
related_internal_pg_pin : VVDD;
values( "0.010, 0.020, 0.030, 0.040, 0.050", \
        "0.011, 0.021, 0.031, 0.041, 0.051", \
        "0.012, 0.022, 0.032, 0.042, 0.052", \
        "0.013, 0.023, 0.033, 0.043, 0.053", \
        "0.014, 0.024, 0.034, 0.044, 0.054");
}

pin(CTL1) {
    switch_pin : true;
    direction : input;
    capacitance : 1.0;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    ...
}

pin(CTL2) {
    switch_pin : true;
    direction : input;
    capacitance : 1.0;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    ...
}

pin(CTL1OUT) {
    direction : output;
    function : "CTL1";
    related_power_pin : VDD;
    related_ground_pin : VSS;
    power_down_function : "!VDD + VSS";
    timing() {
        related_pin : "CTL1"
        ...
    }
} /* end pin group */
pin(CTL2OUT) {
    direction : output;
    function : "CTL1";
    related_power_pin : VDD;
    related_ground_pin : VSS;
    power_down_function : "!VDD + VSS";
    timing() {
        related_pin : "CTL2"
        ...
    }
} /* end pin group */
} /* end cell group */
} /* end library group */
```

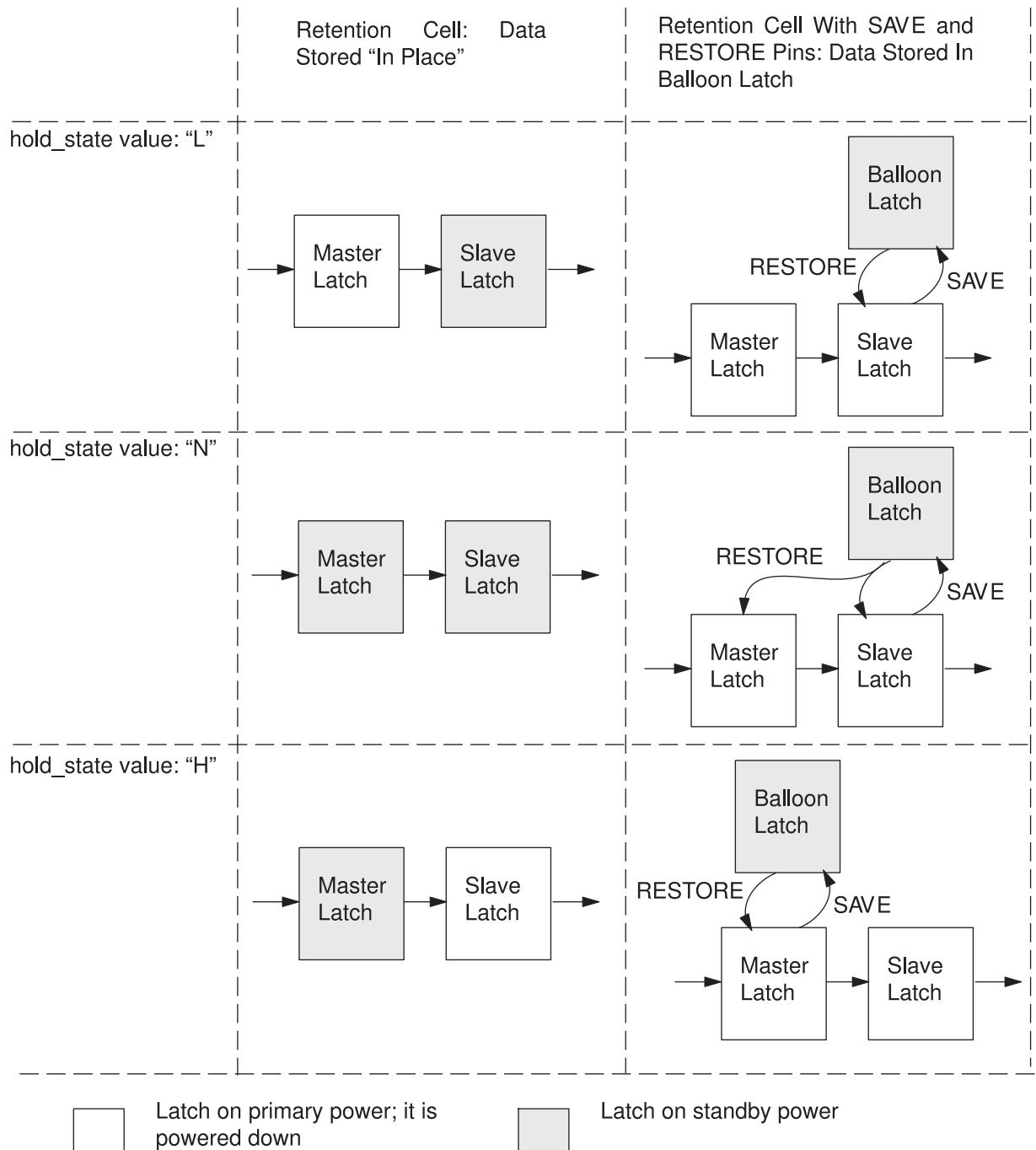
Retention Cell Modeling

Some elements of an electronic design can store the logic state of the design. This is required for the design to wake up in the same state in which it was shut down. The retention cell is one such design element.

Retention cells are sequential cells that hold their state when the power supply is shut down and restore this state when the power is brought up again. The retention cell or register consists of a main register and a shadow register that has a different power supply. The power supply to the shadow register is always powered on to maintain the memory of the state. Therefore, the shadow register has high threshold-voltage transistors to reduce the leakage power. The main register has low threshold-voltage transistors for performance during the normal operation of the retention cell.

Retention cells are broadly classified into the store-in-place and balloon structures. [Figure 84](#) shows the two main retention cell structures. The store-in-place structure includes a master-slave latch where either the slave or the master latch stores the state of the cell when the master-slave latch is shut down. In this structure, the master-slave latch implements the main register while the latch that stores the state of the cell implements the shadow register. The balloon structure includes a flip-flop or master-slave latch and a balloon latch with a control logic that stores the state of the cell when the master-slave latch is shut down. In this structure, the master-slave latch implements the main register and the balloon latch implements the shadow register. The control logic includes signals, such as save and restore signals that control the data storage in the shadow register and the data transfer between the shadow register and the main register.

Figure 84 Retention Cell Structures



Modes of Operation

A retention cell has two modes of operation: *normal mode* and *retention mode*. The retention mode has three stages: *save event*, *sleep mode*, and *restore event*. For example, for the balloon structure, the master-slave latch operates in normal mode and the balloon latch operates in retention mode. The master-slave latch is considered to be edge-triggered. The normal and retention modes are described as follows:

- normal mode

In this mode, the retention cell is fully powered on and the master-slave latch operates normally. The balloon latch does not add to the load at the output of the retention cell.

- retention mode

- save event

During this event, the current state of the master-slave latch is saved before the power to the latch is shut down. The balloon latch is considered to be edge-triggered during the save event. The trailing edge of the save control signal is the triggering edge.

- sleep mode

In this mode, the master-slave latch is shut down.

- restore event

During the restore event, a wake-up signal activates the master-slave latch and the data stored in the balloon latch is fed back to the master-slave latch. The state of the master-slave latch is restored just after the power is restored and before the retention cell operation returns to normal mode. The balloon latch is also considered to be edge-triggered during the restore event. The two methods to restore the state are:

- The leading edge of the restore signal is the triggering edge for the balloon latch. The master-slave latch becomes transparent, that is, it does not latch the data. However, it outputs the same state that was saved in the balloon latch.
- The trailing edge is the triggering edge for the balloon latch. The data is fully restored from the balloon latch and the flip-flop starts operating normally. In this method, the restored data is available for a shorter time.

Retention Cell Modeling Syntax

The following syntax shows the modeling of retention cells. The `reference_input` attribute defines the connectivity information for the input pins based on the `reference_pin_names` variable. The `reference_pin_names` variable specifies the

internal reference input nodes used within the `ff`, `latch`, `ff_bank`, and `latch_bank` groups.

The sequential components of a retention cell are defined by using the flip-flop and latch syntax. The syntax to model the scan retention cells is identical to the syntax to model the scan sequential components. All scan retention cell functional models have a regular cell function that includes the scan pins as part of the function, while the `test_cell` group models the nonscan functionality of the retention cells with the scan pins that have been specified with the `signal_type` attributes.

```
cell(cell_name) {
    retention_cell : retention_cell_style;

    pg_pin (primary_power_name) {
        voltage_name : primary_power_name;
        pg_type : primary_power;
    }
    pg_pin (primary_ground_name) {
        voltage_name : primary_ground_name;
        pg_type : primary_ground;
    }
    pg_pin (backup_power_name) {
        voltage_name : backup_power_name;
        pg_type : backup_power;
    }
    pg_pin (backup_ground_name) {
        voltage_name : backup_ground_name;
        pg_type : backup_ground;
    }
    pin(pin_name) {
        retention_pin(pin_class, disable_value);
        related_ground_pin : backup_ground;
        related_power_pin : backup_power;
        save_action : L|H|R|F;
        restore_action : L|H|R|F;
        restore_edge_type : edge_trigger | leading | trailing;
        ...
    }
    ...
    retention_condition() {
        power_down_function: "Boolean_function";
        required_condition: "Boolean_function";
    }
    clock_condition() {
        clocked_on : "Boolean_expression";
        required_condition : "Boolean_expression";
        hold_state : L|H|N ;
        clocked_on_also : "Boolean_expression";
        required_condition_also : "Boolean_expression";
        hold_state_also : L|H|N;
    }
    preset_condition() {
```

Chapter 9: Advanced Low-Power Modeling
Retention Cell Modeling

```
    input : "Boolean_expression";
    required_condition : "Boolean_expression" ;
}
clear_condition() {
    input : "Boolean_expression" ;
    required_condition : "Boolean_expression" ;
}
pin(pin_name) {
    direction : inout | output | internal;
    function : Boolean_equation_with_internal_node_name;
    reference_input : pin_names;
    ...
}
bus(bus_name) {
    bus_type : bus_type_name;
    direction : inout | output | internal;
    function : Boolean_equation_with_internal_node_name;
    reference_input : pin_names;
    ...
}
ff (["reference_pin_names",] variable1, variable2 ) {
    power_down_function : "Boolean_expression" ;
    ...
}
latch (["reference_pin_names",] variable1, variable2 ) {
    power_down_function : "Boolean_expression";
    ...
}
ff_bank (["reference_pin_names",] variable1, variable2, bits) {
    power_down_function : "Boolean_expression";
    ...
}
latch_bank (["reference_pin_names",] variable1, variable2, bits) {
    power_down_function : "Boolean_expression";
    ...
}
...
statetable (...) {
    power_down_function : "Boolean_expression";
    ...
}
...
}
```

Cell-Level Attributes, Groups, and Variables

This section describes the cell-level attributes, groups, and variables for retention cell modeling.

retention_cell Simple Attribute

The `retention_cell` attribute identifies the type of the retention cell or register. For a given cell, there can be multiple types of retention cells that have the same function in normal mode but different sleep signals, wake signals, or clocking schemes. For example, if a D flip-flop supports two retention strategies, such as `type1` (where the data is transferred when the clock is low) and `type2` (where the data is transferred when the clock is high), the values of the `retention_cell` attribute are different, such as `DFF_type1` and `DFF_type2`.

ff, latch, ff_bank, and latch_bank Groups

The `ff`, `latch`, `ff_bank`, and `latch_bank` groups define sequential blocks. Define these groups at the cell level. You can specify one or more of these groups within a `cell` group.

retention_condition Group

The `retention_condition` group is a group of attributes that specify the conditions for the retention cell to hold its state during the retention mode. The `retention_condition` group includes the `power_down_function` and `required_condition` attributes.

power_down_function Attribute

The `power_down_function` attribute specifies the Boolean condition for the retention cell to be powered down—that is, the primary power to the cell is shut down. When this Boolean condition evaluates to true, it triggers the evaluation of the control input conditions specified by the `required_condition` attribute.

required_condition Attribute

The `required_condition` attribute specifies the control input conditions during the retention mode. For example, in the figure in [Retention Cell With Multiple latch Groups](#), the retention signal, RET, is low during the retention mode. These conditions are checked when the Boolean condition specified by the `power_down_function` attribute evaluates to true. If these conditions are not met, the cell is considered to be in an illegal state.

Note:

Within the `retention_condition` group, the `power_down_function` attribute by itself does not specify the retention mode of the cell. The conditions specified by the `required_condition` attribute ensure that the retention control pin is in the correct state when the primary power to the cell is shut down.

clock_condition Group

The `clock_condition` group is a group of attributes that specify the conditions for correct signal during clock-based events. The `clock_condition` group includes two classes of attributes: attributes without the `_also` suffix and attributes with the `_also` suffix. These are similar to the `clocked_on` and `clocked_on_also` attributes of the `ff` group.

clocked_on and clocked_on_also Attributes

The `clocked_on` and `clocked_on_also` attributes specify the active edge of the clock signal. The Boolean expression of the `clocked_on` attribute must be identical to the one specified in the `clocked_on` attribute of the corresponding `ff` or `ff_bank` group.

For example, for a master-slave latch, use the `clocked_on` attribute on the clock to the master latch and the `clocked_on_also` attribute on the clock to the slave latch.

Note:

A single-stage flip-flop or latch does not use the `clocked_on_also` attribute.

required_condition and required_condition_also Attributes

The `required_condition` and `required_condition_also` attributes specify the input conditions during the active edge of the clock signal. These conditions are checked, respectively, at the values specified by the `clocked_on` and `clocked_on_also` attributes. If any one of the conditions are not met, the cell is considered to be in an illegal state.

For example, when the `required_condition` attribute is checked at the rising edge of the clock signal specified by the `clocked_on` attribute, the `required_condition_also` attribute is also checked at the rising edge of the clock signal specified by the `clocked_on_also` attribute. If the `clocked_on_also` attribute is not specified, the `required_condition_also` attribute is checked at the falling edge of the clock signal specified by the `clocked_on` attribute. This condition is checked when the slave latch is in the hold state.

hold_state and hold_state_also Attributes

The `hold_state` and `hold_state_also` attributes specify the values for the Boolean expressions of the `clocked_on` and `clocked_on_also` attributes during the retention mode. The valid values are `L` (low), `H` (high), or `N` (no change).

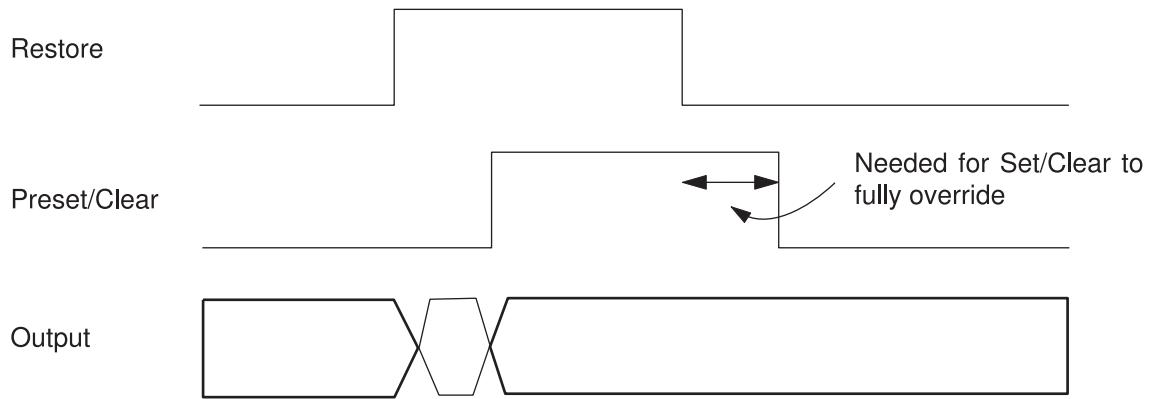
If the data is restored to both the master and slave latches, the value of the `hold_state` attribute is `N`. If the data is restored to the slave latch, the value of the `hold_state` attribute is `L`.

preset_condition and clear_condition Groups

The `preset_condition` and `clear_condition` groups contain attributes that specify the conditions, respectively, for the present and clear signals in the normal mode of the retention cell.

The asynchronous control signals, preset and clear, have higher priority over the clock signal. However, these signals do not control the balloon latch. Therefore, if the preset or clear signals are asserted during the restore event, these signals need to be active for a time longer than the restore event so that the master-slave latch content is successfully overwritten, as shown in [Figure 85](#). Therefore, the preset and clear signals must be checked at the trailing edge.

Figure 85 Preset and Clear Signal Overlaps With Restore



input Attribute

The `input` attribute specifies how the preset or clear control signal is asserted. The Boolean expression of the `input` attribute must be identical to one specified in the `input` attribute of the `ff` group.

required_condition Attribute

The `required_condition` attribute specifies the input condition during the active edge of the preset or clear signal. The `required_condition` attribute is checked at the trailing edge of the preset or clear signal. When the input condition is not met, the cell is in an illegal state.

reference_pin_names Variable

The `reference_pin_names` variable specifies the input nodes that are used for internal reference within the `ff`, `latch`, `ff_bank`, or `latch_bank` groups. If you do not specify the `reference_pin_names` variable, the node names in the `ff`, `latch`, `ff_bank`, or `latch_bank` groups are considered to be the actual pin or bus names of the cell.

variable1 and variable2 Variables

The `variable1` and `variable2` variables define the output nodes for internal reference.

The values of the `variable1` and `variable2` variables in the `ff`, `latch`, `ff_bank`, or `latch_bank` groups must be unique for a cell.

bits Variable

The `bits` variable defines the width of the `ff_bank` and `latch_bank` component.

Pin-Level Attributes

This section describes the pin-level attributes for retention cell modeling.

retention_pin Complex Attribute

The `retention_pin` attribute identifies the retention pins of a retention cell. In the normal mode, the retention pins are disabled.

The `retention_pin` attribute has the following arguments:

- Pin class

The values are

- `restore`

Restores the state of the cell.

- `save`

Saves the state of the cell.

- `save_restore`

Saves and restores the state of the cell. When a single pin in a retention cell performs both the `save` and `restore` operations, specify the `retention_pin` attribute with the `save_restore` value. The retention pin is in `save` mode when it saves the data. The retention pin is in `restore` mode when it restores the data.

function Attribute

The `function` attribute maps an output, inout, or internal pin to a corresponding internal node or a value of the `variable1` or `variable2` variable in a `ff`, `latch`, `ff_bank`, or `latch_bank` group. The `function` attribute also accepts a Boolean equation containing the `variable1` or `variable2` variable and other input, inout, or internal pins. Define the `function` attribute in a `pin` or `bus` group.

reference_input Attribute

The `reference_input` attribute specifies the input pins that map to the reference pin names of the corresponding `ff`, `latch`, `ff_bank`, or `latch_bank` group. For each inout, output, or internal pin, the `variable1` or `variable2` value specified in the function statement determines the corresponding `ff`, `latch`, `ff_bank`, or `latch_bank` group. You can define the `reference_input` attribute in a `pin` or `bus` group.

save_action and restore_action Attributes

The `save_action` and `restore_action` attributes specify where the save and restore events occur with respect to the save and restore control signals, respectively. Valid values are `L` (low), `H` (high), `R` (rise), and `F` (fall). The `L` or `H` values indicate that the data is actually stored in the balloon latch at the trailing edge of the save signal. The `R` or `F` values indicate that this edge of the restore signal specifies when the data stored in the balloon latch is available at the output of the master-slave latch.

restore_edge_type Attribute

The `restore_edge_type` attribute specifies the type of the edge of the restore signal where the output of the master-slave latch is restored. The `restore_edge_type` attribute supports the following edge types: `edge_trigger`, `leading`, and `trailing`. The default edge type is `leading`. This is because the other control signals, such as clock, preset, and clear are of `leading` edge type, that is, they make the data available at the output of the master-slave latch when the latch is transparent.

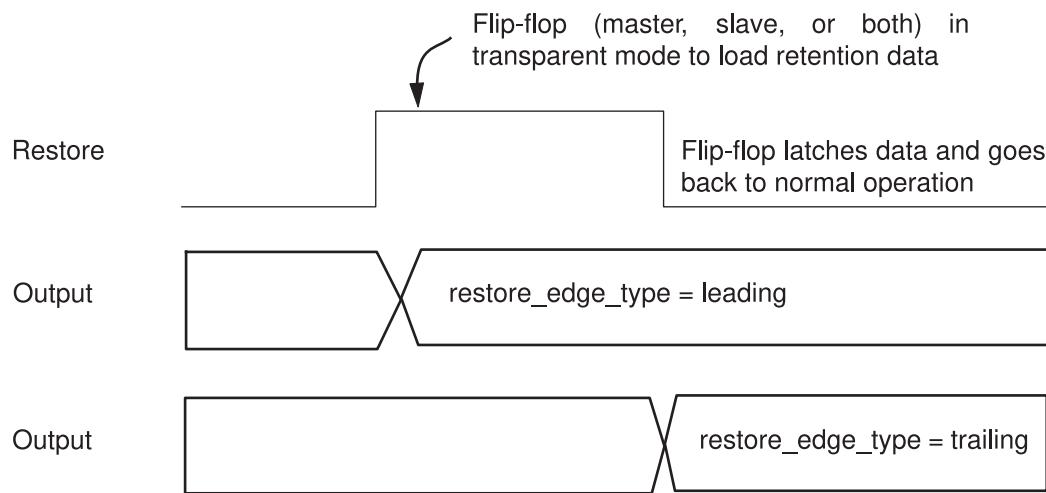
The `edge_trigger` type indicates that the output of the master-slave latch is restored at the edge of the restore signal. The master-slave latch resumes normal operation immediately thereafter.

The `leading` edge type indicates that the output of the master-slave latch is restored at the leading edge of the restore signal. The master-slave latch resumes normal operation after the trailing edge of the restore signal.

The `trailing` edge type indicates that the output of the master-slave latch is restored at the trailing edge of the restore signal. The master-slave latch resumes normal operation after the trailing edge of the restore signal.

[Figure 86](#) shows the valid data windows when the `restore_edge_type` attribute is set to the `leading` and `trailing` edge types.

Figure 86 Valid Data Window for leading and trailing Values of the `restore_edge_type` Attribute



save_condition and restore_condition Attributes

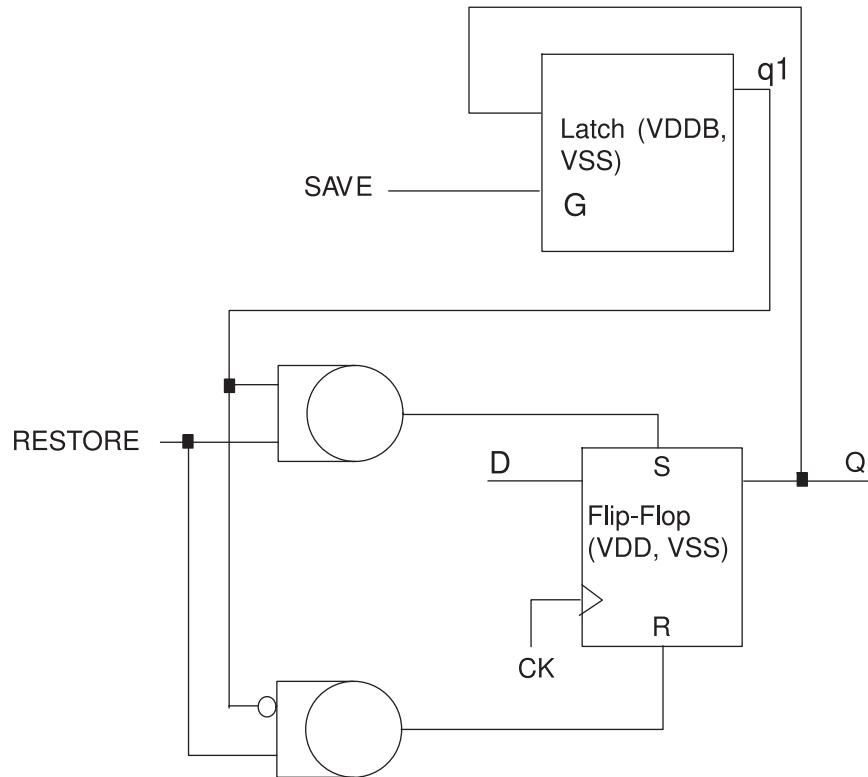
The `save_condition` and `restore_condition` attributes specify the input conditions during the `save` and `restore` events respectively. These conditions are respectively checked at the values of the `save_action` and `restore_action` attributes. If any one of the conditions are not met, the cell is considered to be in an illegal state.

Retention Cell Model Examples

Retention Cell With Balloon Latch

[Figure 87](#) shows a retention cell structure that is defined using the `ff` or `latch` group. The cell has a balloon latch structure to store the data when the main flip-flop is shut down. The data is transferred to the output of the retention cell at the rising edge of the clock signal, CK. The `SAVE` and `RESTORE` pins save and restore the data.

Figure 87 Retention Cell Model Schematic With Balloon Latch



Example 104 Retention Cell With Balloon Latch

```
library (BALLOON_RET_FLOP) {
    delay_model : table_lookup;
    input_threshold_pct_rise : 50 ;
    input_threshold_pct_fall : 50 ;
    output_threshold_pct_rise : 50 ;
    output_threshold_pct_fall : 50 ;
    slew_lower_threshold_pct_fall : 30.0 ;
    slew_lower_threshold_pct_rise : 30.0 ;
    slew_upper_threshold_pct_fall : 70.0 ;
    slew_upper_threshold_pct_rise : 70.0 ;

    time_unit : "1ns";
    voltage_unit : "1V";
    current_unit : "1uA";
    pulling_resistance_unit : "1kohm";
    capacitive_load_unit (0.1,ff);

    voltage_map (VDD, 1.0);
    voltage_map (VDDB, 0.9);
    voltage_map (VSS, 0.0);
```

```
nom_process : 1.0;
nom_temperature : 25.0;
nom_voltage : 1.1;

operating_conditions(typ) {
    process : 1.0 ;
    temperature : 25 ;
    voltage : 1.1 ;
    tree_type : "balanced_tree" ;
}
default_operating_conditions : typ;
wire_load("05*05") {
    resistance : 1.0 ;
    capacitance : 25 ;
    area : 1.1 ;
    slope : "balanced_tree" ;
    fanout_length(1,0.39);
}

cell (balloon_ret_cell) {

    retention_cell : retdiff;
    area : 1.0 ;

    pg_pin(VDD) {
        voltage_name : VDD;
        pg_type : primary_power;
    }
    pg_pin(VSS) {
        voltage_name : VSS;
        pg_type : primary_ground;
    }
    pg_pin(VDDB) {
        voltage_name : VDDB;
        pg_type : backup_power;
    }

    ff ( Q1, QN1 ) {
        clocked_on : " CK ";
        next_state : " D ";
        clear : " RESTORE * !Q2 ";
        preset : " RESTORE * Q2 ";
        clear_preset_var1 : "L";
        clear_preset_var2 : "H";
        power_down_function : "!VDD+VSS";
    /* Latch 1 is powered by primary power supply */
    }
    latch("Q2", "QN2") {
        enable : " SAVE ";
        data_in : "Q";
        power_down_function : "!VDDB+VSS";
    /* Latch 2 is powered by backup power supply */
    }
}
```

```
        }
    clock_condition() {
        clocked_on : "CK";
        required_condition : "RESTORE";
        hold_state : L;
    }

    pin(RESTORE) {
        direction : input;
        capacitance : 0.1;
        related_power_pin : VDDB;
        related_ground_pin : VSS;
        retention_pin	restore, "0";
        restore_action : "H";
        restore_condition : "!CK";
        restore_edge_type : "leading";
    }
    pin(SAVE) {
        direction : input;
        capacitance : 0.1;
        related_power_pin : VDDB;
        related_ground_pin : VSS;
        retention_pin	save, "0";
        save_action : "H";
        save_condition : "!CK";
    }
    pin(D) {
        direction : input;
        capacitance : 0.1;
        related_power_pin : VDD;
        related_ground_pin : VSS;
    }
    pin(CK) {
        direction : input;
        clock : true;
        capacitance : 0.1;
        related_power_pin : VDD;
        related_ground_pin : VSS;
    }
    pin(Q) {
        direction : output;
        function : "Q1";
        related_power_pin : VDD;
        related_ground_pin : VSS;
        timing() {
            related_pin : "CK";
            timing_type : rising_edge;
            cell_rise(scalar) { values ( "0.1"); }
            rise_transition(scalar) { values ( "0.1"); }
            cell_fall(scalar) { values ( "0.1"); }
            fall_transition(scalar) { values ( "0.1"); }
        }
    }
}
```

```
retention_condition() {
    power_down_function : "!VDD+VSS";
    required_condition: "!SAVE";
}
/*
pin(q1) {
    direction : internal;
    function : "Q2";
}
*/
} /* End cell group */
} /* End Library group */
```

Retention Cell With Multiple latch Groups

Figure 88 shows a schematic of a basic retention cell. The state of the cell is stored inside the slave latch that is powered by the backup power VDDB. Figure 89 shows the valid values of the input control signals when the cell is in retention mode.

In the figure, and in Example 105, the reference cells are defined by using the latch group syntax. The connectivity information for the input pins is specified in the `reference_input` attribute that is mapped to the reference pin names of the corresponding latch group.

Figure 88 Simple Retention Cell Model Schematic

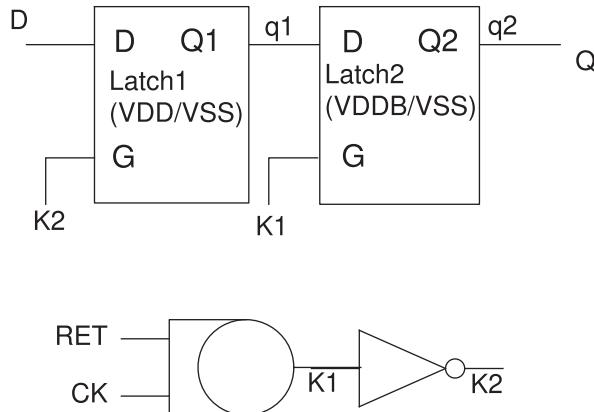
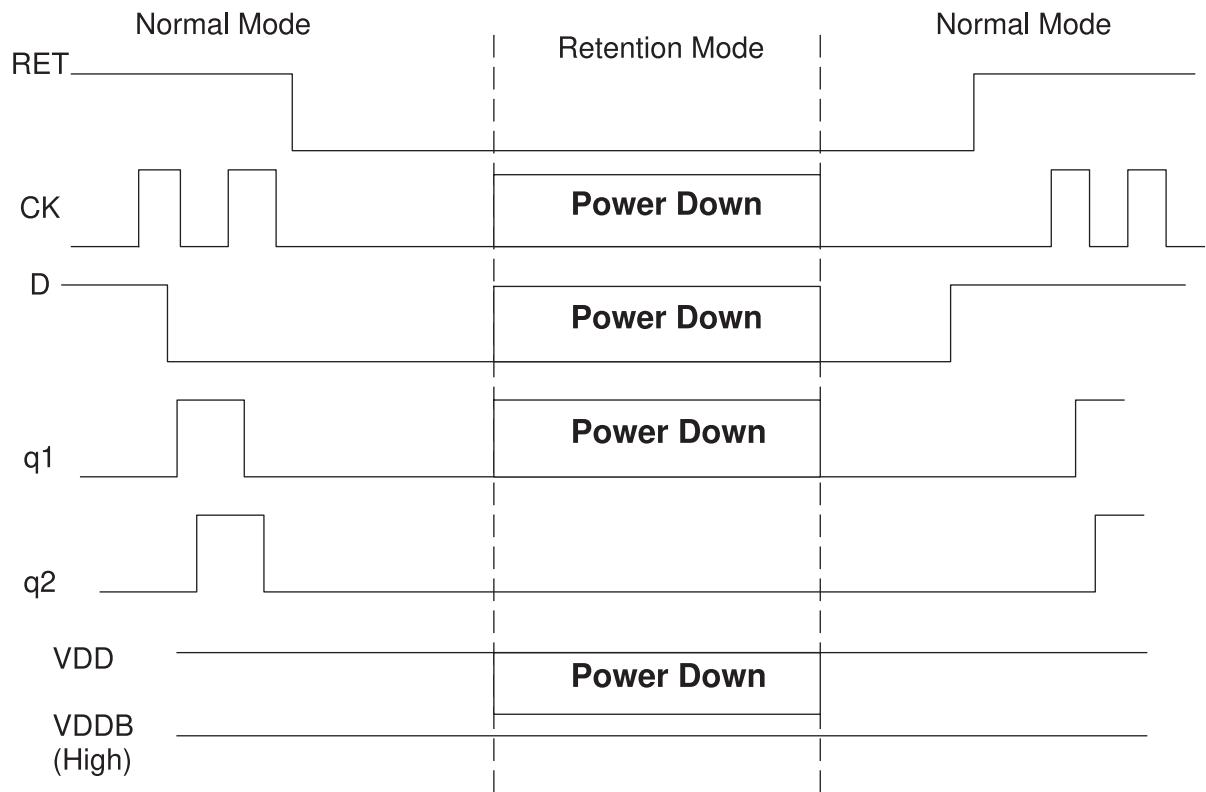


Figure 89 Valid RET and CK Signal Values in the Retention Mode



Note:

To retain the stored data, the retention signal, RET, must be low during the retention mode. This condition is specified by the `required_condition` attribute of the `retention_condition` group.

Example 105 Retention Cell Model Example Using Multiple Latch Group

```
library (RET_FLOP) {
  delay_model : table_lookup;

  input_threshold_pct_rise      : 50 ;
  input_threshold_pct_fall      : 50 ;
  output_threshold_pct_rise     : 50 ;
  output_threshold_pct_fall     : 50 ;
  slew_lower_threshold_pct_fall : 30.0 ;
  slew_lower_threshold_pct_rise : 30.0 ;
  slew_upper_threshold_pct_fall : 70.0 ;
  slew_upper_threshold_pct_rise : 70.0 ;

  time_unit                     : "1ns";
  voltage_unit                  : "1V";
```

Chapter 9: Advanced Low-Power Modeling
Retention Cell Modeling

```
current_unit : "1uA";
pulling_resistance_unit : "1kohm";
capacitive_load_unit (0.1,ff);

voltage_map (VDD, 1.0);
voltage_map (VDDB, 0.9);
voltage_map (VSS, 0.0);

nom_process : 1.0;
nom_temperature : 25.0;
nom_voltage : 1.1;

operating_conditions(xyz) {
    process : 1.0 ;
    temperature : 25 ;
    voltage : 1.1 ;
    tree_type : "balanced_tree" ;
}
default_operating_conditions : xyz;
... /* Other library level attributes and groups */

cell (retention_flip_flop) {
    retention_cell : retdiff;
    area : 1.0;

    pg_pin (VDD) {
        voltage_name : VDD;
        pg_type : primary_power;
    }
    pg_pin (VSS) {
        voltage_name : VSS;
        pg_type : primary_ground;
    }
    pg_pin (VDDB) {
        voltage_name : VDDB;
        pg_type : backup_power;
    }
    latch ( Q1, QN1 ) {
        enable : "(RET * CK)'";
        data_in : "D";
        power_down_function : "!VDD+VSS";
    /* Latch1 is powered by primary power supply */
    }
    latch("Q2", "QN2") {
        enable : "RET * CK";
        data_in : "Q1";
        power_down_function : "!VDDDB+VSS";
    /* Latch2 is powered by backup power supply */
    }
    clock_condition() {
        clocked_on : "CK";
        required_condition : "RET";
    }
}
```

Chapter 9: Advanced Low-Power Modeling
Retention Cell Modeling

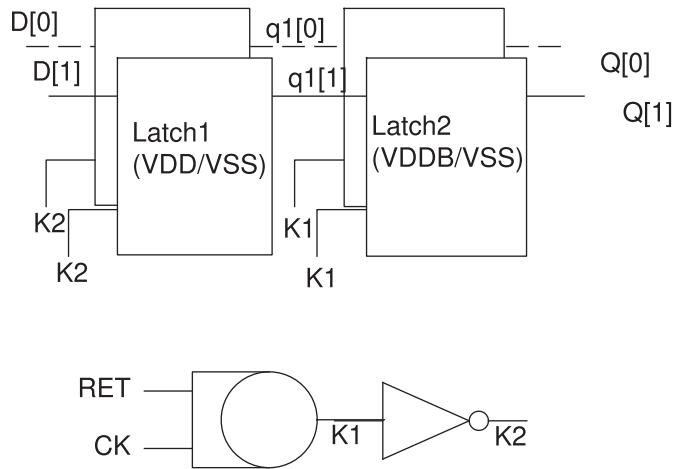
```
hold_state : "L";

pin (RET) {
    direction : input;
    capacitance : 0.1;
    related_power_pin : VDDB;
    related_ground_pin : VSS;
    retention_pin(save_restore, "1");
    save_action : "L";
    restore_action : "H";
    save_condition : "!CK";
    restore_condition : "!CK";
    restore_edge_type : "leading";
}
pin(D) {
    direction : input;
    capacitance : 0.1;
    related_power_pin : VDD;
    related_ground_pin : VSS;
}
pin(CK) {
    direction : input;
    clock : true;
    capacitance : 0.1;
    related_power_pin : VDD;
    related_ground_pin : VSS;
}
pin (Q) {
    direction : output;
    function : "Q2";
    related_power_pin : VDD;
    related_ground_pin : VSS;
    timing() {
        related_pin : "CK";
        timing_type : rising_edge;
        cell_rise(scalar) { values ( "0.1"); }
        rise_transition(scalar) { values ( "0.1"); }
        cell_fall(scalar) { values ( "0.1"); }
        fall_transition(scalar) { values ( "0.1"); }
    }
}
retention_condition() {
    power_down_function : "!VDD+VSS";
    required_condition: "!RET";
}
} /* End Cell group */
} /* End Library group */
```

Retention Cell With Multiple latch_bank Groups

Figure 90 and Example 106 show a retention cell structure that is defined using the `latch_bank` group that has multibit parallel inputs and output buses in the datapath and the clock path.

Figure 90 Multibit (2-bit) Retention Cell Model Schematic



Example 106 Retention Cell Model Example Using Multiple `latch_bank` Groups

```
library (RET_FLOP_BANK) {  
  
    input_threshold_pct_rise : 50 ;  
    input_threshold_pct_fall : 50 ;  
    output_threshold_pct_rise : 50 ;  
    output_threshold_pct_fall : 50 ;  
    slew_lower_threshold_pct_fall : 30.0 ;  
    slew_lower_threshold_pct_rise : 30.0 ;  
    slew_upper_threshold_pct_fall : 70.0 ;  
    slew_upper_threshold_pct_rise : 70.0 ;  
  
    time_unit : "1ns";  
    voltage_unit : "1V";  
    current_unit : "1uA";  
    pulling_resistance_unit : "1kohm";  
    capacitive_load_unit (0.1,ff);  
  
    voltage_map (VDD, 1.0);  
    voltage_map (VDDB, 0.9);  
    voltage_map (VSS, 0.0);  
  
    nom_process : 1.0;  
    nom_temperature : 25.0;
```

```
nom_voltage : 1.1;

operating_conditions(xyz) {
    process : 1.0 ;
    temperature : 25 ;
    voltage : 1.1 ;
    tree_type : "balanced_tree" ;
}
default_operating_conditions : xyz;

type (bus2) {
    base_type : array;
    data_type : bit;
    bit_width : 2;
    bit_from : 0;
    bit_to : 1;
    downto : false;
}

cell (retention_flip_bank) {
    retention_cell : retdiff_bank;
    area : 1.0;

    pg_pin(VDD) {
        voltage_name : VDD;
        pg_type : primary_power;
    }

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

    pg_pin(VDDB) {
        voltage_name : VDDB;
        pg_type : backup_power;
    }

    latch_bank ("Q1", "QN1", 2) {
        enable : "(RET * CK)'";
        data_in : "SE'*D + SE*SI";
        power_down_function : "!VDD+VSS";
    }
    latch_bank ("Q2", "QN2", 2) {
        enable : " RET * CK ";
        data_in : " q1 ";
        power_down_function : "!VDDB+VSS";
    }
    clock_condition() {
        clocked_on : " CK ";
        required_condition : " RET ";
        hold-state : " L ";
    }
}
```

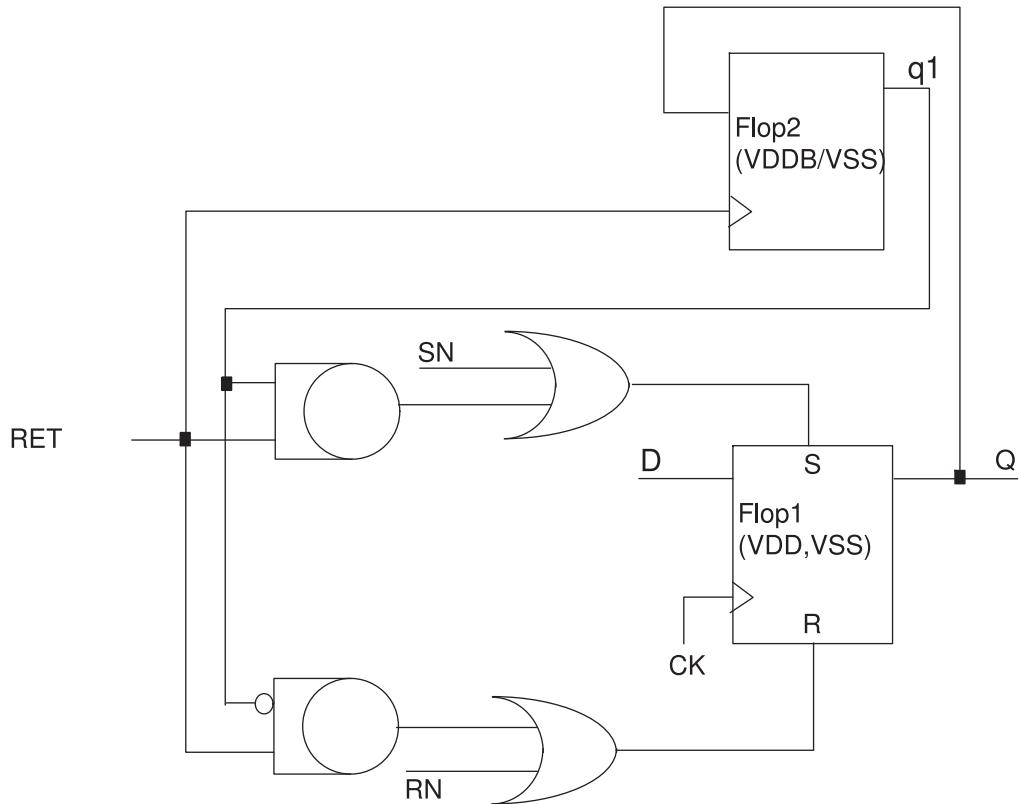
```
bus(D) {
    bus_type : bus2;
    direction : input;
    capacitance : 0.1;
    related_power_pin : VDD;
    related_ground_pin : VSS;
}
bus(Q1) {
    bus_type : bus2;
    direction : internal;
    function : "Q1";
    related_power_pin : VDD;
    related_ground_pin : VSS;
}
bus(Q) {
    bus_type : bus2;
    direction : output;
    function : "Q2";
    related_power_pin : VDD;
    related_ground_pin : VSS;
}
pin(RET) {
    direction : input;
    capacitance : 0.1;
    related_power_pin : VDDB;
    related_ground_pin : VSS;
    retention_pin("restore", 1);
    save_action : "L";
    restore_action : "H";
    save_condition : "!CK";
    restore_condition : "!CK";
    restore_edge_type : "leading";
}
pin(CK) {
    direction : input;
    clock : true;
    capacitance : 0.1;
    related_power_pin : VDD;
    related_ground_pin : VSS;
}
retention_condition() {
    power_down_function : "!VDD+VSS";
    required_condition: "!RET";
}
bus(SI) {
    bus_type : bus2;
    direction : input;
    clock : true;
    capacitance : 0.1;
    related_power_pin : VDD;
    related_ground_pin : VSS;
}
```

```
bus(SE) {  
    bus_type : bus2;  
    direction : input;  
    clock : true;  
    capacitance : 0.1;  
    related_power_pin : VDD;  
    related_ground_pin : VSS;  
}  
cell_leakage_power : 1.000000;  
  
} /* End Cell group */  
} /* End Library group */
```

Retention Cell With Edge-Triggered Balloon Logic

Figure 91 shows a retention cell structure with two `ff` groups. The cell has a balloon latch or flip-flop to store the data when the main flip-flop is shut down. The data is transferred to the output of the main flip-flop, Flop1, on the rising edge of the clock signal, CK, when the asynchronous preset, SN, and clear, RN, are inactive. In retention mode, the retention pin, RET, saves the data into the balloon flip-flop and restores the data from the balloon flip-flop to the output pin of the retention cell. At the rising edge of the RET signal, the data is saved inside the balloon flip-flop, Flop2, and Flop1 is shut down. When the power to Flop1 is brought up and the retention pin, RET, becomes inactive, the data from Flop2 is restored to Flop1 at the falling edge of the RET signal.

Figure 91 Edge-Triggered Retention Cell Model Schematic



Example 107 Retention Cell With Edge-Triggered Balloon Logic

```
library(edge_triggered_retention) {
  delay_model : table_lookup;
  time_unit           : "1ns";
  voltage_unit        : "1V";
  current_unit        : "1uA";
  capacitive_load_unit(0.1,ff);
  default_fanout_load : 1.0;
  default inout pin cap : 1.0;
  default input pin cap : 1.0;
  default output pin cap : 1.0;
  input threshold pct rise   : 50 ;
  input threshold pct fall   : 50 ;
  output threshold pct rise  : 50 ;
  output threshold pct fall  : 50 ;
  slew lower threshold pct fall : 30.0 ;
  slew lower threshold pct rise : 30.0 ;
  slew upper threshold pct fall : 70.0 ;
  slew upper threshold pct rise : 70.0 ;
```

Chapter 9: Advanced Low-Power Modeling

Retention Cell Modeling

```
voltage_map(VDD, 1.0);
voltage_map(VDDB, 1.0);
voltage_map(VSS, 0.0);
    nom_process : 1.0;
    nom_temperature : 25.0;
    nom_voltage : 1.1;
    operating_conditions(xyz) {
        process : 1.0 ;
        temperature : 25 ;
        voltage : 1.1 ;
        tree_type : "balanced_tree" ;

    }
    default_operating_conditions : xyz;
cell (edge_trigger) {
    retention_cell : "edge_trigger";
    ... /* Other cell-level attributes and groups */
    pg_pin (VDD) {
        voltage_name : "VDD";
        pg_type : "primary_power";
    }
    pg_pin (VDDB) {
        voltage_name : "VDDB";
        pg_type : "backup_power";
    }
    pg_pin (VSS) {
        voltage_name : "VSS";
        pg_type : "primary_ground";
    }
    ff ("IQ1" , "IQN1") {
        next_state : "D";
        clocked_on : "CK";
        clear : "RN + (RET * q1')";
        preset : "SN + (RET * q1)";
        clear_preset_var1 : L;
        clear_preset_var1 : H;
        power_down_function : "!VDD + VSS"; /* Flip-Flop "Flop1" is powered
by Primary power supply */
    }
    ff ("IQ2" , "IQN2") {
        next_state : "Q";
        clocked_on : "RET";
        power_down_function : "!VDDS + VSS"; /* Flip-Flop "Flop2" is powered
by Primary power supply */
    }
    clock_condition() {
        clocked_on : "CK"; /* clock must be Low to go into retention mode */
        hold_state : "N"; /* when clock switches (either direction), RET
must
be High */
        condition : "RET";
    }
    clear_condition() {
```

Chapter 9: Advanced Low-Power Modeling
Retention Cell Modeling

```
    input : "!RN"; /* When clear de-asserts, RET must be high to allow
                     Low value to be transferred to Flop1 */

    required_condition : "RET";
}
preset_condition() {
    input : "!SN"; /* When clear de-asserts, RET must be high to allow
                     High value to be transferred to Flop1 */
    required_condition : "RET";
}
retention_condition() {
    power_down_function : "!VDD+VSS";
    required_condition: "!RET";
}
pin (q1) {
    direction : "internal";
    function : "IQ2";
    related_power_pin : "VDD";
    related_ground_pin : "VSS";
    ... /* Other pin-level attributes and groups */
}
pin (CK) {
    direction : "input";
    clock : true;
    capacitance : 1.0;
    related_power_pin : "VDD";
    related_ground_pin : "VSS";

    ... /* Other pin-level attributes and groups */
}
pin (RET) {
    related_power_pin : "VDD";
    related_ground_pin : "VSS";
    capacitance : 1.0;
    direction : "input";
    retention_pin("save_restore",0);
    save_action : "R";
    restore_action : "R";
    save_condition : "!CK";
    restore_condition : "!CK";
    restore_edge_type : "leading";
    ... /* Other pin-level attributes and groups */
}
pin (D) {
    direction : "input";
    related_power_pin : "VDD";
    related_ground_pin : "VSS";
    capacitance : 1.0;
    ... /* Other pin-level attributes and groups */
}
pin (RN) {
    direction : "input";
    related_power_pin : "VDD";
```

```

related_ground_pin : "VSS";
capacitance : 1.0;
... /* Other pin-level attributes and groups */
}
pin (SN) {
    direction : "input";
    related_power_pin : "VDD";

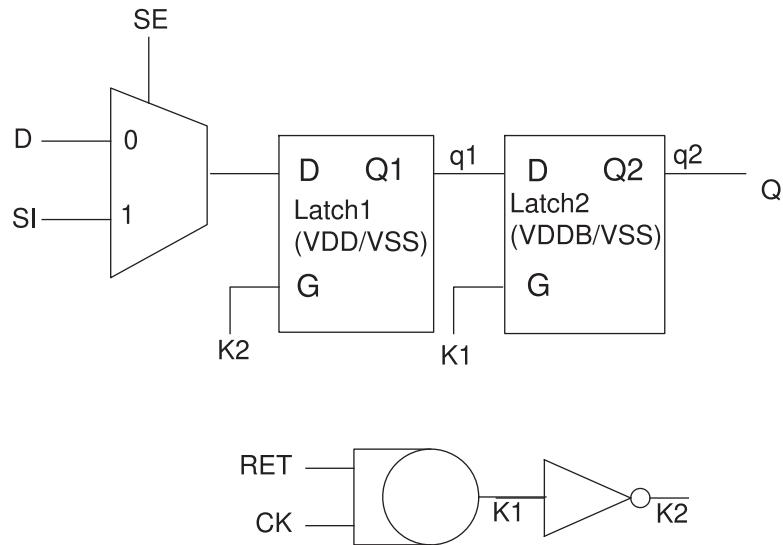
    related_ground_pin : "VSS";
    capacitance : 1.0;
    ... /* Other pin-level attributes and groups */
}
pin (Q) {
    direction : "output";
    function : "IQ1";
    related_power_pin : "VDD";
    related_ground_pin : "VSS";
    ... /* Other pin-level attributes and groups */
} /* End Pin group */
} /* End Cell group */
} /* End Library group */

```

MUX-Scan Retention Cell

[Figure 92](#) and [Example 108](#) show a scan retention cell structure that is defined using the latch group. The cell is the scan version of the retention cell modeled in the example in [Retention Cell With Multiple latch Groups](#). The test_cell group includes only the function of the nonscan version of the retention cell.

Figure 92 MUX-Scan Retention Cell Model Schematic



Example 108 MUX-Scan Retention Cell Model

```
library (Retention_cell_Example) {
    delay_model : table_lookup;
    time_unit : "1ns";
    voltage_unit : "1V";
    current_unit : "1uA";
    capacitive_load_unit (0.1,ff);
    default_fanout_load : 1.0;
    default inout_pin_cap : 1.0;
    default input_pin_cap : 1.0;
    default output_pin_cap : 1.0;
    input_threshold_pct_rise : 50 ;
    input_threshold_pct_fall : 50 ;
    output_threshold_pct_rise : 50 ;
    output_threshold_pct_fall : 50 ;
    slew_lower_threshold_pct_fall : 30.0 ;
    slew_lower_threshold_pct_rise : 30.0 ;
    slew_upper_threshold_pct_fall : 70.0 ;
    slew_upper_threshold_pct_rise : 70.0 ;
    voltage_map (VDD, 1.0);
    voltage_map (VDDB, 0.9);
    voltage_map (VSS, 0.0);

    nom_process : 1.0;
    nom_temperature : 25.0;
    nom_voltage : 1.1;

    operating_conditions(xyz) {
        process : 1.0 ;
        temperature : 25 ;
        voltage : 1.1 ;
        tree_type : "balanced_tree" ;
    }
    default_operating_conditions : xyz;
    ... /* Other library-level attributes */

    cell (scan_retention_cell) {
        retention_cell : my_scan_ret_cell;
        ... /* Other cell-level attributes and groups */
        pg_pin(VDD) {
            voltage_name : VDD;
            pg_type : primary_power;
        }
        pg_pin(VSS) {
            voltage_name : VSS;
            pg_type : primary_ground;
        }
        pg_pin(VDDB) {
            voltage_name : VDDB;
            pg_type : backup_power;
        }
        pin(RET) {
```

Chapter 9: Advanced Low-Power Modeling
Retention Cell Modeling

```
direction : input;
capacitance : 0.1;
related_power_pin : VDDB;
related_ground_pin : VSS;
retention_pin(save_restore, "1");
... /* Other pin-level attributes and groups */
}
pin(D) {
    direction : input;
    capacitance : 0.1;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    ... /* Other pin-level attributes and groups */
}
pin(CK) {
    direction : input;
    clock : true;
    capacitance : 0.1;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    ... /* Other pin-level attributes and groups */
}
pin(Q) {
    direction : output;
    function : "Q2";
    related_power_pin : VDD;
    related_ground_pin : VSS;
    reference_input : "RET CK q1";
    ... /* Other pin-level attributes and groups */
}
pin(q1) {
    direction : internal;
    function : "Q1";
    related_power_pin : VDD;
    related_ground_pin : VSS;
    reference_input : "RET CK SE D SI";
    ... /* Other pin-level attributes and groups */
}
retention_condition() {
    power_down_function : "!VDD+VSS";
    required_condition: "!RET";
}
latch ("p1 p2,p3,p4,p5", "Q1", "QN1") {
    enable : " p1' + p2' ";
    data_in : " p3'*p4 + p3*p5 ";
    power_down_function : "!VDD+VSS"; /* Latch 1 is powered by Primary
power supply */
}
latch ("p1 p2 p3", "Q2", "QN2") {
    enable : " p1 * p2 ";
    data_in : " p3 ";
    power_down_function : "!VDDB+VSS"; /* Latch 2 is powered by Backup
```

```
power supply /*  
}  
test_cell() {  
    pin(SI) {  
        direction : input;  
        signal_type : "test_scan_in";  
    }  
    pin(RET) {  
        direction : input;  
    }  
    pin(D) {  
        direction : input;  
    }  
    pin(SE) {  
        direction : input;  
        signal_type : "test_scan_enable";  
    }  
    pin(CK) {  
        direction : input;  
    }  
    latch ("Q1", "QN1") {  
        enable : " RET' + CK' ";  
        data_in : " D ";  
    }  
  
    latch ("Q2", "QN2") {  
        enable : " RET * CK ";  
        data_in : " q1 ";  
    }  
    pin (q1) {  
        direction : internal;  
        function : "Q1";  
    }  
    pin(Q) {  
        direction : output;  
        signal_type : "test_scan_out";  
        function : "Q2";  
    } /* End Pin group */  
} /* End test_cell group */  
} /* End cell group */  
} /* End Library group */
```

Always-On Cell Modeling

In complex low-power designs, some signals need to be routed through blocks that have been shut down. As a result, a variety of cell categories require “always-on” signal pins. Always-on cells remain powered on by a backup power supply in the region where they are placed even when the main power supply is switched off. The cells also have a secondary backup power pin that supplies the current that is necessary when the main supply is not available.

For tools to recognize always-on cells in the reference library and use them during special always-on synthesis, library models need an attribute that can identify them. The `always_on` attribute identifies always-on cells and pins.

When you run the `read_lib` command, the `always_on` attribute is automatically added to buffer or inverter cells that have input and output pins that are linked to backup power or backup ground PG pins. The following buffers and inverters are automatically identified as always-on cells:

- Cells with one primary power pin, one primary ground pin, and one backup ground pin, if both the input and output signal pins are linked to the primary power and backup ground pins.
- Cells with one primary power pin, one backup power pin, and one primary ground pin, if both the input and output signal pins are linked to the backup power and primary ground pins.
- Cells with one primary power pin, one backup power pin, one primary ground pin, and one backup ground pin if
 - Both the input and output signal pins are linked to the backup power pin and the backup ground pin.
 - Both the input and output signal pins are linked to the primary power pin and the backup ground pin.
 - Both the input and output signal pins are linked to the backup power pin and the primary ground pin.

The `always_on` attribute also identifies always-on input pins so that tools can trace all always-on nets crossing domains that are shut down. Always-on pins are automatically created for the following cells:

- Save and restore pins on retention cells
- Control pins on switch cells
- Enable pins on isolation cells and enable level-shifter cells

The cell library does not require that you specify these cells as always-on cells. If you have other cells that need to be specified as always-on, you can add them to the library cell model.

Always-On Cell Syntax

```
library (library_name) {  
  ...  
  cell (cell_name) {  
    always_on : true;
```

```
...
pin (pin_name) {
    always_on : true;
    ...
}
...
}
```

always_on Simple Attribute

The `always_on` simple attribute models always-on cells and signal pins. During compilation, the `always_on` attribute is automatically added to buffer or inverter cells that have input and output pins that are linked to backup power or backup ground PG pins. The `always_on` attribute is supported at the cell level and at the pin level.

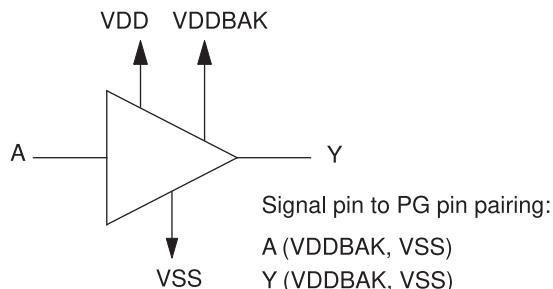
Note:

Some macro cell input pins require that you specify the `always_on` attribute for always-on pins.

Always-On Simple Buffer Example

Figure 93 and the example that follows it show a simple always-on cell buffer. In the figure, the A signal pin and the Y signal pin are linked to the VDDBAK and VSS power and ground pin pair.

Figure 93 Simple Always-On Cell Buffer



```
library (my_library) {
...
voltage_map (VDD, 1.0);
voltage_map (VDDBAK, 1.0);
voltage_map (VSS, 0.0);
...
cell(buffer_type_A0) {
    always_on : true;
/* The always-on attribute is not required at the cell
```

```
level if the cell is an always-on cell*/
/* Other cell level information */

pg_pin(VDD) {
    voltage_name : VDD;
    pg_type : primary_power;
}

pg_pin(VDDBAK) {
    voltage_name : VDDBAK;
    pg_type : backup_power;
}

pg_pin(VSS) {
    voltage_name : VSS;
    pg_type : primary_ground;
}
...
pin (A) {
/* The always-on attribute is not required at the pin
level if the cell is an always-on cell*/
    related_power_pin : VDDBAK;
    related_ground_pin : VSS;
/* Other pin level information */
}
pin (Y) {
/* The always-on attribute is not required at the pin
level if the cell is an always-on cell*/
    function : "A";
    related_power_pin : VDDBAK;
    related_ground_pin : VSS;
    power_down_function : "!VDDBAK + VSS";
/* Other pin level information */
} /* End Pin group */

} /* End Cell group */
...
} /* End Library group */
```

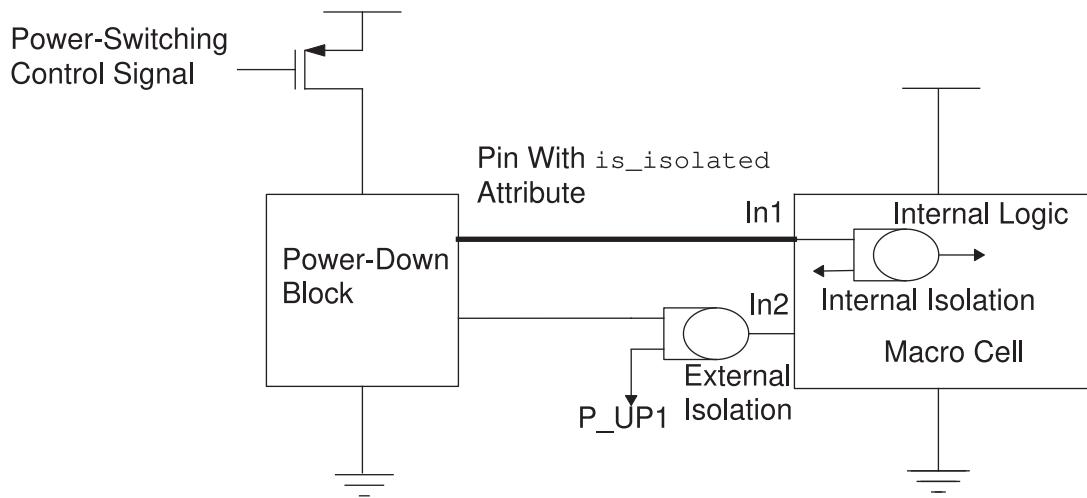
Macro Cell Modeling

Macro cells do not contain internal logic information. At the cell-level, macro cells are modeled using the `is_macro_cell` attribute. To model macro cells for power management, use the power management pin-level attributes described in the following sections.

Macro Cell Isolation Modeling

To indicate that a macro cell is internally isolated and does not require external isolation, set the `is_isolated` attribute. [Figure 94](#) shows a macro cell with the `is_isolated` attribute. The macro cell is connected to a power-switching circuit. The macro cell pin, `In1` is internally isolated, and `In2` is connected to an external isolation cell. When the `is_isolated` attribute is set on the pin, `In1`, the pin is considered to be internally isolated.

Figure 94 Macro Cell With the `is_isolated` Attribute



[Example 109](#) shows the modeling syntax of an internally isolated macro cell.

Example 109 Macro Cell Isolation Modeling Syntax

```
cell (cell_name) {  
    ...  
    is_macro_cell : true;  
    pin (pin_name) {  
        direction : input | inout | output;  
        is_isolated : true | false;  
        isolation_enable_condition : Boolean_expression;  
        ...  
    }  
    bus (bus_name) {  
        direction : input | inout | output;  
        is_isolated : true | false;
```

```
    isolation_enable_condition : Boolean_expression;
    ...
}
bundle (bundle_name) {
    direction : input | inout | output;
    is_isolated : true | false;
    isolation_enable_condition : Boolean_expression;
    ...
}
...
}
```

[Example 110](#) shows a typical model of an internally isolated macro cell.

Example 110 Modeling an Internally Isolated Macro Cell by Using the is_isolated and isolation_enable_condition Attributes

```
library (internal_isolated_pin_example) {
    ...
type ( bus_type_name ) {
    base_type : array;
    data_type : bit;
    bit_width : 2;
    bit_from : 1;
    bit_to : 0;
}
cell ( macro ) {
    is_macro_cell : true;
    pg_pin(GND) {
        voltage_name : VSS;
        pg_type : primary_ground;
    }
    pg_pin(VDDI) {
        voltage_name : VDDH;
        pg_type : primary_power;
    }
    .....
    pin (en) {
        direction : input;
        ...
    }
    pin (in_bus) {
        bus_type : bus_type_name;
        direction : input;
        ...
    }
    pin (sp) {
        direction : input|inout;
        is_isolated : true;
        isolation_enable_condition : "en'";
        related_power_pin : VDDI;
        related_ground_pin : GND;
        .....
    }
}
```

```
        }
        pin (out) {
            direction : output;
            is_isolated : true;
            isolation_enable_condition : "en'";
            related_power_pin : VDDI;
            related_ground_pin : GND;
            .....
        }
        bus (bus_a) {
            bus_type : bus_type_name;
            is_isolated : true;
            isolation_enable_condition : "en' * in_bus * in_bundle";
            related_power_pin : VDDI;
            related_ground_pin : GND;
            ...
        }
        ...
    }
}
```

Pin-Level Attributes

This section describes the pin-level attributes of isolated macro cells.

is_isolated Attribute

The `is_isolated` attribute indicates that a pin, bus, or bundle of a macro cell is internally isolated and does not require the insertion of an external isolation cell. The default is `false`.

Note:

The `is_isolated` attribute also supports the internal isolation of pad-cell pins.

isolation_enable_condition Attribute

The `isolation_enable_condition` attribute specifies the condition of isolation for internally isolated pins, buses, or bundles of a macro cell. When this attribute is defined in a `pin` group, the corresponding Boolean expression can include only input and inout pins. Do not include the output pins of an internally isolated macro cell in the Boolean expression.

When the `isolation_enable_condition` attribute is defined in a `bus` or `bundle` group, the corresponding Boolean expression can include pins, and buses and bundles of the same bit-width. For example, when the Boolean expression includes a bus and a bundle, both of them must have the same bit-width.

All the pins, buses, and bundles specified in the Boolean expression of the `isolation_enable_condition` attribute must have the `always_on` attribute.

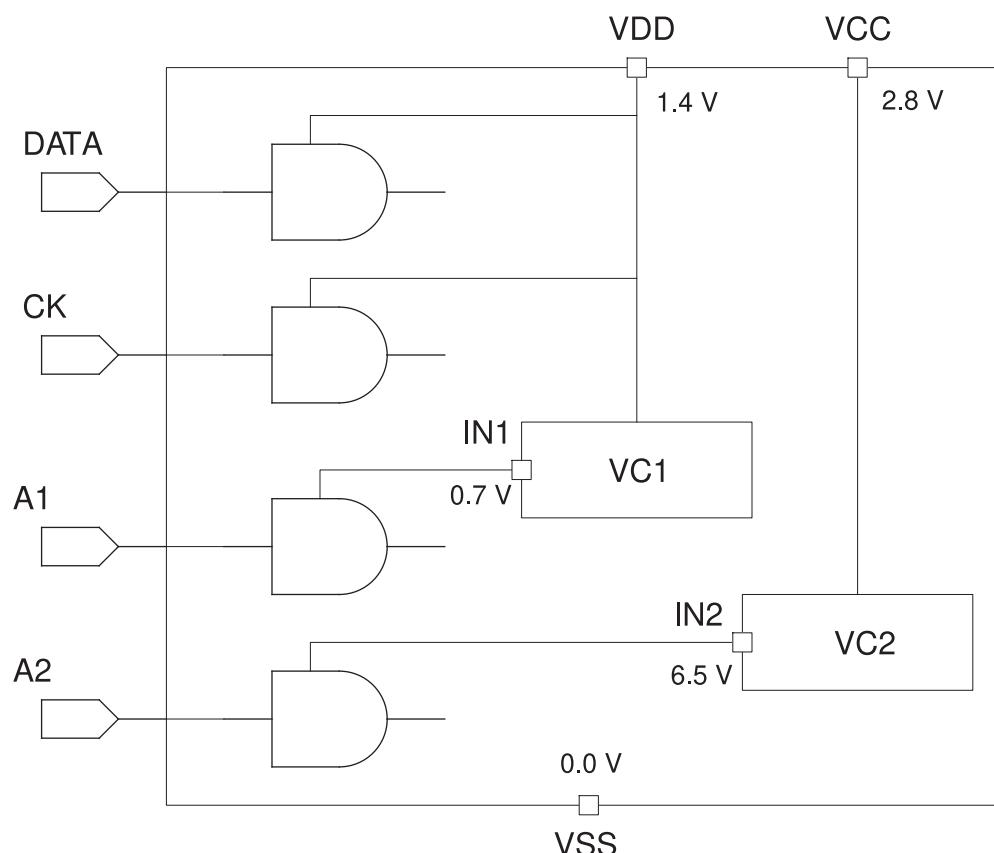
Note:

The `isolation_enable_condition` attribute also supports the internal isolation of pad-cell pins.

Modeling Macro Cells With Internal PG Pins

This capability is useful for modeling macro cells containing internal voltage converters, VC1 and VC2. VC1 and VC2 convert the external supply voltages, VDD and VCC, into internal supply voltages, IN1 and IN2, respectively. The pins, IN1 and IN2, are internal PG pins of the macro cell and are used to power the logic connected to the input ports, A1 and A2.

Figure 95 *Macro Cell With Internal PG Pins*



Example 111 shows a typical model of the macro cell. The pg_type attribute is set to internal_power, the direction attribute is set to internal, and the pg_function attribute is set to VDD and VCC, on the PG pins, IN1 and IN2, respectively.

Example 111 Macro Cell Model With Internal PG Pins

```
voltage_map (VDD, 1.4);
voltage_map (VCC, 2.8);
voltage_map (VDD, 1.4);
voltage_map (VCC, 2.8);
voltage_map (VSS, 0.0);
voltage_map (IN1, 0.7);
voltage_map (IN2, 6.5);
cell(new_macro_cell) {
    ...
    is_macro_cell : true;
    pg_pin(VDD) {
        voltage_name : VDD;
        pg_type : primary_power;
    }
    pg_pin(VCC) {
        voltage_name : VCC;
        pg_type : primary_power;
    }
    pg_pin(VSS) {
        voltage_name : VSS;
        pg_type : primary_ground;
    }
    pg_pin(IN1) {
        voltage_name : IN1;
        pg_type : internal_power;
        direction : internal ;
        pg_function : VDD;
    }
    pg_pin(IN2) {
        voltage_name : IN2;
        pg_type : internal_power;
        direction : internal ;
        pg_function : VCC;
    }
    ...
    pin(CK) {
        related_power_pin : VDD ;
        related_ground_pin : VSS ;
        direction : input ;
        ...
    }
    pin(A1) {
        related_power_pin : IN1 ;
        related_ground_pin : VSS ;
        direction : input ;
        ...
    }
}
```

```
        }
    pin(A2) {
        related_power_pin : IN2 ;
        related_ground_pin : VSS ;
        direction : input ;
    }
    ...
}
```

Note:

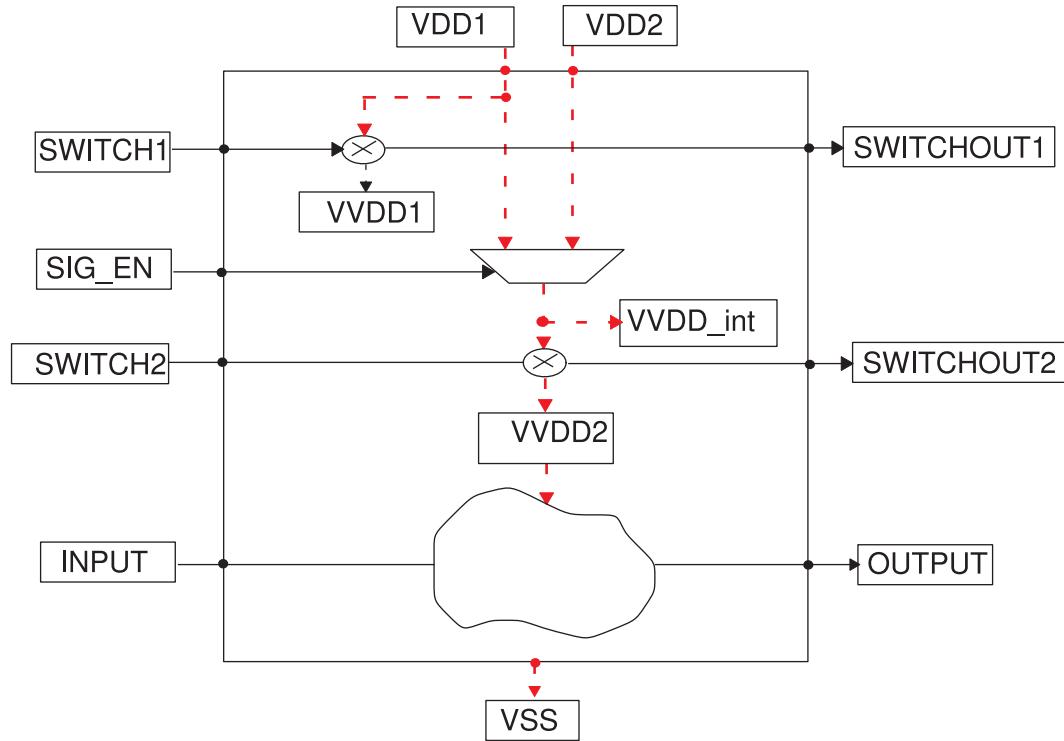
You do not need to specify the `switch_cell_type` and `switch_function` attributes for this macro cell because it does not include any built-in switches.

Macro Cell With Built-in Multiplexer

The figure in [Macro Cell With Fine-Grained Internal Power Switches](#) and the example that follows show a macro cell with fine-grained internal power switches and a built-in power multiplexer with input PG pin rails, VDD1 and VDD2.

Some of the signal pins are related to the primary PG pins. For the signal pins, INPUT and OUTPUT, the related power pin is VDD1. However for the SWITCH2 signal pin, the related power pin is VVDD_int.

Figure 96 Macro Cell With Built-in Multiplexer



In the following Liberty model, the internal PG pin, VVDD_int, does not have the `switch_function` attribute. The Boolean expression of the VVDD_int `pg_function` attribute includes the signal pin, SIG_EN.

```
library(mylib) {
    .../* unit attributes */
    voltage_map(VDD1, 0.80); /* primary power */
    voltage_map(VDD2, 1.20); /* secondary power */
    voltage_map(VVDD1, 0.80); /* internal power */
    voltage_map(VVDD_int, 0.80); /* internal power */
    voltage_map(VVDD2, 0.80); /* internal power */
    voltage_map(VSS, 0.0); /* primary ground */
    ... /* nominal PVT definitions */
    ... /* operation conditions */
    ... /* threshold definitions */
    .../* default attributes */
    ... /* table templates */
    cell(PG_MUX_macro) {
        area : 1.0;
        switch_cell_type : fine_grain;
        is_macro_cell : true;
        pin(SIG_EN) {
```

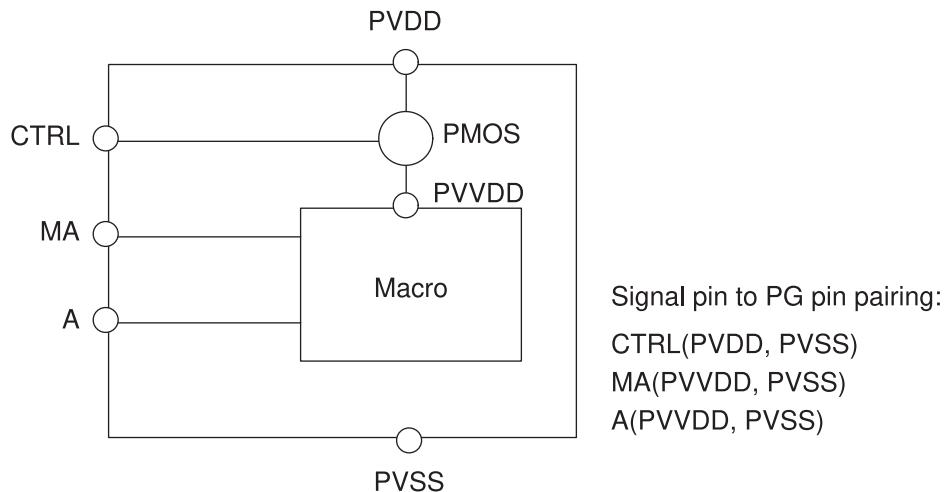
```
    direction : input;
    capacitance : 1.00;
    related_power_pin : VDD1;
    related_ground_pin : VSS;
    switch_pin : true;
    internal_power() {
        ...
    }
}
pin(SWITCH1) {
    direction : input;
    capacitance : 1.00;
    related_power_pin : VDD1;
    related_ground_pin : VSS;
    switch_pin : true;
    internal_power() {
        ...
    }
}
pin(SWITCH2) {
    direction : input;
    capacitance : 1.00;
    related_power_pin : VVDD_int;
    related_ground_pin : VSS;
    switch_pin : true;
    internal_power() {
        ...
    }
}
pg_pin(VDD1) {
    pg_type : primary_power;
    direction : input ;
    voltage_name : VDD1;
}
pg_pin(VDD2) {
    pg_type : primary_power;
    direction : input ;
    voltage_name : VDD2;
}
pg_pin (VVDD1) {
    pg_type : internal_power;
    voltage_name : VVDD1;
    direction : internal ;
    switch_function : "(SWITCH1)";
    pg_function : "VDD1";
}
pg_pin (VVDD_int) {
    pg_type : internal_power;
    voltage_name : VVDD_int;
    direction : internal ;
    pg_function : "(VDD1 * !SIG_EN + VDD2 * SIG_EN)";
}
pg_pin (VVDD2) {
```

```
    pg_type : internal_power;
    voltage_name : VVDD2;
    direction : internal ;
    switch_function : "(SWITCH2)";
    pg_function : "(VVDD_int)";
}
pg_pin ( VSS ) {
    pg_type : primary_ground;
    voltage_name : VSS;
    direction : inout ;
}
pin(SWITCHOUT1) {
    direction : output;
    related_power_pin : VDD1;
    related_ground_pin : VSS;
    function : "SWITCH1";
    power_down_function : "!VDD1 + VSS";
    timing() {
        related_pin : "SWITCH1";
        ...
    }
}
pin(SWITCHOUT2) {
    direction : output;
    related_power_pin : VVDD_int;
    related_ground_pin : VSS;
    function : "SWITCH2";
    power_down_function : "!VVDD_int + VSS";
    timing() {
        related_pin : "SWITCH1";
        ...
    }
}
pin(INPUT) {
    direction : input;
    related_power_pin : VVDD2;
    related_ground_pin : VSS;
}
pin(OUTPUT) {
    direction : output;
    related_power_pin : VVDD2;
    related_ground_pin : VSS;
    power_down_function : "!VVDD2 + VSS";
    timing() {
        related_pin : "INPUT";
        ...
    }
}
} /*end library group*/
```

Macro Cell With Fine-Grained Internal Power Switches

Figure 97 and the example that follows it show a macro cell with fine-grained internal power switches. In the figure, the CTRL signal pin is linked to the PVDD and PVSS power and ground pin pair, and the MA signal pin and the A signal pin are linked to the PVVDD and PVSS power and ground pin pair.

Figure 97 Macro Cell With Fine-Grained Power Switch Schematics



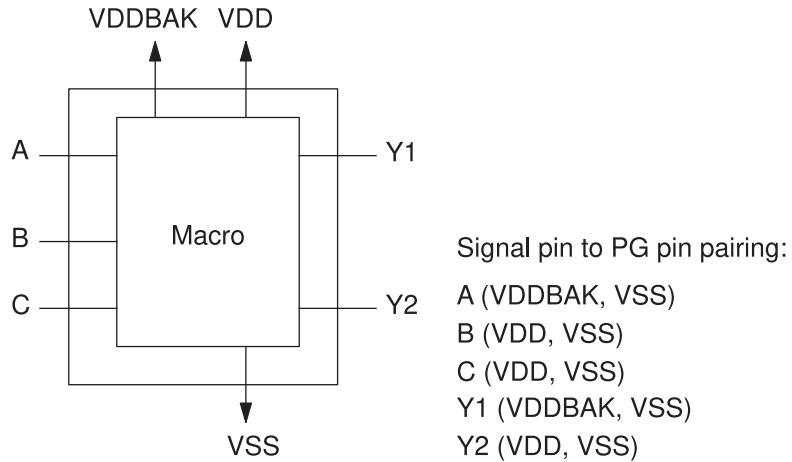
```
library (macro_switch) {  
...  
Voltage_map (PVDD, 1.0);  
Voltage_map (PVVDD, 1.0);  
Voltage_map (PVSS, 0.0);  
  
operating_conditions(XYZ) {  
    process : 1.0;  
    voltage : 1.0;  
    temperature : 25.0;  
}  
default_operating_conditions : XYZ;  
  
cell(MACRO) {  
    is_macro_cell : true;  
    switch_cell_type : fine_grain;  
...  
  
pg_pin(PVDD) {  
    voltage_name : PVDD;  
    pg_type : primary_power;  
    direction : input;
```

```
        }
    pg_pin(PVSS) {
        voltage_name : PVSS;
        pg_type : primary_ground;
        direction : input;
    }
    pg_pin(PVVDD) {
        voltage_name : PVVDD;
        pg_type : internal_power;
        direction : internal;
        switch_function : "CTRL";
        pg_function : "PVDD";
    }
    pin(CTRL) {
        direction : input;
        switch_pin : true;
        related_power_pin : PVDD;
        related_ground_pin : PVSS;
        ...
    }
    pin(A) {
        direction : input;
        related_power_pin : PVVDD;
        related_ground_pin : PVSS;
        ...
    }
    pin(MA) {
        direction : input;
        power_down_function : "!PVDD + PVSS";
        related_power_pin : PVVDD;
        related_ground_pin : PVSS;
        ...
    }
}
```

Macro Cell With an Always-On Pin Example

Figure 98 and the example that follows it show a macro cell with one always-on pin. In the figure, the A signal pin and Y1 signal pin are linked to the VDDBAK and VSS power and ground pin pair. The B, C, and Y2 signal pins are linked to the VDD and VSS power and ground pin pair.

Figure 98 Macro Cell With an Always-On Pin



```
library (my_library) {
...
    voltage_map (VDD, 2.0) ;
    voltage_map (VSS, 0.1) ;
    voltage_map (VDDBAK, 1.0) ;
...
cell(Macro_cell_with_AO_pins) {
/* other cell level information */

    pg_pin(VDD) {
        voltage_name : VDD;
        pg_type : primary_power;
    }
    pg_pin(VSS) {
        voltage_name : VSS;
        pg_type : primary_ground;
    }
    pg_pin(VDDBAK) {
        voltage_name : VDDBAK;
        pg_type : backup_power;
    }
...
    pin (A) {
        always_on : true;
        related_power_pin : VDDBAK;
        related_ground_pin : VSS;
    /* Other pin level information */
    }
    pin (B C) {
        related_power_pin : VDD;
        related_ground_pin : VSS;
```

```
/* Other pin level information */
}
pin (Y1) {
    always_on : true;
    function : "A";
    related_power_pin : VDDBAK;
    related_ground_pin : VSS;
    power_down_function : "!VDD + !VDDBAK + VSS";
/* Other pin specific information */
} /* End Pin group */
pin (Y2) {
    function : "A";
    related_power_pin : VDD;
    related_ground_pin : VSS;
    power_down_function : "!VDD + !VDDBAK + VSS";
/* Other pin specific information */
} /* End Pin group */
...
} /* End Cell group */
...
} /* End Library group */
```

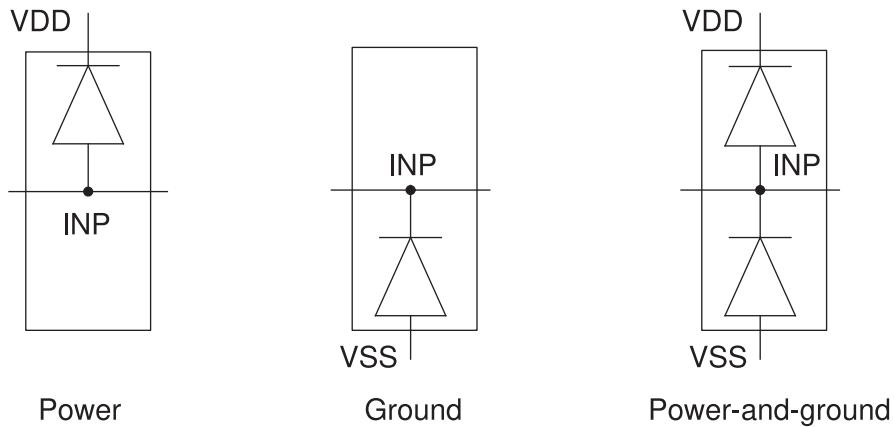
Modeling Antenna Diodes

Modeling antenna diodes includes modeling antenna-diode cells and cells with built-in antenna-diode ports.

Antenna-Diode Cell Modeling

An antenna-diode cell has only one input to a diode that discharges electrical charges. The cell is typically inserted at the boundary between two power domains and can be placed in any one of the power domains. [Figure 99](#) shows the three types of antenna-diode cells.

Figure 99 Types of Antenna-Diode Cells



In multivoltage designs, the power type antenna-diode cell is connected to VDD of a power domain where VSS is shut down. The ground type antenna-diode cell is connected to VSS of a power domain where VDD is shut down. The power-and-ground type antenna-diode cell is connected to both VDD and VSS. To eliminate leakage paths that can result in chip failure, the correct type of antenna-diode cell must be inserted.

The modeling syntax of the antenna-diode cell is as follows:

```
cell (cell name) {
    antenna_diode_type : power | ground | power_and_ground;
    pin (pin name) {
        antenna_diode_related_power_pins : power pin name;
        antenna_diode_related_ground_pins : ground pin name;
        ...
    }
    ...
}
```

In a library, a cell that has a single pin with the `direction` attribute set to `input` or `inout` is considered to be an antenna-diode cell.

Cell-Level Attribute

antenna_diode_type Attribute

The `antenna_diode_type` attribute specifies the type of antenna-diode cell. Valid values are `power`, `ground`, and `power_and_ground`.

Pin-Level Attributes

antenna_diode_related_power_pins Attribute

The `antenna_diode_related_power_pins` attribute specifies the related power pin of the antenna-diode cell. Apply the `antenna_diode_related_power_pins` attribute to the input pin of the cell.

antenna_diode_related_ground_pins Attribute

The `antenna_diode_related_ground_pins` attribute specifies the related ground pin of the antenna-diode cell. Apply the `antenna_diode_related_ground_pins` attribute to the input pin of the cell.

Antenna-Diode Cell Modeling Example

[Example 112](#) shows a typical model of the antenna-diode cell.

Example 112 Antenna-Diode Cell Model

```
library (antenna_library)
...
cell (power_diode_cell) {
    antenna_diode_type : "power";
    pg_pin (VDD) {
        voltage_name : "VDD";
        pg_type : "primary_power";
    }
    pin (INP) {
        antenna_diode_related_power_pins : "VDD";
        direction : "input";
    }
}
cell (ground_diode_cell) {
    antenna_diode_type : "ground";
    pg_pin (VSS) {
        voltage_name : "VSS";
        pg_type : "primary_ground";
    }
    pin (INP) {
        antenna_diode_related_ground_pins : "VSS";
        direction : "input";
    }
}
cell (pg_diode_cell) {
    antenna_diode_type : "power_and_ground";
    pg_pin (VDD) {
        voltage_name : "VDD";
        pg_type : "primary_power";
    }
    pg_pin (VSS) {
```

```
        voltage_name : "VSS";
        pg_type : "primary_ground";
    }
    pin (INP) {
        antenna_diode_related_power_pins : "VDD";
        antenna_diode_related_ground_pins : "VSS";
        direction : "input";
    }
}
```

Modeling Cells With Built-In Antenna-Diode Ports

To model a cell with a built-in antenna-diode pin, specify the `antenna_diode_type` attribute on the pin.

The modeling syntax of the cell with a built-in antenna-diode pin is as follows:

```
cell (cell name) {
    ...
    pin (pin name) {
        antenna_diode_type : power | ground | power_and_ground;
        antenna_diode_related_power_pins : "power_pin1 power_pin2";
        antenna_diode_related_ground_pins : "ground_pin1 ground_pin2";
        ...
    }
    pin (pin name) {
        ...
    }
    ...
}
```

Pin-Level Attributes

antenna_diode_type Attribute

The `antenna_diode_type` attribute specifies the type of antenna-diode pin. Valid values are `power`, `ground`, and `power_and_ground`.

antenna_diode_related_power_pins Attribute

The `antenna_diode_related_power_pins` attribute specifies the related power pins for the antenna-diode pin. Apply the `antenna_diode_related_power_pins` attribute to the antenna-diode pin.

antenna_diode_related_ground_pins Attribute

The `antenna_diode_related_ground_pins` attribute specifies the related ground pins for the antenna-diode pin. Apply the `antenna_diode_related_ground_pins` attribute to the antenna-diode pin.

Antenna-Diode Pin Modeling Example

[Example 113](#) shows a typical model of a cell with a built-in antenna-diode pin.

Example 113 A Cell Model With Built-In power_and_ground Antenna-Diode Pin Port

```
cell (cell_with_internal_diode_port)
    area : "1.0";
    pg_pin (VDD) {
        voltage_name : "VDD";
        pg_type : "primary_power";
    }
    pg_pin (VDD1) {
        voltage_name : "VDD1";
        pg_type : "primary_power";
    }
    pg_pin (VSS) {
        voltage_name : "VSS";
        pg_type : "primary_ground";
    }
    pg_pin (VSS1) {
        voltage_name : "VSS1";
        pg_type : "primary_ground";
    }
    pin (antenna_diode) {
        antenna_diode_type: power_and_ground;
        antenna_diode_related_power_pins : "VDD VDD1";
        antenna_diode_related_power_pins : "VSS VSS1";
        direction : "input";
        capacitance : "1.0";
    }
    pin (INP1) {
        direction : "input";
        capacitance : "1.0";
    }
    pin (INP2) {
        direction : "input";
        capacitance : "1.0";
    }
    pin (OUT1) {
        related_power_pin : "VDD";
        related_ground_pin : "VSS";
        direction : "output";
    }
    pin (OUT1) {
        related_power_pin : "VDD";
        related_ground_pin : "VSS";
        direction : "output";
    }
}
```

10

Composite Current Source Modeling

This chapter provides an overview of composite current source modeling (CCS).

It covers the syntax for CCS modeling in the following sections:

- [Modeling Cells With Composite Current Source Information](#)
- [Representing Composite Current Source Driver Information](#)
- [Representing Composite Current Source Receiver Information](#)
- [Comparison Between Two-Segment and Multisegment Receiver Models](#)
- [Two-Segment Receiver Capacitance Model](#)
- [Multisegment Receiver Capacitance Model](#)
- [CCS Retain Arc Support](#)
- [Composite Current Source Driver and Receiver Model Example](#)

Modeling Cells With Composite Current Source Information

Composite current source (CCS) models include driver and receiver models. A driver model can be used with or without the receiver model.

Nanometer transistor geometries with nonlinear waveforms, nonlinear interconnect delays, and high circuit speeds require accurate delay calculation. CCS models support driver complexity by using a time- and voltage- dependent current source with an infinite drive resistance. The driver model maps the arbitrary transistor behavior for lumped loads to that for an arbitrary detailed parasitic network, which results in high accuracy.

In the receiver model, the input capacitance of a receiver is dynamically adjusted during the voltage transitions.

Representing Composite Current Source Driver Information

Specify the nonlinear delay information based on input slew and output load at the pin level by defining a current lookup table in a timing group.

Composite Current Source Lookup Tables

To define the lookup tables, use the following groups and attributes:

- `output_current_template` group in the `library` group
 - `output_current_rise` and `output_current_fall` groups in the `timing` group
-

Defining the `output_current_template` Group

Use this library-level group to create templates of common information that multiple current vectors can use. A table template specifies the composite current source driver model and the breakpoints for the axis. Specifying `index_1`, `index_2`, and `index_3` values at the library level is optional.

`output_current_template` Syntax

```
library(name_id) {  
    ...  
    output_current_template(template_name_id) {  
        variable_1: input_net_transition;  
        variable_2: total_output_net_capacitance;  
        variable_3: time;  
        ...  
    }  
    ...  
}
```

Template Variables for CCS Driver Models

The table template specifying composite current source driver models can have three variables: `variable_1`, `variable_2`, and `variable_3`. The valid values for `variable_1` and `variable_2` are `input_net_transition` and `total_output_net_capacitance`. The only valid value for `variable_3` is `time`.

`output_current_template` Example

```
library (new_lib) {  
    ...  
    output_current_template (CCT) {  
        variable_1: input_net_transition;  
        variable_2: total_output_net_capacitance;  
        variable_3: time;  
        ...  
    }  
    ...  
}
```

Defining the Lookup Table Output Current Groups

To specify the output current for the nonlinear table model, use the `output_current_rise` and `output_current_fall` groups within the `timing` group.

Note:

The `output_current_rise` group can include negative current values and the `output_current_fall` group can include positive current values.

`output_current_rise` Syntax

```
timing() {
    output_current_rise () {
        vector (template_name_id) {
            reference_time : float;
            index_1 (float);
            index_2 (float);
            index_3 ("float,..., float");
            values("float,..., float");
        }
    }
}
```

vector Group

Define the `vector` group in the `output_current_rise` or `output_current_fall` group. This group stores current information for a particular input slew and output load.

`reference_time` Simple Attribute

Define the `reference_time` simple attribute in the `vector` group. The `reference_time` attribute represents the time at which the input waveform crosses the rising or falling input delay threshold.

Template Variables for CCS Driver Models

The table template specifying composite current source driver models can have three variables: `variable_1`, `variable_2`, and `variable_3`. The valid values for `variable_1` and `variable_2` are `input_net_transition` and `total_output_net_capacitance`. The only valid value for `variable_3` is `time`.

The index value for `input_net_transition` or `total_output_net_capacitance` is a single floating-point number. The index values for `time` are a list of floating-point numbers.

The `values` attribute defines a list of floating-point numbers that represent the current values of the driver model.

vector Group Example

```
library (new_lib) {
    ...
    output_current_template (CCT) {
        variable_1: input_net_transition;
        variable_2: total_output_net_capacitance;
        variable_3: time;
    }
    ...
    timing() {
        output_current_rise() {
            vector(CCT) {
                reference_time : 0.05;
                index_1(0.1);
                index_2(2.1);
                index_3("1.0, 1.5, 2.0, 2.5, 3.0");
                values("1.1, 1.2, 1.4, 1.3, 1.5");
            }
            ...
        }
        output_current_fall() {
            vector(CCT) {
                reference_time : 0.05;
                index_1(0.1);
                index_2(2.1);
                index_3("1.0, 1.5, 2.0, 2.5, 3.0");
                values("1.1, 1.2, 1.4, 1.3, 1.5");
            }
            ...
        }
    }
}
```

Representing Composite Current Source Receiver Information

Switching transistors have nonlinear capacitance. At an input or inout node, the collective capacitance of all the transistors connected to that node is modeled as the receiver capacitance. The Miller effect due to the output switching in the opposite direction adds to the nonlinearity. The capacitances are dependent on the input slew and output load.

CCS receiver models are of two types:

- Two-segment receiver capacitance model

The voltage rise or fall at an input or inout node is divided into two segments and the corresponding capacitance values are stored as receiver_capacitance1 and receiver_capacitance2 lookup tables.

- Multisegment capacitance model

The voltage rise or fall at an input or inout node is divided into multiple segments and the corresponding capacitance values are stored as lookup tables. The number of segments is represented by the letter N. This model better represents the nonlinearity of the net receiver capacitance.

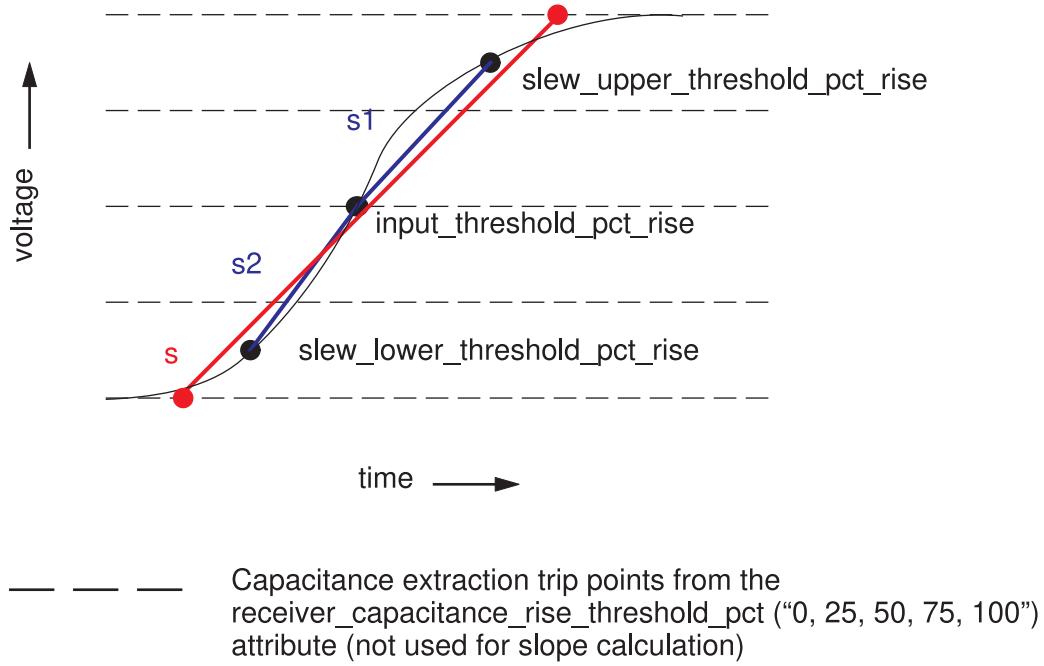
Comparison Between Two-Segment and Multisegment Receiver Models

You might achieve different results with the two-segment and the multisegment receiver capacitance models.

This is because the number of segments are different. Also, the multisegment model uses the input slew value of the library-level `slew_derate_from_library` attribute, shown with the solid red line in [Figure 100](#). This slew is used for the `input_net_transition(index_1)` values of all the N segments, regardless of the local waveform slopes.

The two-segment model uses the local slope of the two segments (s_1 and s_2 in [Figure 100](#)), shown with the solid blue lines. So even at $N = 2$, the multisegment and two-segment model values might not match.

Figure 100 Slope Calculation in Multisegment and Two-Segment Receiver Models



Two-Segment Receiver Capacitance Model

Define the CCS receiver model

- At the pin level using the `receiver_capacitance group`.
Specify the `receiver_capacitance1_rise`, `receiver_capacitance1_fall`, `receiver_capacitance2_rise`, `receiver_capacitance2_fall` groups in the `receiver_capacitance group`.
The pin-level definition is not a function of the output capacitance and is useful when there are no forward timing arcs.
- At the timing level by specifying the `receiver_capacitance1_rise`, `receiver_capacitance1_fall`, `receiver_capacitance2_rise`, `receiver_capacitance2_fall` groups in the timing group.

Syntax

The following is the Liberty syntax of CCS timing receiver models.

```
cell(cell_name) {
    mode_definition(mode_name) {
        mode_value(name_string) {
            when : "Boolean_expression";
            sdf_cond : "Boolean_expression";
        } ...
    } ...
    pin(pin_name) {
        direction : input; /* or "inout" */
        receiver_capacitance() {
            when : "Boolean_expression";
            char_when : "Boolean_expression";

            mode (mode_name, mode_value);
            receiver_capacitance1_rise (lu_template_name) {
                index_1("float, ... , float");
                index_2("float, ... , float");
                values("float, ... , float");
            }
            receiver_capacitance1_fall (lu_template_name) { ... }
            receiver_capacitance2_rise (lu_template_name) { ... }
            receiver_capacitance2_fall (lu_template_name) { ... }
        }
    }
    pin(pin_name) {
        direction : output; /* or "inout" */
        timing() {
            when : "Boolean_expression";
            mode (mode_name, mode_value);
            ...

            receiver_capacitance() {
                receiver_capacitance1_rise (lu_template_name) {
                    index_1("float, ... , float");
                    index_2("float, ... , float");
                    values("float, ... , float");
                }
                receiver_capacitance1_fall (lu_template_name) { ... }
                receiver_capacitance2_rise (lu_template_name) { ... }
                receiver_capacitance2_fall (lu_template_name) { ... }
            }
        }
    } ...
}
```

Defining the receiver_capacitance Group at the Pin Level

To model the CCS receiver capacitance at the pin level, define the receiver_capacitance group in the pin group. In the receiver_capacitance group, specify the receiver_capacitance1_rise, receiver_capacitance1_fall, receiver_capacitance2_rise, and receiver_capacitance2_fall groups. Each

of these groups specify a lookup table. The lookup table can be one-dimensional, two-dimensional, or scalar.

The index values in the `index_1` and `index_2` attributes are a list of ascending floating-point numbers. The `values` attribute defines a list of floating-point numbers that represent the capacitances of the receiver model.

Define the template of the lookup table in the library-level `lu_table_template` group. To use the template, specify the name of the `lu_table_template` group as the receiver capacitance group name.

Defining the `lu_table_template` Group

The `lu_table_template` group defines the template for the following groups specified in the `receiver_capacitance` group:

- `receiver_capacitance1_rise`
- `receiver_capacitance1_fall`
- `receiver_capacitance2_rise`
- `receiver_capacitance2_fall`

The template definition includes the `variable_1`, `variable_2`, `index_1`, and `index_2` attributes. The `variable_1` and `variable_2` attributes specify the index variables of the lookup tables of the receiver capacitance groups. The `index_1` and `index_2` attributes specify the numerical values of these variables.

The values of the `variable_1` and `variable_2` attributes are `input_net_transition` and `total_output_net_capacitance`.

You can specify the following types of lookup tables in receiver capacitance groups:

- One-dimensional tables with a single index, such as `input_net_transition`
- Two-dimensional tables with the `indexes`, `input_net_transition` and `total_output_net_capacitance`
- A scalar that has a single variation value irrespective of the `input_net_transition` and `total_output_net_capacitance` values

`lu_table_template` Group Syntax

```
library(name) {  
...  
    lu_table_template(template_name) {  
        variable_1: input_net_transition;  
        variable_2: total_output_net_capacitance;  
        index_1 ("float,..., float");  
        index_2 ("float,..., float");  
    }  
}
```

```
...
}
...
}
```

Conditional Data Modeling

Use the following attributes in the `receiver_capacitance` group for conditional data modeling in CCS receiver models.

when Attribute

The `when` attribute specifies the Boolean condition for the input state of the receiver capacitance timing arc.

char_when Attribute

The `char_when` attribute specifies the input state at which the receiver capacitance timing arc was characterized.

char_when_rise Attribute

The `char_when_rise` attribute specifies the rising input state at which the receiver capacitance timing arc was characterized.

char_when_fall Attribute

The `char_when_fall` attribute specifies the falling input state at which the receiver capacitance timing arc was characterized.

mode Attribute

The complex `mode` attribute specifies the current mode of the cell. When defined in the `receiver_capacitance` group, the `mode` attribute specifies the receiver capacitance for the given mode. If you specify the `mode` attribute, you must define the `mode_definition` group at the cell level.

Examples

[Example 114](#) shows a receiver capacitance model with conditional data, that is, the `when` and `mode` attributes.

Example 114 Modeling Receiver Capacitance With when and mode Attributes

```
library(new_lib) {
...
output_current_template(CCT) {
    variable_1: Input_net_transition;
    variable_2: total_output_net_capacitance;
    variable_3: time;
    index_1("0.1, 0.2");
    index_2("1, 2");
    index_3("1, 2, 3, 4, 5");
}
```

Chapter 10: Composite Current Source Modeling
Two-Segment Receiver Capacitance Model

```
lu_table_template(LTT1) {
    variable_1: input_net_transition;
    index_1("0.1, 0.2, 0.3, 0.4");
}
lu_table_template(LTT2) {
    variable_1: input_net_transition;
    variable_2: total_output_net_capacitance;
    index_1("0.1, 0.2");
    index_2("1, 2");
}
...
cell(my_cell) {
    ...
    mode_definition(rw) {
        mode_value(read) {
            when : "I";
            sdf_cond : "I == 1";
        }
        mode_value(write) {
            when : "!I";
            sdf_cond : "I == 0";
        }
    }
    pin(I) /* pin-based receiver model defined for pin 'A' */
        direction : input;
        /* receiver capacitance for condition 1 */
        receiver_capacitance() {
            when : "I"; /* or using mode as next commented line */
            /* mode (rw, read); */
            receiver_capacitance1_rise(LTT1) {
                values("1, 2, 3, 4");
            }
            receiver_capacitance1_fall(LTT1) {
                values("1, 2, 3, 4");
            }
            receiver_capacitance2_rise(LTT1) {
                values("1, 2, 3, 4");
            }
            receiver_capacitance2_fall(LTT1) {
                values("1, 2, 3, 4");
            }
        }
        /* receiver capacitance for condition 2 */
        receiver_capacitance() {
            when : "!I"; /* or using mode as next commented line */
            /* mode (rw, write); */
            receiver_capacitance1_rise(LTT1) {
                values("1, 2, 3, 4");
            }
            receiver_capacitance1_fall(LTT1) {
                values("1, 2, 3, 4");
            }
            receiver_capacitance2_rise(LTT1) {
                values("1, 2, 3, 4");
            }
            receiver_capacitance2_fall(LTT1) {
                values("1, 2, 3, 4");
            }
        }
    pin (ZN) {
```

Chapter 10: Composite Current Source Modeling Two-Segment Receiver Capacitance Model

```
direction : input;
capacitance : 1.2;
...
timing() {
}
}
} /* end cell */
} /* end library */
```

Example 115 shows the use of the `char_when` attribute in the pin-level receiver capacitance model.

Example 115 Receiver Capacitance Model With the char_when Attribute

```
library (mylib) {
    /* library unit, slew, OC and other library level information */
    /* receiver model template */
    lu_table_template ("template_8x1") {
        variable_1 : "input_net_transition";
        index_1("0, 1, 2, 3, 4, 5, 6, 7");
    }
    ...
    cell (mycell) {
        pin (Y) {
            direction : "output";
            function : "A1 * A2 * A3";
            ...
            /* timing-based ccs receiver model and driver model */
            timing () {
                related_pin : "A1";
                ...
            }
            timing () {
                related_pin : "A2";
                ...
            }
            timing () {
                related_pin : "A3";
                ...
            }
            ...
        } /* end pin Y */
        pin (A1) {
            direction : "input";
            ...
            /* pin-based ccs receiver model */
            receiver_capacitance () {
                /* default condition */
                /* characterization input-state */
            }
        }
    }
}
```

Chapter 10: Composite Current Source Modeling
Two-Segment Receiver Capacitance Model

```
char_when : "!A2 * !A3";
receiver_capacitance1_rise ("template_8x1") {
    values("0.11111, 0.22222, 0.33333, 0.44444, 0.55555, \
           0.66666, 0.77777, 0.88888");
}
receiver_capacitance2_rise ("template_8x1") {
    values("0.11111, 0.22222, 0.33333, 0.44444, 0.55555, \
           0.66666, 0.77777, 0.88888");
}
receiver_capacitance1_fall ("template_8x1") {
    values("0.11111, 0.22222, 0.33333, 0.44444, 0.55555, \
           0.66666, 0.77777, 0.88888");
}
receiver_capacitance2_fall ("template_8x1") {
    values("0.11111, 0.22222, 0.33333, 0.44444, 0.55555, \
           0.66666, 0.77777, 0.88888");
}
}
...
} /* end pin A1 */

pin (A2) {
    direction : "input";
    ...
    /* pin-based ccs receiver model */
}
} /* end pin A2 */

pin (A3) {
    direction : "input";
    ...
    /* pin-based CCS receiver model */
    /* enumerate all conditions */
    receiver_capacitance () {
        when : "!A1 * A2";
        receiver_capacitance1_rise ("template_8x1") {
            values("0.11111, 0.22222, 0.33333, 0.44444, 0.55555, \
                   0.66666, 0.77777, 0.88888");
        }
        receiver_capacitance2_rise ("template_8x1") {
            values("0.11111, 0.22222, 0.33333, 0.44444, 0.55555, \
                   0.66666, 0.77777, 0.88888");
        }
        receiver_capacitance1_fall ("template_8x1") {
            values("0.11111, 0.22222, 0.33333, 0.44444, 0.55555, \
                   0.66666, 0.77777, 0.88888");
        }
        receiver_capacitance2_fall ("template_8x1") {
            values("0.11111, 0.22222, 0.33333, 0.44444, 0.55555, \
                   0.66666, 0.77777, 0.88888");
        }
    }
}
```

Chapter 10: Composite Current Source Modeling Two-Segment Receiver Capacitance Model

```

}
receiver_capacitance () {
    when : "A1 * !A2";
    receiver_capacitance1_rise ("template_8x1") {
        values("0.11111, 0.22222, 0.33333, 0.44444, 0.55555, \
                0.66666, 0.77777, 0.88888");
    }
    receiver_capacitance2_rise ("template_8x1") {
        values("0.11111, 0.22222, 0.33333, 0.44444, 0.55555, \
                0.66666, 0.77777, 0.88888");
    }
    receiver_capacitance1_fall ("template_8x1") {
        values("0.11111, 0.22222, 0.33333, 0.44444, 0.55555, \
                0.66666, 0.77777, 0.88888");
    }
    receiver_capacitance2_fall ("template_8x1") {
        values("0.11111, 0.22222, 0.33333, 0.44444, 0.55555, \
                0.66666, 0.77777, 0.88888");
    }
}
receiver_capacitance () {
    when : "!A1 * !A2";
    receiver_capacitance1_rise ("template_8x1") {
        values("0.11111, 0.22222, 0.33333, 0.44444, 0.55555, \
                0.66666, 0.77777, 0.88888");
    }
    receiver_capacitance2_rise ("template_8x1") {
        values("0.11111, 0.22222, 0.33333, 0.44444, 0.55555, \
                0.66666, 0.77777, 0.88888");
    }
    receiver_capacitance1_fall ("template_8x1") {
        values("0.11111, 0.22222, 0.33333, 0.44444, 0.55555, \
                0.66666, 0.77777, 0.88888");
    }
    receiver_capacitance2_fall ("template_8x1") {
        values("0.11111, 0.22222, 0.33333, 0.44444, 0.55555, \
                0.66666, 0.77777, 0.88888");
    }
}
...
} /* end pin A3 */
} /* end cell mycell */
} /* end library mylib */

```

Example 116 shows the use of the `char_when_rise` and `char_when_fall` attributes in the pin-level receiver capacitance model.

Example 116 char_when_rise and char_when_fall Attributes in receiver_capacitance Group

```

library (mylib) {
...
cell (mycell) {
    pin (Y) {

```

```
    direction : "output";
    function : "A1 * A2 * A3";
    ...
} /* end pin Y */
pin (A1) {
    direction : "input";
    /* pin-based ccs receiver model */
    receiver_capacitance () {
        char_when_rise : "!A2 * !A3";
        char_when_fall : "!A2 * A3";
        ...
    }
    ...
} /* end pin A1 */
...
} /* end cell mycell */
} /* end library mylib */
```

Defining the Receiver Capacitance Groups at the Timing Level

At the timing level, you do not need to define the `receiver_capacitance` group. Define the receiver capacitance for the timing arcs by using the `receiver_capacitance1_rise`, `receiver_capacitance1_fall`, `receiver_capacitance2_rise`, and `receiver_capacitance2_fall` groups in the `timing` group.

Conditional Data Modeling

Use the `mode` and `when` attributes in timing arcs for conditional timing arcs and constraints.

Defining the `lu_table_template` Group

Use this library-level group to create templates of common information that multiple lookup tables can use. A table template specifies the table parameters and the breakpoints for each axis. Assign each template a name so that lookup tables can refer to it.

lu_table_template Group

Define your lookup table templates in the `library` group.

lu_table_template Group Syntax

Define your lookup table templates in the `library` group.

```
library(name_id) {
    ...
    lu_table_template(template_name_id) {
        variable_1: input_net_transition;
        variable_2: total_output_net_capacitance;
        index_1 ("float,..., float");
        index_2 ("float,..., float");
        ...
    }
}
```

```
    }
    ...
}
```

Template Variables for CCS Receiver Models

The table template specifying composite current source receiver models can have only two variables: `variable_1` and `variable_2`. The parameters are the input transition time and the total output capacitance of a constrained pin.

The index values in the `index_1` and `index_2` attributes are a list of ascending floating-point numbers.

In the timing level, the table template specifying composite current source receiver models can have two variables: `variable_1` and `variable_2`. The valid values for either variable are `input_net_transition` and `total_output_net_capacitance`.

The index values in the `index_1` and `index_2` attributes are a list of ascending floating-point numbers.

The `values` attribute defines a list of floating-point numbers that represent the capacitance for the receiver model.

lu_table_template Example

```
...
    lu_table_template(LTT2) {
        variable_1: input_net_transition;
        variable_2: total_output_net_capacitance;
        index_1("0.1, 0.2, 0.4, 0.3");
        index_2("1.0, 2.0");
    }
```

Timing-Level receiver_capacitance Example

```
timing() /* timing arc-based receiver model*/
{
    ...
    related_pin : "B"
    receiver_capacitance1_rise(LTT2) {
        values("1.1, 4., 2.0, 3.2");
    }
    receiver_capacitance1_fall(LTT2) {
        values("1.0, 3.2, 4.0, 2.1");
    }
    receiver_capacitance2_rise(LTT2) {
        values("1.1, 4., 2.0, 3.2");
    }
    receiver_capacitance2_fall(LTT2) {
        values("1.0, 3.2, 4.0, 2.1");
    }
    ...
}
```

Multisegment Receiver Capacitance Model

In the multisegment receiver capacitance model, the rise or fall voltage at an input or inout node is divided into N segments, where N is an integer greater than 2. For each segment, the capacitance values are stored in a lookup table using the `receiver_capacitance_rise` or `receiver_capacitance_fall` groups.

The modeling syntax is:

```
library (library_name) {
    receiver_capacitance_rise_threshold_pct ("float,...");
    receiver_capacitance_fall_threshold_pct ("float,...");
    ...
    cell(cell_name) {
        ...
        /* The receiver_capacitance group can be under a pin or timing group. */

        pin (pin_name) {
            direction : input;
            ...
            /* pin based receiver model */
            receiver_capacitance () {
                when : boolean expression;
                mode (mode_name, mode_value);
                receiver_capacitance_rise (lu_template_name) {
                    segment : "integer";
                    ...
                }
                receiver_capacitance_fall (lu_template_name) {
                    segment : "integer";
                    ...
                }
            } /* end receiver_capacitance group */
            pin (pin_name) {
                direction : output;
                ...
                timing() {
                    ...
                    when : boolean_expression;
                    mode (mode_name, mode_value);
                    receiver_capacitance_rise (lu_template_name) {
                        segment : "integer";
                        ...
                    }
                    receiver_capacitance_fall (lu_template_name) {
                        segment : "integer";
                        ...
                    }
                } /* end timing group */
            } /* end pin group */
        } /* end cell group */
```

```
} /* end library group */
```

Library-Level Groups and Attributes

This section describes the library-level groups and attributes for multisegment receiver capacitance modeling.

lu_table_template Group

The `lu_table_template` group defines the template for the `receiver_capacitance_rise` and `receiver_capacitance_fall` groups specified in the `receiver_capacitance` and `timing` groups:

The template definition includes the `variable_1`, `variable_2`, `index_1`, and `index_2` attributes. The `variable_1` and `variable_2` attributes specify the index variables for the lookup tables (LUTs) in these groups. The `index_1` and `index_2` attributes specify the numerical values of the variables.

The values of `variable_1` and `variable_2` attributes are `input_net_transition` and `total_output_net_capacitance`.

You can specify the following types of lookup tables within these groups:

- One-dimensional tables with a single index, such as `input_net_transition`
- Two-dimensional tables with the indexes, `input_net_transition` and `total_output_net_capacitance`
- A scalar that has a single variation value irrespective of the `input_net_transition` and `total_output_net_capacitance` values

receiver_capacitance_rise_threshold_pct and receiver_capacitance_fall_threshold_pct Attributes

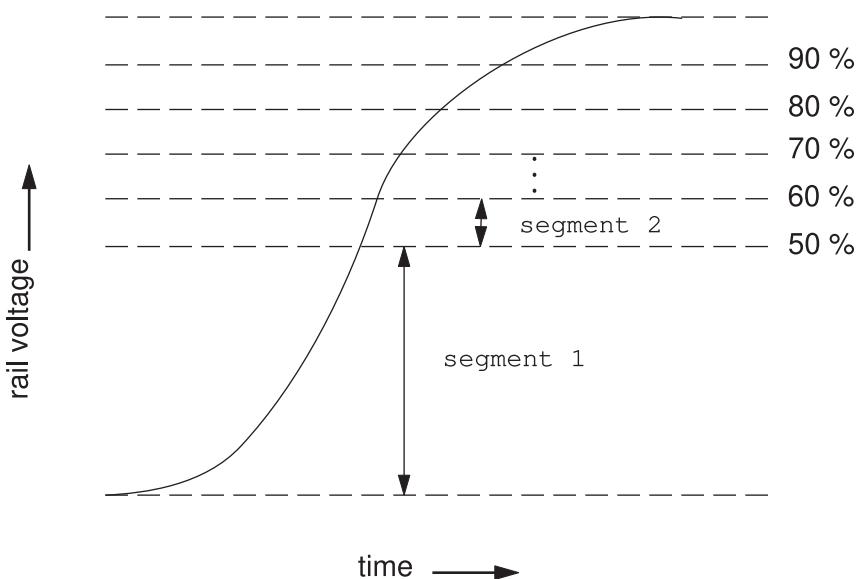
The `receiver_capacitance_rise_threshold_pct` and `receiver_capacitance_fall_threshold_pct` attributes specify the points that separate the voltage rise and fall segments. Specify each point as a percentage of the rail voltage between 0.0 and 100.0.

Specify monotonically increasing values with the `receiver_capacitance_rise_threshold_pct` attribute, and monotonically decreasing values with the `receiver_capacitance_fall_threshold_pct` attribute. For example,

```
receiver_capacitance_rise_threshold_pct ("0, 50, 60, 70, 80, 90, 100") ;  
receiver_capacitance_fall_threshold_pct ("100, 50, 40, 30, 20, 10, 0") ;
```

The number of segments, N, is one less than the number of points. So, N is 6. The following figure shows the segments for a voltage rise waveform.

Figure 101 Segments in a Multisegment Receiver Capacitance Model



Pin and Timing Level Groups

This section describes the pin and timing level groups and attributes for multisegment receiver capacitance modeling.

receiver_capacitance Group

At the pin level, define the `receiver_capacitance_rise` and `receiver_capacitance_fall` groups in the `receiver_capacitance` group.

Each of these groups specify a lookup table. Define the template of the lookup table in the library-level `lu_table_template` group. To use the template, specify the template name as the `receiver_capacitance_rise` or `receiver_capacitance_fall` group name.

Note:

At the timing level, define the `receiver_capacitance_rise` and `receiver_capacitance_fall` groups in the timing group.

receiver_capacitance_rise and receiver_capacitance_fall Groups

The `receiver_capacitance_rise` and `receiver_capacitance_fall` groups specify the receiver capacitance values of each of the N rise and fall voltage segments, in lookup tables. Specify N `receiver_capacitance_rise` groups, each for a unique segment. The `receiver_capacitance_fall` group has the same requirement.

The `receiver_capacitance_rise` and `receiver_capacitance_fall` groups are structurally similar to the `receiver_capacitance1_rise`, `receiver_capacitance2_rise`, `receiver_capacitance1_fall`, and `receiver_capacitance2_fall` groups and include all of their attributes and subgroups.

segment Attribute

The `segment` attribute specifies the segment that the `receiver_capacitance_rise` or the `receiver_capacitance_fall` group represents. The values vary from 1 to N.

Conditional Data Modeling

You can use the `mode` and `when` attributes in both the `receiver_capacitance` and timing groups.

when Attribute

The `when` attribute specifies the Boolean condition for the input state of the receiver capacitance timing arc.

mode Attribute

The complex `mode` attribute specifies the current mode of the cell. When defined in the `receiver_capacitance` group, the `mode` attribute specifies the receiver capacitance for the given mode. If you specify the `mode` attribute, you must define the `mode_definition` group at the cell level.

Example

The following example shows a typical multisegment receiver capacitance model.

```
library (mylib) {
    receiver_capacitance_rise_threshold_pct ("0, 50, 60, 70, 80, 90,
100");
    receiver_capacitance_fall_threshold_pct ("100, 50, 40, 30, 20, 10, 0");
    /* receiver model template */
    lu_table_template ("delay_template_8x1") {
        variable 1 : "input_net_transition";
        index_1("0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8");
    }
    lu_table_template ("delay_template_8x8") {
        variable_1 : "input_net_transition";
        variable_2 : "total_output_net_capacitance";
        index_1("0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8");
        index_2("0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8");
    }
    ...
    cell (flop) {
        pin (D) {
            direction : input;
            /* pin-based CCS receiver model */
        }
    }
}
```

Chapter 10: Composite Current Source Modeling
Multisegment Receiver Capacitance Model

```
receiver_capacitance () {
/* default condition */
/* The old model groups are shown for comparison
 receiver_capacitance1_rise(lu_template_name) {...}
 receiver_capacitance2_rise(lu_template_name) {...}
 receiver_capacitance1_fall(lu_template_name) {...}
 receiver_capacitance2_fall(lu_template_name) {...}
*/
    receiver_capacitance_rise ("delay_template_8x1") {
        segment : 1;
        values("0.11111, 0.22222, 0.33333, 0.44444, \
               0.55555, 0.66666, 0.77777, 0.88888");
    }
    ...
    receiver_capacitance_rise ("delay_template_8x1") {
        segment : 6;
        values("0.1111, 0.2222, 0.3333, 0.4444, \
               0.5555, 0.6666, 0.77777, 0.88888");
    }
    receiver_capacitance_fall ("delay_template_8x1") {
        segment : 1;
        values("0.11111, 0.22222, 0.33333, 0.44444, \
               0.55555, 0.66666, 0.77777, 0.88888");
    }
    ...
    receiver_capacitance_fall ("delay_template_8x1") {
        segment : 6;
        values("0.1111, 0.2222, 0.3333, 0.4444, \
               0.5555, 0.6666, 0.7777, 0.8888");
    }
} /* end receiver_capacitance group */
} /* end pin D group */
pin (CLK) {
    direction : input;
    ...
}
pin (Y) {
    direction : output;
    function : "IQ";
    ...
    timing () {
        related_pin : "CLK";
        ...
        /* arc-based ccs receiver model */
        receiver_capacitance_rise ("delay_template_8x8") {
            segment : 1;
            values("0.11111, 0.22222, 0.33333, 0.44444, \
                   0.55555, 0.66666, 0.77777, 0.88888, ...");
        }
        ...
        receiver_capacitance_rise ("delay_template_8x8") {
            segment : 6;
            values("0.11111, 0.22222, 0.33333, 0.44444, \
                   0.55555, 0.66666, 0.77777, 0.88888, ...");
        }
    }
}
```

```

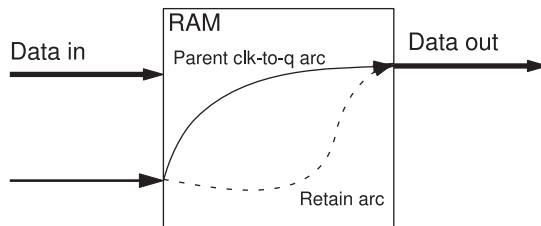
        0.55555, 0.66666, 0.77777, 0.88888, ...");
}
receiver_capacitance_fall ("delay_template_8x8") {
    segment : 1;
    values("0.11111, 0.22222, 0.33333, 0.44444, \
            0.55555, 0.66666, 0.77777, 0.88888,...");
}
...
receiver_capacitance_fall ("delay_template_8x8") {
    segment : 6;
    values("0.11111, 0.22222, 0.33333, 0.44444, \
            0.55555, 0.66666, 0.77777, 0.88888,...");
}
} /* end timing group */
} /* end pin Y group*/
} /* end cell flop group*/
} /* end library mylib */

```

CCS Retain Arc Support

A *retain delay* is the shortest delay among all parallel arcs, from the input port to the output port. Access time is the longest delay from the input port to the output port. The output value is uncertain and unstable in the time interval between the retain delay and the access time. A retain arc, as shown in Figure 102, ensures that the output does not change during this time interval.

Figure 102 Retain Arc Example



Retain arcs:

- Guarantee that the output does not change for a certain time interval.
- Are usually defined for memory cells.
- Are not inferred as a timing check but are inferred as a delay arc.

Liberty syntax supports retain arcs in nonlinear delay models by providing the following timing groups:

- retaining_rise

- retaining_fall
- retain_rise_slew
- retain_fall_slew

CCS Retain Arc Syntax

The following syntax supports CCS timing models including the expanded and compact CCS timing models. Because retain arcs have no relation to CCS receiver models, only syntax for CCS driver models is described as follows:

Syntax

```
library (library_name) {
    delay_model : table_lookup;
...
    output_current_template(template_name) {
        variable_1: input_net_transition;
        variable_2: total_output_net_capacitance;
        variable_3: time;
        index_1("float, ..., float"); /* optional at library level */
        index_2("float, ..., float"); /* optional at library level*/
        index_3("float, ..., float"); /* optional at library level*/
    }

    cell(name) {
        pin (name) {
            timing() {
                ccs_retain_rise() {
                    vector(template_name) {
                        reference_time : float;
                        index_1("float");
                        index_2("float");
                        index_3("float, ..., float");
                        values("float, ..., float");
                    }
                    vector(template_name) { . . . } ...
                }
                ccs_retain_fall() {
                    vector(template_name) {
                        reference_time : float;
                        index_1("float");
                        index_2("float");
                        index_3("float, ..., float");
                        values("float, ..., float");
                    }
                    vector(template_name) { . . . } ...
                }
                output_current_rise() { ... }
                output_current_fall() { ... }
            }
        }
    }
}
```

```
        }
    }
}
. . .
}
```

The format of the expanded CCS retain arc group is the same as the general CCS timing arcs that are defined by using the `output_current_rise` and `output_current_fall` groups.

`ccs_retain_rise` and `ccs_retain_fall` Groups

The `ccs_retain_rise` and `ccs_retain_fall` groups are provided in the timing group for expanded CCS retain arcs.

vector Group

The current `vector` group in the `ccs_retain_rise` and `ccs_retain_fall` groups uses the lookup table template defined by `output_current_template`. The `vector` group has the following parameters:

- `input_net_transition`
- `total_output_net_capacitance`
- `time`

For every value pair (such as `input_net_transition` and `total_output_net_capacitance`), there is a specified `current(time)` vector.

`reference_time` Attribute

The `reference_time` simple attribute specifies the time that the input signal waveform crosses the rising or falling input delay threshold.

Compact CCS Retain Arc Syntax

The compact CCS retain arc format is the same as a general compact CCS timing arc. The following retain arc syntax supports compact CCS timing.

Syntax

```
library(my_lib) {
...
base_curves (base_curves_name) {
    base_curve_type: enum (ccs_timing_half_curve);
    curve_x ("float..., float");
    curve_y (integer, "float..., float");
    curve_y (integer, "float..., float");
...
}
```

```
}

compact_lut_template(template_name) {
    base_curves_group: base_curves_name;
    variable_1 : input_net_transition;
    variable_2 : total_output_net_capacitance;
    variable_3 : curve_parameters;
    index_1 ("float..., float");
    index_2 ("float..., float");
    index_3 ("string..., string");
}
...

cell(cell_name) {
    ...
    pin(pin_name) {
        direction : string;
        capacitance : float;
        timing() {
            compact_ccs_retain_rise (template_name) {
                base_curves_group : "base_curves_name";
                index_1 ("float..., float");
                index_2 ("float..., float");
                index_3 ("string..., string");
                values ("..."...)
            }
            compact_ccs_retain_fall (template_name) {
                base_curves_group : "base_curves_name";
                index_1 ("float..., float");
                index_2 ("float..., float");
                index_3 ("string..., string");
                values ("..."...)
            }
            compact_ccs_rise(template_name) { ... }
            compact_ccs_fall(template_name) { ... }
            ...
        } /*end of timing */
    } /*end of pin */
} /*end of cell */

...
}/* end of library*/
```

compact_ccs_retain_rise and compact_ccs_retain_fall Groups

The `compact_ccs_retain_rise` and `compact_ccs_retain_fall` groups are provided in the timing group for compact CCS retain arcs.

base_curves_group Attribute

The `base_curves_group` attribute is optional. The attribute is required when `base_curves_name` is different from that defined in the `compact_lut_template` template name.

index_1, index_2, and index_3 Attributes

The values for the `index_1` and `index_2` attributes are `input_net_transition` and `total_output_net_capacitance`.

The values for the `index_3` attribute must contain the following base curve parameters:

- `init_current`
- `peak_current`
- `peak_voltage`
- `peak_time`
- `left_id`
- `right_id`

values Attribute

The `values` attribute provides the compact CCS retain arc data values. The `left_id` and `right_id` values for compact CCS timing base curves should be integers, and they must be predefined in the `base_curves` group.

Composite Current Source Driver and Receiver Model Example

[Example 117](#) is an example of composite current source driver and receiver model syntax.

Example 117 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*/
```

Chapter 10: Composite Current Source Modeling Composite Current Source Driver and Receiver Model Example

```
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" \
                    "3.0, 2.3");
            }
            receiver_capacitance1_fall(LTT2) {
                values("1.1, 2.3" \
                    "1.3, 0.4");
            }
            receiver_capacitance2_rise(LTT2) {
                values("1.3, 0.2" \
                    "1.3, 0.4");
            }
            receiver_capacitance2_fall(LTT2) {
                values("1.3, 2.1" \
                    "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;
```

Chapter 10: Composite Current Source Modeling
Composite Current Source Driver and Receiver Model Example

```
    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) */
.
.
```

Chapter 10: Composite Current Source Modeling
Composite Current Source Driver and Receiver Model Example

```
    } /* end of cell */  
    . . .  
} /* end of library */
```

11

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](#)
 - [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. To achieve high accuracy, this driver model maps the transistor behavior for lumped loads to that for parasitic network instead of modeling the transistor behavior.

The composite current source model has better receiver model accuracy because the input capacitance of a receiver is dynamically adjusted during the transition by using two capacitance values. You can use the driver model with or without the receiver model.

Compact CCS Timing Model

Conventional CCS timing driver models require you to describe each CCS driver switching current waveform by sampling data points. As the number of timing arcs increase in a standard cell library, the CCS timing library size can become very large.

This section describes the syntax of a compact model that uses indirectly shared base curves to model the shape of switching curves. By allowing each base curve to model multiple switching curves with similar shapes, the modeling efficiency is improved and the library size is compressed.

The topics in the following sections include:

- Describing CCS timing base curves.

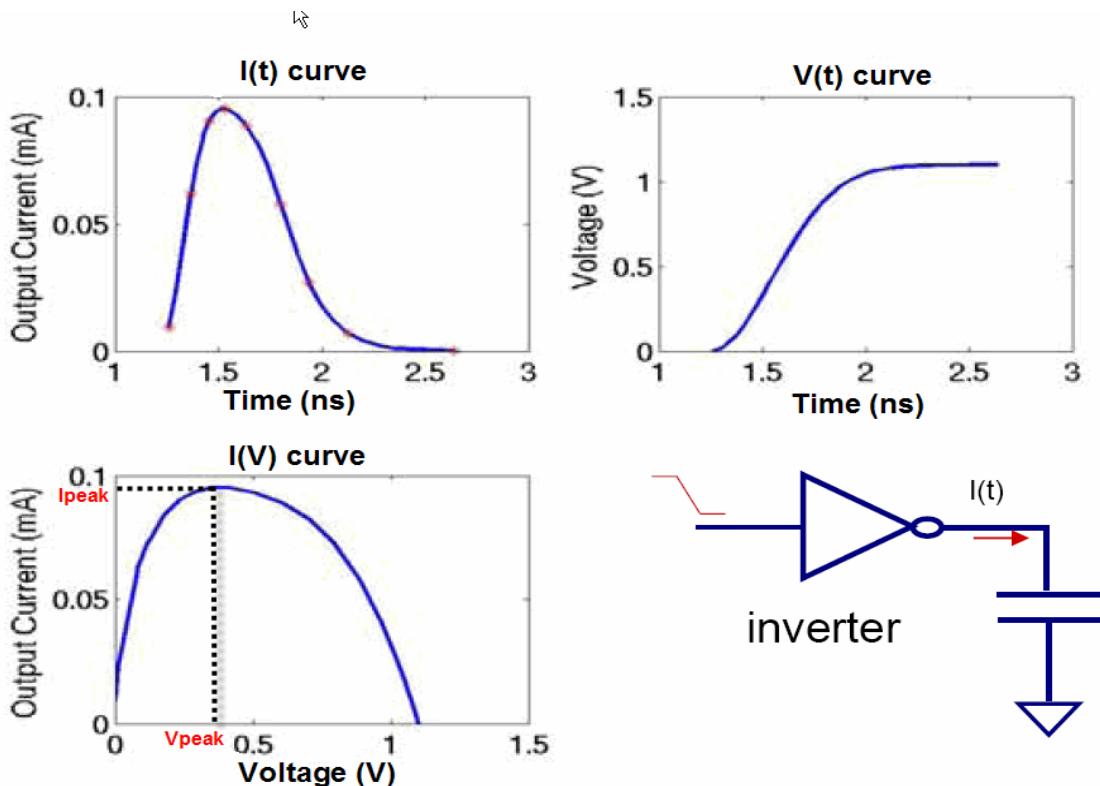
- Describing the syntax of base curves and the compact CCS driver modeling format.

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.

In Figure 103, the CCS segmentation process samples nine data points from the $I(t)$ curve based on the tolerance. 18 floating-point numbers, that is, nine time points and nine current points are stored in the CCS library.

Figure 103 I(t), V(t), and I-V Curves for Inverter Cell Rise Transition With Existing CCS Format



In Figure 104, 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. The six parameters are as follows:

Iinit

Switching current value at the starting point.

Ipeak

Peak switching current value.

Vpeak

Voltage value when current reaches peak value.

Tpeak

Time when current reaches peak value.

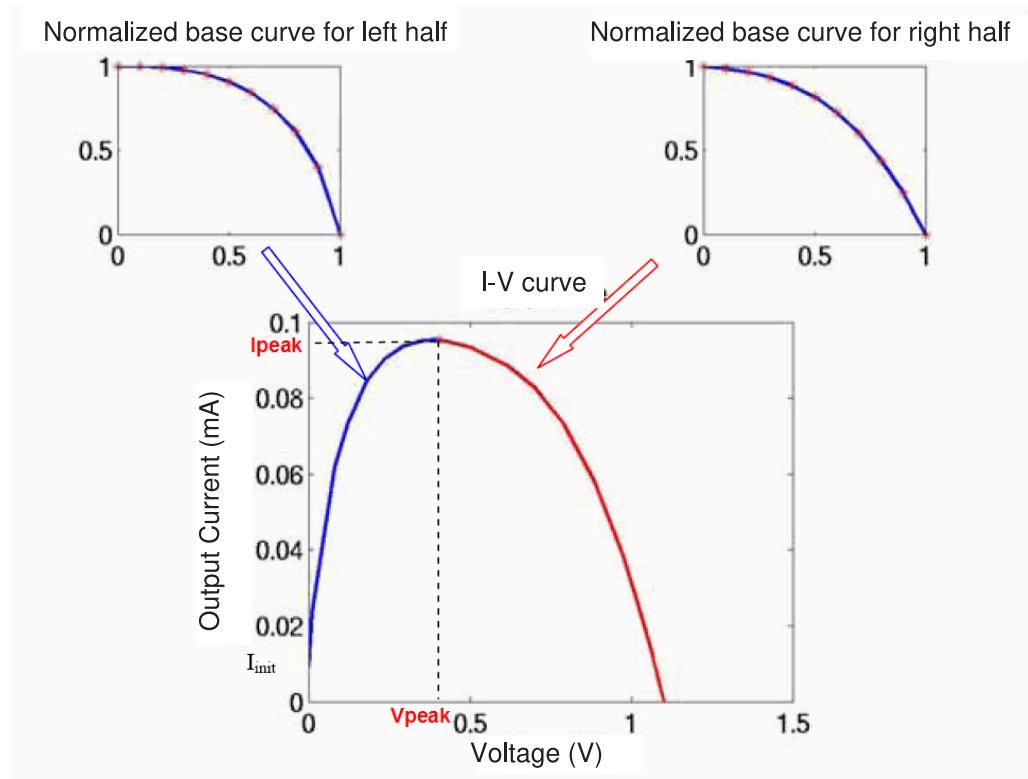
Left id

Reference id of the base curve that matches the left half.

Right id

Reference id of the base curve that matches the right half.

Figure 104 Using Base Curves to Simplify I-V Curve Modeling



Compact CCS Timing Model Syntax

In the figure (in [Modeling With CCS Timing Base Curves](#)), each switching I-V curve can be modeled by normalized base curves using the following parameters: `init_current`, `peak_current`, `peak_voltage`, `peak_time`, `left_id`, and `right_id`.

The `init_current`, `peak_current`, `peak_voltage`, and `peak_time` parameters are the critical characteristics of the curve. The `left_id` and `right_id` describe the two base curves that represent the two halves of the I-V curve.

The syntax for compact CCS timing model is as follows:

```
library(my_compact_ccs_lib) {  
...  
base_curves (base_curves_name) {  
    base_curve_type: enum (ccs_timing_half_curve);  
    curve_x ("float..., float");  
    curve_y (curve_id, "float..., float");  
    curve_y (curve_id, "float..., float");  
...  
    curve_y (curve_id, "float..., float");  
}
```

```
}

compact_lut_template(template_name) {
    base_curves_group: base_curves_name;
    variable_1 : input_net_transition |
    total_output_net_capacitance;
    variable_2 : input_net_transition |
    total_output_net_capacitance;
    variable_3 : curve_parameters;
    index_1 ("float..., float");
    index_2 ("float..., float");
    index_3 ("string..., string");
}
...

cell(cell_name) {
...
pin(pin_name) {
    direction : string;
    capacitance : float;
    timing() {
        /*Compact CCS arc-based driver model*/
        compact_ccs_rise (template_name) {
            base_curves_group : "base_curves_name";
            values (... , ... \
            "...", "... \
            "...", "... \
            "...", ...)
        }/*end of compact_ccs_rise() */
        compact_ccs_fall(template_name) {
            ...
        }/*end of compact_ccs_fall() */
        ...
    } /*end of timing */
} /*end of pin */
} /*end of cell */
...
} /* end of library*/
```

The groups described in the following sections support compact CCS timing models.

base_curves Group

The `base_curves` library-level group contains the detailed description of normalized base curves. The `base_curves` group has the following attributes:

base_curve_type Complex Attribute

The `base_curve_type` attribute specifies the type of base curve. The only valid value for this attribute is `ccs_timing_half_curve`.

curve_x complex Attribute

The data array contains the x-axis values of the normalized base curve. Only one `curve_x` value is allowed for each `base_curves` group. See the figure (in [Modeling With CCS Timing Base Curves](#)), 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 the figure in [Modeling With CCS Timing Base Curves](#)) is composed of one `curve_x` and one `curve_y`. You should define the `curve_x` base curve before `curve_y` for better clarity and easier implementation.

There are two data sections in the `curve_y` complex attribute:

- `curve_id` is the identifier of the base curve.
- Data array is the y-axis values of the normalized base curve.

compact_lut_template Group

The `compact_lut_template` group is used for compact CCS timing modeling. (The `lu_table_template` group is used for other timing models.) The `compact_lut_template` group has the following attributes:

base_curves_group Attribute

This is a required attribute in the `compact_lut_template` group. Its value should be a predefined `base_curves` group name. The `base_curve_type` of `base_curves` group determines the values in `index_3`. It also determines where you can use this template.

variable_1 and variable_2 Attributes

Only `input_net_transition` and `total_output_net_capacitance` are valid values for `variable_1` and `variable_2`.

variable_3 Attribute

Only `curve_parameters` is a string value for this variable.

index_1 and index_2 Attributes

These are required attributes. They define the `input_net_transition` or `total_output_net_capacitance` values using the float notation.

index_3 Attribute

String values in `index_3` are determined by the `base_curve_type` in `base_curves` group. When the `base_curve_type` is a `ccs_timing_half_curve`, at least six string parameters

should be defined. They are `init_current`, `peak_current`, `peak_voltage`, `peak_time`, `left_id`, and `right_id`. If any of these six parameters are missing, a compilation error is issued.

compact_ccs_{rise|fall} Group

This is the compact CCS timing data in timing arc group, which has the following attributes:

base_curves_group Attribute

Defining this attribute is optional when the `base_curves_name` is same as that defined in the `compact_lut_template` group. This group is referenced by the `compact_ccs_{rise|fall}` group.

values Attribute

Values of compact CCS timing data depend on how you define `index_3` values.

For compact CCS timing, base curves data `left_id` and `right_id` values can only be integers.

Compact CCS Timing Library Example

```
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.

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 sections include information about

- Variation-aware modeling for compact CCS timing driver models
- Variation-aware modeling for CCS timing receivers
- Variation-aware modeling for regular or interdependent timing constraints
- Conditional data modeling for variation-aware timing receiver models

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 should 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 */
```

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

The `va_parameters` attribute specifies a list of variation parameters with the following rules:

- 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 the `va_parameters` attribute is specified at the library level, all cells under the library default to the same variation parameters.
- If the `va_parameters` attribute is defined in the variation group, all `va_values` and `nominal_va_values` under the same variation group refers to this `va_parameters`.

The attribute values can be user-defined or predefined parameters. For more information, see the examples in [Variation-Aware Syntax Examples](#).

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 Complex Attribute](#)”.

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`, these attributes are considered 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`, these attributes are taken 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 one-to-one 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.

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.

- 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 examples in [Variation-Aware Syntax Examples](#).
- All variation-aware compact CCS driver model groups inside the `timing_based_variation` share the same `va_parameters`.

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 one-to-one mapping to the corresponding `va_parameters`.

For an example that shows how to specify `va_values` with three variation parameters, see examples in [Variation-Aware Syntax Examples](#). 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.

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 569](#) 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 examples in [Variation-Aware Syntax Examples](#).

- 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.

- 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`.

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 one-to-one mapping to the corresponding `va_parameters` attribute.

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. They are defined in the following syntax and sections.

```
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) {
```

Chapter 11: Advanced Composite Current Source Modeling
Variation-Aware Timing Modeling Support

```
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 the `pin_based_variation` group.
- If nominal CCS receiver model groups exist in the timing group, variation-aware CCS receiver model groups are required in the `timing_based_variation` group.

va_parameters Complex Attribute

This attribute specifies a list of variation parameters within the `timing_based_variation` or `pin_based_variation` groups. See [va_parameters Attribute on page 569](#) for more details.

nominal_va_values Complex Attribute

This complex attribute specifies the nominal values of all variation parameters.

- The attribute is required for 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 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 the examples in [Variation-Aware Syntax Examples](#).
- 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.

- You should characterize two corners at each side of the nominal value of all variation parameters specified in `va_parameters` attribute.

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 one-to-one mapping to the corresponding `va_parameters` attribute.

Variation-Aware Timing Constraint Modeling

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, which is defined in the following 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 569](#) 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 570](#) 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 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` attribute.
- You are expected to characterize two corners at each side of the nominal value of all variation parameters as specified in `va_parameters` attribute.

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 the `va_rise_constraint` and `va_fall_constraint` groups in the `timing_based_variation` group share the same `va_parameters` attribute.

va_values Attribute

Specifies values of each variation parameter for all corners characterized in the va_rise_constraint and va_fall_constraint groups.

- Required for the va_rise_constraint and va_fall_constraint groups.
- The value of this attribute has a one-to-one mapping to the corresponding va_parameters attribute.

Conditional Data Modeling for Variation-Aware Timing Receiver Models

Liberty provides the following syntax to support conditional data modeling for pin-based variation-aware timing receiver models:

```
library(library_name) {
    ...
    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, ...);
    ...

    cell(cell_name) {
        mode_definition(mode_name) {
            mode_value(namestring) {
                when : "boolean expression";
                sdf_cond : "boolean expression";
            }
        ...
    }
    pin(pin_name) {
        direction : input; /* or "inout" */
        receiver_capacitance() {
            when : "boolean expression";
            mode (mode_name, mode_value);
            receiver_capacitance1_rise (template_name) { ... }
            receiver_capacitance1_fall (template_name) { ... }
            receiver_capacitance2_rise (template_name) { ... }
            receiver_capacitance2_fall (template_name) { ... }
        }
        pin_based_variation() {
            /* The "when" and "mode" attributes should be exactly the same as
            defined in the
            receiver_capacitance group above */
        }
    }
}
```

```
when : "boolean expression";
mode (mode_name, mode_value);
va_parameters(string, ...);
nominal_va_values(float, ...);
va_receiver_capacitance1_rise (pin_based_template_name) { ... }
va_receiver_capacitance1_fall (pin_based_template_name) { ... }
va_receiver_capacitance2_rise (pin_based_template_name) { ... }
va_receiver_capacitance2_fall (pin_based_template_name) { ... }
...
} /* end of pin_based_variation */
...
} /* end of pin group */
pin(pin_name) {
    direction : output; /* or "inout" */
    timing() {
when : "boolean expression";
mode (mode_name, mode_value);
...
    receiver_capacitance() {
        receiver_capacitance1_rise (template_name) { ... }
        receiver_capacitance1_fall (template_name) { ... }
        receiver_capacitance2_rise (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_receiver_capacitance1_fall (timing_based_template_name) { ... }
        va_receiver_capacitance2_rise (timing_based_template_name) { ... }
        va_receiver_capacitance2_fall (timing_based_template_name) { ... }
    }
    ...
} /* end of timing_based_variation */
}
...
} /* end of pin group */
} /* end of cell group */
...
} /* end of library */
```

when Attribute

The `when` string attribute is provided in the `pin_based_variation` group to support conditional data modeling.

mode Attribute

The `mode` complex attribute is provided in the `pin_based_variation` group to support conditional data modeling. If the `mode` attribute is specified, `mode_name` and `mode_value` must be predefined in the `mode_definition` group at the cell level.

The following example shows conditional data modeling for pin-based variation-aware timing receiver models:

```
library(new_lib) {
    ...
    output_current_template(CCT) {
        variable_1: input_net_transition;
        variable_2: total_output_net_capacitance;
        variable_3: time;
        index_1("0.1, 0.2");
        index_2("1, 2");
        index_3("1, 2, 3, 4, 5");
    }
    lu_table_template(LTT1) {
        variable_1: input_net_transition;
        index_1("0.1, 0.2, 0.3, 0.4");
    }
    lu_table_template(LTT2) {
        variable_1: input_net_transition;
        variable_2: total_output_net_capacitance;
        index_1("0.1, 0.2");
        index_2("1, 2");
    }
}
...
cell(my_cell) {
    ...
    mode_definition(rw) {
        mode_value(read) {
            when : "I";
            sdf_cond : "I == 1";
        }
        mode_value(write) {
            when : "!I";
            sdf_cond : "I == 0";
        }
    }
    pin(I) /* pin-based receiver model defined for pin 'A' */
    direction : input;
    /* receiver capacitance for condition 1 */
    receiver_capacitance() {
        when : "I"; /* or using mode as next commented line */
        /* mode (rw, read); */
        receiver_capacitance1_rise(LTT1) {
            values("1, 2, 3, 4");
        }
        receiver_capacitance1_fall(LTT1) {
            values("1, 2, 3, 4");
        }
        receiver_capacitance2_rise(LTT1) {
            values("1, 2, 3, 4");
        }
        receiver_capacitance2_fall(LTT1) {
            values("1, 2, 3, 4");
        }
    }
}
```

Chapter 11: Advanced Composite Current Source Modeling
Variation-Aware Timing Modeling Support

```
        }
    }
pin_based_variation ( ) {
    when : "I"; /* or using mode as next commented line */
/* mode (rw, read); */
va_parameters(channel_length, threshold_voltage);

nominal_va_values(0.5, 0.5) ;

va_receiver_capacitance1_rise (LTT1) {
    va_values(0.50, 0.45);
    values("1, 2, 3, 4");
}
va_receiver_capacitance1_rise (LTT1) {
    va_values(0.50, 0.55);
    values("1, 2, 3, 4");
}
va_receiver_capacitance1_rise (LTT1) {
    va_values(0.45, 0.5);
    values("1, 2, 3, 4");
}
va_receiver_capacitance1_rise (LTT1) {
    va_values(0.55, 0.5);
    values("1, 2, 3, 4");
}

va_receiver_capacitance2_rise (LTT1) {
    va_values(0.50, 0.45);
    values("1, 2, 3, 4");
}
va_receiver_capacitance2_rise (LTT1) {
    va_values(0.50, 0.55);
    values("1, 2, 3, 4");
}
va_receiver_capacitance2_rise (LTT1) {
    va_values(0.45, 0.5);
    values("1, 2, 3, 4");
}
va_receiver_capacitance2_rise (LTT1) {
    va_values(0.55, 0.5);
    values("1, 2, 3, 4");
}

va_receiver_capacitance1_fall (LTT1) {
    va_values(0.50, 0.45);
    values("1, 2, 3, 4");
}
va_receiver_capacitance1_fall (LTT1) {
    va_values(0.50, 0.55);
    values("1, 2, 3, 4");
}
va_receiver_capacitance1_fall (LTT1) {
    va_values(0.45, 0.5);
```

Chapter 11: Advanced Composite Current Source Modeling
Variation-Aware Timing Modeling Support

```
    values("1, 2, 3, 4");
}
va_receiver_capacitance1_fall (LTT1) {
    va_values(0.55, 0.5);
    values("1, 2, 3, 4");
}

va_receiver_capacitance2_fall (LTT1) {
    va_values(0.50, 0.45);
    values("1, 2, 3, 4");
}
va_receiver_capacitance2_fall (LTT1) {
    va_values(0.50, 0.55);
    values("1, 2, 3, 4");
}
va_receiver_capacitance2_fall (LTT1) {
    va_values(0.45, 0.5);
    values("1, 2, 3, 4");
}
va_receiver_capacitance2_fall (LTT1) {
    va_values(0.55, 0.5);
    values("1, 2, 3, 4");
}

/* receiver capacitance for condition 2 */
receiver_capacitance() {
    when : "!I"; /* or using mode as next commented line */
    /* mode (rw, write); */
    receiver_capacitance1_rise(LTT1) {
        values("1, 2, 3, 4");
    }
    receiver_capacitance1_fall(LTT1) {
        values("1, 2, 3, 4");
    }
    receiver_capacitance2_rise(LTT1) {
        values("1, 2, 3, 4");
    }
    receiver_capacitance2_fall(LTT1) {
        values("1, 2, 3, 4");
    }
}
pin_based_variation () {
    when : "!I"; /* or using mode as next commented line */
    /* mode (rw, write); */

    va_parameters(channel_length, threshold_voltage);

    nominal_va_values(0.5, 0.5) ;

    va_receiver_capacitance1_rise (LTT1) {
        va_values(0.50, 0.45);
        values("1, 2, 3, 4");
    }
}
```

Chapter 11: Advanced Composite Current Source Modeling
Variation-Aware Timing Modeling Support

```
        }
va_receiver_capacitance1_rise (LTT1) {
    va_values(0.50, 0.55);
    values("1, 2, 3, 4");
}
va_receiver_capacitance1_rise (LTT1) {
    va_values(0.45, 0.5);
    values("1, 2, 3, 4");
}
va_receiver_capacitance1_rise (LTT1) {
    va_values(0.55, 0.5);
    values("1, 2, 3, 4");
}

va_receiver_capacitance2_rise (LTT1) {
    va_values(0.50, 0.45);
    values("1, 2, 3, 4");
}
va_receiver_capacitance2_rise (LTT1) {
    va_values(0.50, 0.55);
    values("1, 2, 3, 4");
}
va_receiver_capacitance2_rise (LTT1) {
    va_values(0.45, 0.5);
    values("1, 2, 3, 4");
}
va_receiver_capacitance2_rise (LTT1) {
    va_values(0.55, 0.5);
    values("1, 2, 3, 4");
}

va_receiver_capacitance1_fall (LTT1) {
    va_values(0.50, 0.45);
    values("1, 2, 3, 4");
}
va_receiver_capacitance1_fall (LTT1) {
    va_values(0.50, 0.55);
    values("1, 2, 3, 4");
}
va_receiver_capacitance1_fall (LTT1) {
    va_values(0.45, 0.5);
    values("1, 2, 3, 4");
}
va_receiver_capacitance1_fall (LTT1) {
    va_values(0.55, 0.5);
    values("1, 2, 3, 4");
}

va_receiver_capacitance2_fall (LTT1) {
    va_values(0.50, 0.45);
    values("1, 2, 3, 4");
}
va_receiver_capacitance2_fall (LTT1) {
```

```
        va_values(0.50, 0.55);
        values("1, 2, 3, 4");
    }
    va_receiver_capacitance2_fall (LTT1) {
        va_values(0.45, 0.5);
        values("1, 2, 3, 4");
    }
    va_receiver_capacitance2_fall (LTT1) {
        va_values(0.55, 0.5);
        values("1, 2, 3, 4");
    }
}
pin (ZN) {
    direction : input;
    capacitance : 1.2;
    ...
    timing() {
        ...
    }
}
...
} /* end cell */
...
} /* end library */
```

Variation-Aware Compact CCS Retain Arcs

Variation-aware timing models include:

- Timing-based modeling for compact CCS timing drivers.
- Timing-based and pin-based modeling for CCS timing receivers.
- Timing-based modeling for regular or interdependent timing constraints.

Liberty provides the following syntax in the `timing_based_variation` group to support retain arcs for compact CCS driver models:

```
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;
        variable_2 : total_output_net_capacitance;
        variable_3 : curve_parameters;
```

```
    ...
}
va_parameters(string , ...);
...
cell(cell_name) {
    ...
    pin(pin_name) {
        direction : string;
        capacitance : float;
        timing() {
            compact_ccs_rise(template_name) { ... }
            compact_ccs_fall(template_name) { ... }
            timing_based_variation() {
                va_parameters(string , ... );
                nominal_va_values(float, ...);
                va_compact_ccs_retain_rise(template_name) {
                    va_values(float, ...);
                    values ("..., float, ..., integer, ...", ...);
                }
                ...
                va_compact_ccs_retain_fall(template_name) {
                    va_values(float, ...);
                    values ("..., float, ..., integer, ...", ...);
                }
            }
            ...
            va_compact_ccs_rise(template_name) { ... } ...
            va_compact_ccs_fall(template_name) { ... } ...
        } /* end of timing_based_variation group */
        ...
    } /* end of pin group */
    ...
} /* end of cell group */
...
} /* end of library group*/
```

The format of variation-aware compact CCS retain arcs is the same as general variation-aware compact CCS timing arcs.

va_compact_ccs_retain_rise and va_compact_ccs_retain_fall Groups

The `va_compact_ccs_retain_rise` and `va_compact_ccs_retain_fall` groups in the `timing_based_variation` group specify characterization corners with variation value parameters for retain arcs.

va_values Attribute

The `va_values` attribute defines the values of each variation parameter for all corners characterized in variation-aware compact CCS retain arcs. The value of this attribute is mapped one-to-one to the corresponding `va_parameters`.

values Attribute

The `values` attribute follows the same rules as general variation-aware compact CCS timing models.

Variation-Aware Syntax Examples

va_parameters in Advanced CCS Modeling Usage

Example 118 va_parameters in Advanced CCS Modeling Usage

```
library (example) {
    ...
    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 (example) {
    ...
    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
    "voltage" is taken as an user-defined parameter. */

    va_parameters(VDD1, GND2, voltage,...);
```

For information about `va_parameters`, see [va_parameters Attribute on page 569](#).

Example 119 va_parameters in Advanced CCS Modeling Usage

```
library (example) {
    ...
}
```

```
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 (example) {
...
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
"voltage" is taken as an user-defined parameter. */
va_parameters(VDD1, GND2, voltage,...);
```

For information about `va_parameters`, see [va_parameters Complex Attribute](#).

va_compact_ccs_rise and va_compact_ccs_fall Groups

Example 120 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 [va_compact_ccs_rise and va_compact_ccs_fall Groups](#).

va_values With Three Variation Parameters

Example 121 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 [va_compact_ccs_rise and va_compact_ccs_fall Groups on page 570](#).

peak_voltage in Values Attribute

Example 122 peak_voltage in Values Attribute

```
...
library(va_ccs) {
    compact_lut_template(clt) {
        index_3 ("init_current, peak_current, peak_voltage, peak_time,\n"
                 "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. */
                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 [va_compact_ccs_rise](#) and [va_compact_ccs_fall Groups on page 570](#).

pin_based_variation Group

Example 123 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 about `peak_voltage` in `values` attribute see [timing_based_variation](#) and [pin_based_variation Groups on page 574](#).

Pin-based Model With nominal CCS receiver model and Variation-Aware CCS Receiver Model Groups

Example 124 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 is 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 is defined before all
       cell groups (or all nominal_va_values and va_values
       attributes) */
} /* end of library */
```

For information about `peak_voltage` in values attribute see [timing_based_variation Group on page 569](#).

nominal_va_values in Advanced CCS Modeling Usage

Example 125 nominal_va_values in Advanced CCS Modeling Usage

```
/* ASSUME that there is no voltage_map defined in library */

library (example) {
    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. The value 2.00 is wrong. */

/* There is voltage_map defined at library level. */
library (example) {
    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);
        /* Note: "voltage" is an user-defined parameter. */
    ...
}
```

Variational Values in Advanced CCS Modeling Usage

Example 126 Variational Values in Advanced CCS Modeling Usage

```
/* When var2 has a nominal value (1.0), var1 has three variational
values
(0.45, 0.55 and 0.50). However, only two values are allowed.
When var1 has a nominal value (0.50), var2 has two variational values
(0.8 and 1.0). The value 0.8 is less than the nominal value (1.0).
However, the value 1.0 is not greater than the nominal value,
```

```
and this is incorrect. /*  
...  
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);  
        ...  
    }  
}  
...  
}
```

Variation-Aware CCS Driver or Receiver with Timing Constraints

Example 127 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,\n            left_id, right_id");  
  
        base_curves_group: "ctbct1";  
    }  
    lu_table_template(LUT3) {  
        ...  
    }  
}
```

Chapter 11: Advanced Composite Current Source Modeling
Variation-Aware Timing Modeling Support

```
        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);
        }
    }
}
```

Chapter 11: Advanced Composite Current Source Modeling
Variation-Aware Timing Modeling Support

```
    ...
}

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);
        }
    }
}
```

Chapter 11: Advanced Composite Current Source Modeling
Variation-Aware Timing Modeling Support

```
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) {
            ...
        }
    }
}
```

Chapter 11: Advanced Composite Current Source Modeling
Variation-Aware Timing Modeling Support

```
        }
    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);
            ...
        }
        ...
    }
```

...

Chapter 11: Advanced Composite Current Source Modeling Variation-Aware Timing Modeling Support

```
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, 0.3329", \
            "-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 */
```

12

Nonlinear Signal Integrity Modeling

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](#)
-

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.

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](#)

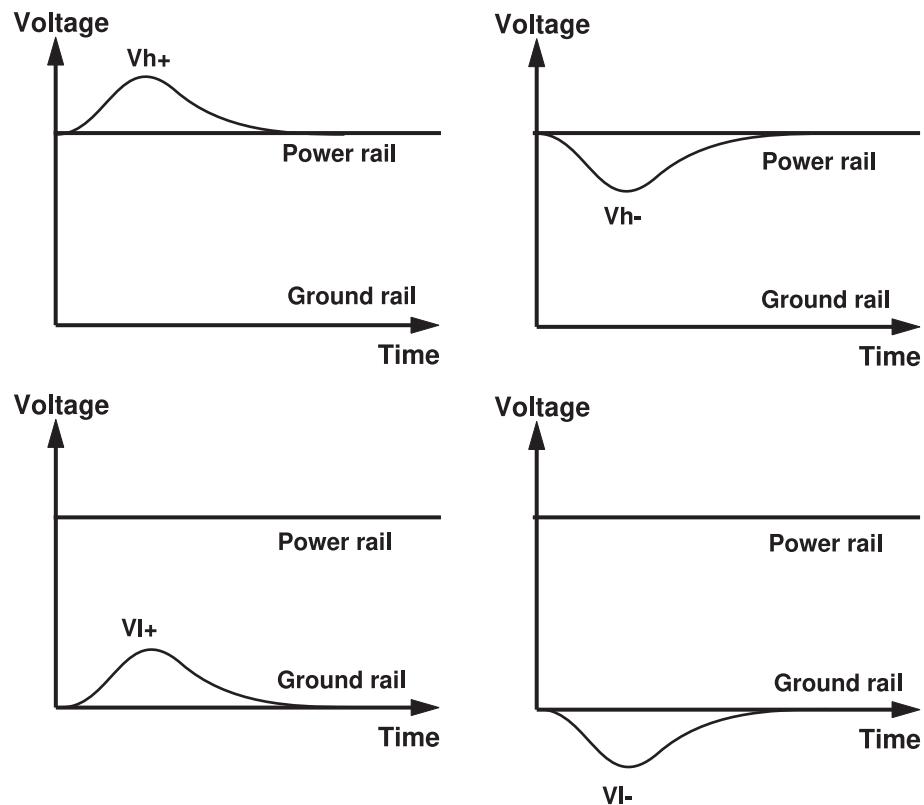
I-V Characteristics and Drive Resistance

To calculate the coupled noise glitch on a victim net, you need to know the effective steady-state drive resistance of the net. [Figure 105 on page 601](#) shows the four different types of noise glitch:

- V_{h+} : Input is high, and the noise is over the high voltage rail.
- V_{h-} : Input is high, and the noise is less than the high voltage rail.
- V_{l+} : Input is low, and the noise is over the low voltage rail.
- V_{l-} : Input is low, and the noise is less than the low voltage rail.

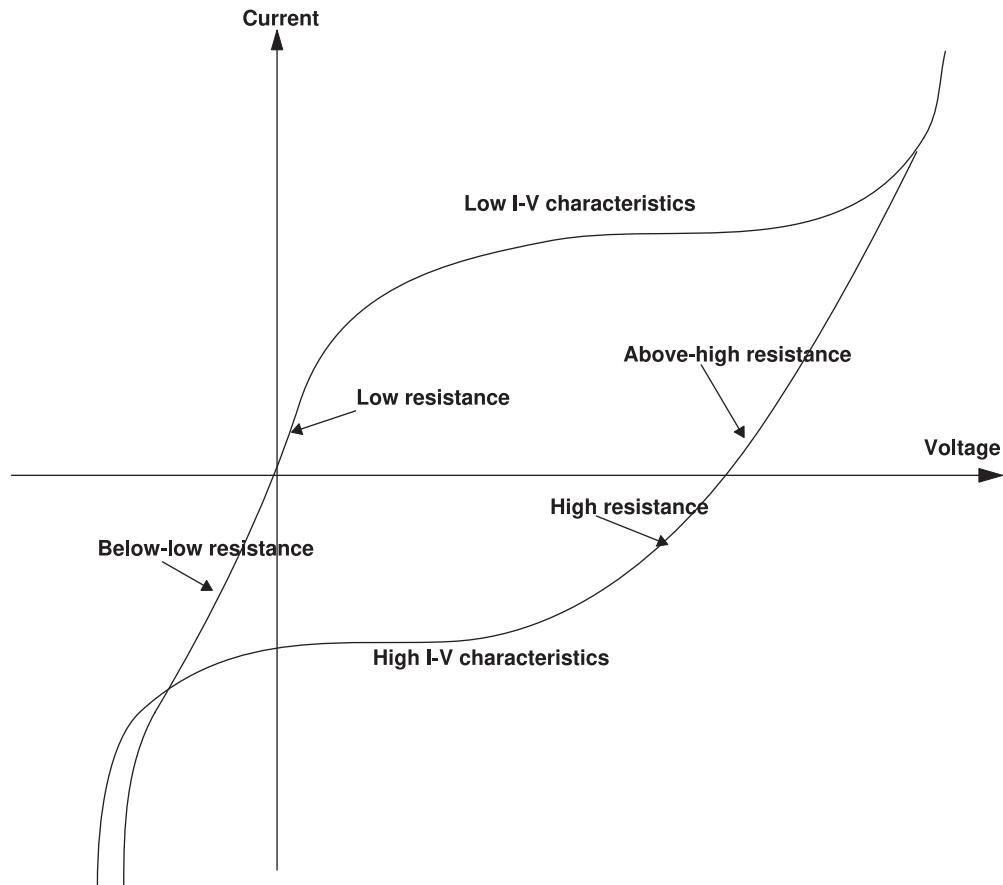
Because the current is a nonlinear function of the voltage, you need to characterize the steady-state I-V characteristics curve, which provides a more accurate view of the behavior of a cell in its steady state. This information is specified for every timing arc of the cell that can propagate a transition. 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 105 Noise Glitch and Steady-State Drive Resistance



[Figure 106 on page 602](#) is an example of two I-V curves and the steady-state resistance value.

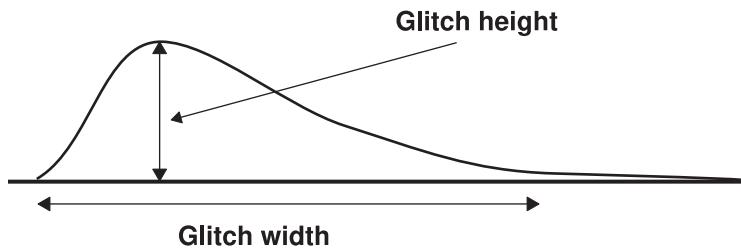
Figure 106 I-V Characteristics and Steady-State Drive Resistance



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 107](#) shows a glitch noise model.

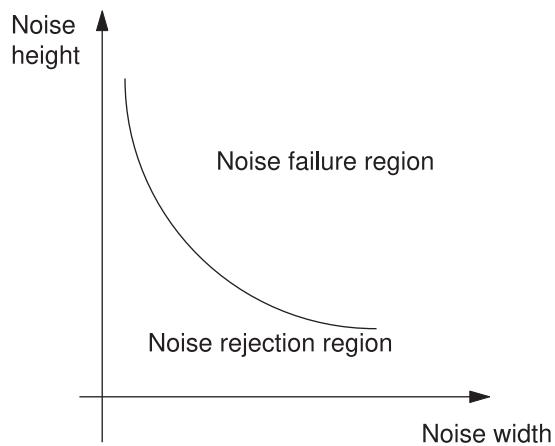
Figure 107 Glitch Noise Modeling



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 108 shows an example of a noise immunity curve.

Figure 108 Noise Immunity Curve

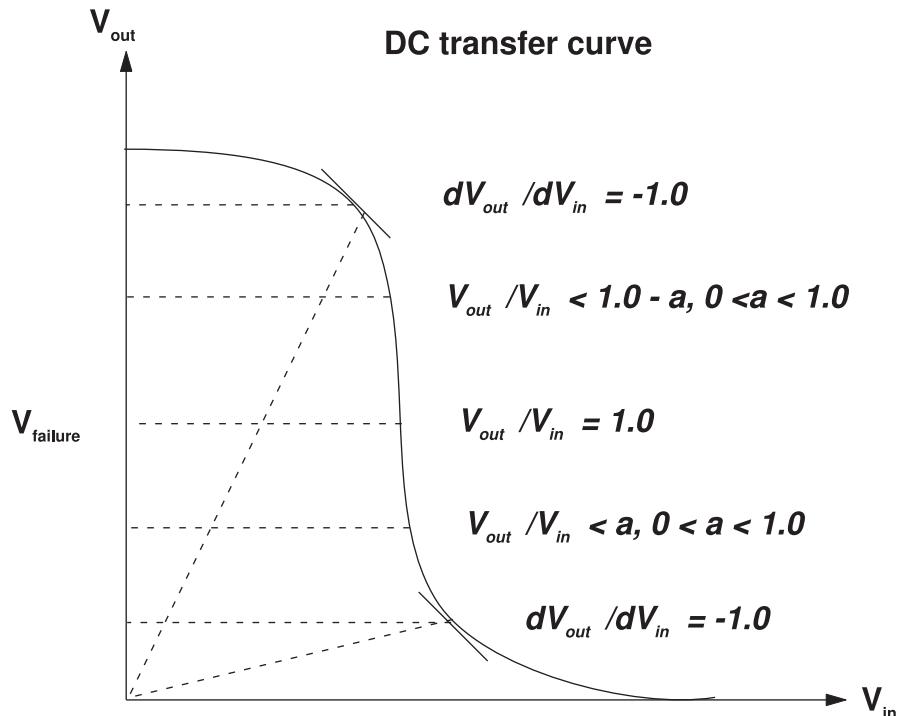


As shown in Figure 108, any noise width and height combination that falls above the noise rejection curve causes functional failure. The selection of cell failure voltage is important for noise immunity curve characterization.

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 109 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:

height = height_coefficient + area_coefficient / (width - width_coefficient);

Your tool 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.

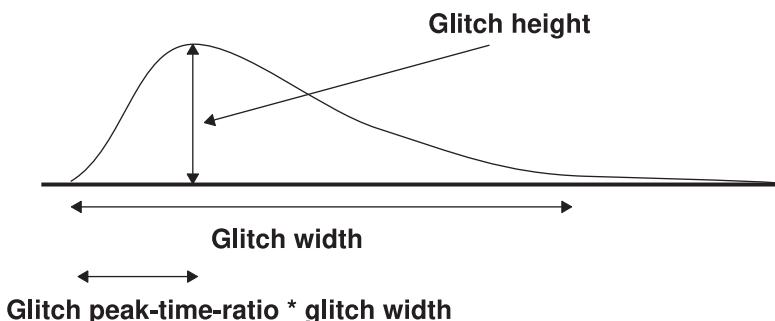
Noise Propagation

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 110](#) illustrates basic noise characteristics.

Figure 110 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 32](#) shows a summary of the syntax used to model cases when the cell is not switching.

Table 32 Summary of Library Requirements for Noise Model

Category		Model type	Description
Noise detection	Voltage ranges (DC noise margin)	Lookup table and polynomial	<code>input_voltage</code> or <code>output_voltage</code> 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
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

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, steady_state_current_high, or steady_state_current_tristate group.

Syntax

```
library(namestring) {  
    ...  
    iv_lut_template(template_namestring) {  
        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_namestring) {
        values("float,..., float");
    }
    steady_state_current_low(template_namestring) {
        values("float,..., float");
    }
    ...
}

timing() { /* for three-state arcs */
    steady_state_current_tristate(template_namestring) {
        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`, `voltagei`, 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.

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_namestring) {
        orders("integer,..., integer");
        coefs("float,..., float");
        domain(domain_namestring) {
            orders("integer,..., integer");
            coefs("float,..., float");
        }
        ...
    }
    steady_state_current_low(template_namestring)
    ...
}

timing() { /* for three-state arcs */
    steady_state_current_tristate(template_namestring) {
        ...
    }
    ...
}
```

The orders, coefs, and variable_range attributes represent the polynomial for the current for high, low, and three-state.

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, the model becomes 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. 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, an error is generated. Similarly if the output pin is tied to high (`function : "1"`) and its timing arc specifies the `steady_state_current_low` group, an error is generated.
- If noise immunity and noise propagation are specified in the timing arcs of a `tied_off` pin, an error is generated.

- If the `related_pin` attribute is specified on a `tied_off` output pin, an error is generated.

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 Liberty 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_namestring) {
            ...
        }
        noise_immunity_high(template_namestring) {
            ...
        }
        noise_immunity_low(template_namestring) {
            ...
        }
    }
```

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 and width combination that causes functional failure.
- The unit for the height is the library voltage unit.
- For points outside table ranges, extrapolation can be 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`, `voltage i` , and `temperature`.

Syntax

```
library(namestring) {
    ...
    poly_template(template_namestring) {
        variables(variable_ienum, ..., variable_nenum);
        variable_i_range: (float, float);
        ...
        variable_n_range: (float, float);
        mapping(voltageenum, power_railid);
        domain(domain_namestring);
        {
            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.

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_namestring) {
        orders("integer,..., integer");
        coefs("float,..., float");
        ...
        domain(domain_namestring) {
            orders("integer,..., integer");
            coefs("float,...,float");
        }
        ...
    }
    noise_immunity_below_low(template_namestring) {
        ...
    }
    noise_immunity_high(template_namestring) {
        ...
    }
    noise_immunity_low(template_namestring) {
        ...
    }
    ...
}
```

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, 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

To specify whether a noise immunity or propagation table is referenced within the noise range indexes, the Liberty 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.
- The `min_input_noise_width` and `max_input_noise_width` attributes must both be defined.
- `min_input_noise_width <= max_input_noise_width`

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

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 `propagation_lut_template` group in the `library` group. In the `timing` group, use the following groups:

- `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`

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(namestring) {  
    ...  
    propagation_lut_template(template_namesstring) {  
        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_namestring) {  
        index_1 ("float,..., float");  
        index_2 ("float,..., float");  
        index_3 ("float,..., float");  
        values("float,..., float,..."float,..., float");  
    }  
    propagated_noise_height_below_low (temp_namestring) {  
        ...  
    }  
    propagated_noise_width_above_high (temp_namestring) {  
        ...  
    }  
    propagated_noise_width_below_low (temp_namestring) {  
        ...  
    }  
    propagated_noise_height_high(template_namestring) {  
        ...  
    }  
    propagated_noise_height_low(template_namestring) {  
        ...  
    }  
    propagated_noise_width_high(template_namestring) {  
        ...  
    }  
    propagated_noise_width_low(template_namestring) {  
        ...  
    }  
    ...  
}
```

Propagation Noise Group Rules

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 can be 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 `poly_template` group in the library group.

In the timing group, use the following groups:

- `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
- propagated_noise_peak_time_ratio_low

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 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_namestring) {
        variable_i_range: (float, float);
        orders("integer,..., integer");
        coefs("float,..., float");
        domain(domain_namestring) {
            variable_i_range: (float, float);
            orders("integer,..., integer");
            coefs("float,..., float");
        }
        ...
    }
    propagated_noise_width_above_high (temp_namestring) {
        ...
    }
    propagated_noise_height_below_low (temp_namestring) {
        ...
    }
}
```

```
}

propagated_noise_width_below_low (temp_namestring) {
    ...
}

propagated_noise_height_high (temp_namestring) {
    ...
}

propagated_noise_width_high (temp_namestring) {
    ...
}

propagated_noise_height_low (temp_namestring) {
    ...
}

propagated_noise_width_low (temp_namestring) {
    ...
}

propagated_noise_peak_time_ratio_above_high (temp_namestring) {
    ...
}

propagated_noise_peak_time_ratio_below_low (temp_namestring) {
    ...
}

propagated_noise_peak_time_ratio_high (temp_namestring) {
    ...
}

propagated_noise_peak_time_ratio_low (temp_namestring) {
    ...
}

...
```

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 ");
}

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 ");
```

}

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 Example

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(VD $\bar{D}$ 1, 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);
```

Chapter 12: Nonlinear Signal Integrity Modeling
Examples of Modeling Noise

```
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 ) */
```

Chapter 12: Nonlinear Signal Integrity Modeling

Examples of Modeling Noise

```
/* 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 */7
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;
        }
    }
}
```

Chapter 12: Nonlinear Signal Integrity Modeling
Examples of Modeling Noise

```
        }
    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, \

```

Chapter 12: Nonlinear Signal Integrity Modeling
Examples of Modeling Noise

```
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 ("I, 2, I, 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 ("I, 2, I, 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 ("I, 2, I, 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, ");
```

Chapter 12: Nonlinear Signal Integrity Modeling
Examples of Modeling Noise

```
}

propagated_noise_width_low(my_propagated_noise) {
    orders ("I, 2, I, 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_low(my_propagated_noise) {
    orders ("I, 2, I, 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_low(my_propagated_noise) {
    orders ("I, 2, I, 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_above_high(my_propagated_noise) {
    orders ("I, 2, I, 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_above_high(my_propagated_noise) {
    orders ("I, 2, I, 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 ("I, 2, I, 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 ("I, 2, I, 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 ("I, 2, I, 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 ("I, 2, I, 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) {
```

Chapter 12: Nonlinear Signal Integrity Modeling
Examples of Modeling 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_*
```

Chapter 12: Nonlinear Signal Integrity Modeling

Examples of Modeling Noise

```
* 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.
```

Chapter 12: Nonlinear Signal Integrity Modeling

Examples of Modeling Noise

```
* 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 */
```

Chapter 12: Nonlinear Signal Integrity Modeling
Examples of Modeling Noise

```
    } /* end of pin (Y)( */  
} /* end of cell (INV) */  
} /* end of library (my_noise_lib)
```

13

Composite Current Source Signal Integrity Modeling

This chapter provides an overview of composite current source (CCS) modeling to support noise (signal integrity) modeling for advanced technologies.

This chapter includes the following sections:

- [CCS Signal Integrity Modeling Overview](#)
 - [CCS Noise Modeling for Unbuffered Cells With a Pass Gate](#)
 - [CCS Noise Modeling for Multivoltage Designs](#)
 - [Referenced CCS Noise Modeling](#)
-

CCS Signal Integrity Modeling Overview

CCS noise modeling can capture essential noise properties of digital circuits using a compact library representation. It enables fast and accurate gate-level noise analysis while maintaining a relatively simple library characterization. CCS noise modeling supports noise combination and driver weakening.

CCS noise is characterization data that provides information for noise failure detection on cell inputs, calculation of noise bumps on cell outputs, and noise propagation through the cell. For the best accuracy, you must add CCS timing data to the library in addition to the CCS noise data. The CCS noise data includes the following:

- Channel-connected block parameters
 - DC current tables
 - Timing tables for rising and falling transitions
 - Timing tables for low and high propagated noise
-

CCS Signal Integrity Modeling Syntax

```
library (name) {  
    ...
```

Chapter 13: Composite Current Source Signal Integrity Modeling
CCS Signal Integrity Modeling Overview

```
lu_table_template(dc_template_name) {
    variable_1 : input_voltage;
    variable_2 : output_voltage;
}
lu_table_template(output_voltage_template_name) {
    variable_1 : input_net_transition;
    variable_2 : total_output_net_capacitance;
    variable_3 : time;
}
lu_table_template(propagated_noise_template_name) {
    variable_1 : input_noise_height;
    variable_2 : input_noise_width;
    variable_3 : total_output_net_capacitance;
    variable_4 : time;
}
cell (name) {
    pin (name) {
        ...
        ccsn_first_stage () {
            is_needed : true | false;
            is_inverting : Boolean;
            stage_type : stage_type_value;
            miller_cap_rise : float;
            miller_cap_fall : float;
            output_signal_level : power_supply_name;
            related_ccb_node : spice_node_name;
            dc_current (dc_current_template) {
                index_1("float, ...");
                index_2("float, ...");
                values("float, ...");
            }
            output_voltage_rise ( )
                vector (output_voltage_template_name) {
                    index_1(float);
                    index_2(float);
                    index_3("float, ...");
                    values("float, ...");
                }
            ...
        }
        output_voltage_fall ( )
            vector (output_voltage_template_name) {
                index_1(float);
                index_2(float);
                index_3("float, ...");
                values("float, ...");
            }
        ...
    }
    propagated_noise_low ( )
        vector (propagated_noise_template_name) {
            index_1(float);
```

```
    index_2(float);
    index_3(float);
    index_4("float, ...");
    values("float, ...");
}
...
}
propagated_noise_high () {
    vector (propagated_noise_template_name) {
        index_1(float);
        index_2(float);
        index_3(float);
        index_4("float, ...");
        values("float, ...");
    }
}
when : "Boolean expression";
} /* ccsn_first_stage */
ccsn_last_stage () {
    is_needed : true | false;
    is_inverting : Boolean;
    stage_type : stage_type_value;
    miller_cap_rise : float;
    miller_cap_fall : float;
    input_signal_level : power_supply_name;
    related_ccb_node : spice_node_name;
    dc_current (dc_current_template) {
        index_1("float, ...");
        index_2("float, ...");
        values("float, ...");
    }
    output_voltage_rise () {
        vector (output_voltage_template_name) {
            index_1(float);
            index_2(float);
            index_3("float, ...");
            values("float, ...");
        }
    }
}
output_voltage_fall () {
    vector (output_voltage_template_name) {
        index_1(float);
        index_2(float);
        index_3("float, ...");
        values("float, ...");
    }
}
...
}
propagated_noise_low () {
    vector (propagated_noise_template_name) {
        index_1(float);
    }
}
```

Chapter 13: Composite Current Source Signal Integrity Modeling
CCS Signal Integrity Modeling Overview

```
    index_2(float);
    index_3(float);
    index_4("float, ...");
    values("float, ...");
}
...
}
propagated_noise_high () {
    vector (propagated_noise_template_name) {
        index_1(float);
        index_2(float);
        index_3(float);
        index_4("float, ...");
        values("float, ...");
    }
}
...
}
when : "Boolean expression";
} /* ccsn_last_stage */
...
timing() {
    ...
ccsn_first_stage () {
    is_needed : true | false;
    is_inverting : Boolean;
    stage_type : stage_type_value;
    miller_cap_rise : float;
    miller_cap_fall : float;
    output_signal_level : power_supply_name;
    related_ccb_node : spice_node_name;
    dc_current(dc_current_template) {
        index_1("float, ...");
        index_2("float, ...");
        values("float, ...");
    }

    output_voltage_rise () {
        vector (output_voltage_template_name) {
            index_1(float);
            index_2(float);
            index_3("float, ...");
            values("float, ...");
        }
    }
}
output_voltage_fall () {
    vector (output_voltage_template_name) {
        index_1(float);
        index_2(float);
        index_3("float, ...");
        values("float, ...");
    }
}
...
}
```

```
        }
        propagated_noise_low () {
            vector (propagated_noise_template_name) {
                index_1(float);
                index_2(float);
                index_3(float);
                index_4("float, ...");
                values("float, ...");
            }
            ...
        }
        propagated_noise_high () {
            vector (propagated_noise_template_name) {
                index_1(float);
                index_2(float);
                index_3(float);
                index_4("float, ...");
                values("float, ...");
            }
            ...
        }
        when : "Boolean expression";
    } /* ccsn_first_stage */
ccsn_last_stage () {
    is_needed : true | false;
    is_inverting : Boolean;
    stage_type : stage_type_value;
    miller_cap_rise : float;
    miller_cap_fall : float;
    input_signal_level : power_supply_name;
    related_ccb_node : spice_node_name;
    dc_current (dc_current_template)
        index_1("float, ...");
        index_2("float, ...");
        values("float, ...");
}

output_voltage_rise ()
vector (output_voltage_template_name) {
    index_1(float);
    index_2(float);
    index_3("float, ...");
    values("float, ...");
}
...
}
output_voltage_fall () {
    vector (output_voltage_template_name) {
        index_1(float);
        index_2(float);
        index_3("float, ...");
        values("float, ...");
}
```

```
        }
        ...
    }
    propagated_noise_low () {
        vector (propagated_noise_template_name) {
            index_1(float);
            index_2(float);
            index_3(float);
            index_4("float, ...");
            values("float, ...");
        }
        ...
    }
    propagated_noise_high () {
        vector (propagated_noise_template_name) {
            index_1(float);
            index_2(float);
            index_3(float);
            index_4("float, ...");
            values("float, ...");
        }
        ...
    }
    when : "Boolean expression";
} /* ccsn_last_stage */
} /* end timing/pin/cell/library group */
```

Library-Level Groups and Attributes

This section describes the library-level groups and attributes used for CCS noise modeling.

lu_table_template Group

The `lu_table_template` group creates the lookup-table template for the `dc_current` group and vectors for the `output_voltage_rise`, `output_voltage_fall`, `propagated_noise_high`, and `propagated_noise_low` groups.

variable_1, variable_2, variable_3, and variable_4 Attributes

Set the `variable_1`, `variable_2`, `variable_3`, and `variable_4` attributes inside the `lu_table_template` group.

You can specify the template used for the following tables and vectors by using a combination of these attributes:

- The `output_current_rise` and `output_current_fall` group vectors
Valid values for `variable_1`, `variable_2`, and `variable_3` are `input_net_transition`, `total_output_net_capacitance`, and `time`, respectively.
- The `propagated_noise_low` and `propagated_noise_high` group vectors
Valid values for `variable_1`, `variable_2`, `variable_3`, and `variable_4` are `input_noise_height`, `input_noise_width`, `total_output_net_capacitance`, and `time`, respectively.
- The template used for the `dc_current` tables
Valid values for `variable_1` and `variable_2` are `input_voltage` and `output_voltage`, respectively.

Pin-Level Groups and Attributes

This section describes the pin-level groups and attributes used for CCS noise modeling.

`ccsn_first_stage` and `ccsn_last_stage` Groups

The `ccsn_first_stage` and `ccsn_last_stage` groups specify CCS noise data for the first stage or the last stage of channel-connected blocks. The `ccsn_first_stage` and `ccsn_last_stage` groups can be defined inside timing or pin groups.

The `ccsn_first_stage` and `ccsn_last_stage` groups contain the following:

- The `is_needed`, `is_inverting`, `stage_type`, `miller_cap_rise` and `miller_cap_fall`, `output_signal_level` or `input_signal_level`, and the optional `related_ccb_node` channel-connected block attributes
- The `dc_current` group, which contains a two-dimensional DC current table
- The `output_current_rise` and `output_current_fall` groups, which contain two timing tables for rising and falling transitions.
- The `propagated_noise_low` and `propagated_noise_high` groups, which contain two noise tables for low and high propagated noise.

Note:

At the pin level, the `ccsn_last_stage` group can be defined only in an output pin or inout pin.

is_needed Attribute

The `is_needed` Boolean attribute determines whether the `dc_current`, `output_current_rise`, `output_current_fall`, `propagated_noise_low`, and `propagated_noise_high` channel-connected block attributes should be specified to include CCS noise data for a cell. The `is_needed` attribute is defined inside the `ccsn_first_stage` and `ccsn_last_stage` groups.

By default, the `is_needed` attribute is set to `true`, which means that CCS noise data is included in the `ccsn_first_stage` and `ccsn_last_stage` groups for the cell. The `is_needed` attribute should be set to `false` for cells that do not need a current-based driver model, such as diodes, antennas, and cload cells. When the attribute is set to `false`, CCS noise data, enabled by the channel-connected block attributes, is not included in the `ccsn_first_stage` and `ccsn_last_stage` groups.

is_inverting Attribute

The `is_inverting` attribute specifies whether the channel-connected block is inverting. If the channel-connected block is inverting, set the `is_inverting` attribute to `true`. Otherwise, set the attribute to `false`. This attribute is mandatory if the `is_needed` attribute is set to `true`. Note that the `is_inverting` attribute is different from the “invertness” or `timing_sense` of the timing arc, which might consist of multiple channel-connected blocks.

stage_type Attribute

The `stage_type` attribute specifies the channel-connected block’s output voltage stage type. The valid values are `pull_up`, which causes the channel-connected block’s output voltage to be pulled up or to rise; `pull_down`, which causes the channel-connected block’s output voltage to be pulled down or to fall; and `both`, which causes the channel-connected block’s output voltage to be pulled up or down.

miller_cap_rise and miller_cap_fall Attributes

The `miller_cap_rise` and `miller_cap_fall` float attributes specify the Miller capacitance value for a rising and falling channel-connected block output transition. The value must be greater than or equal to zero. The attributes are defined inside the `ccsn_first_stage` and `ccsn_last_stage` groups.

output_signal_level and input_signal_level Attributes

The `output_signal_level` and `input_signal_level` attributes specify the power supply voltage names for a channel-connected block output and input, respectively. The `output_signal_level` attribute is defined inside the `ccsn_first_stage` group and the `input_signal_level` attribute is defined inside the `ccsn_last_stage` group.

Note:

The `output_signal_level` and `input_signal_level` attribute specifications within the `ccsn_first_stage` and `ccsn_last_stage` groups override the `output_signal_level` and `input_signal_level` attribute specifications at the pin level.

For a timing arc, the `output_signal_level` attribute specification within the `ccsn_first_stage` group overrides the `output_signal_level` attribute specification for the related pin (defined by the `related_pin` attribute).

related_ccb_node Attribute

The optional `related_ccb_node` attribute specifies the SPICE node in the subcircuit netlist that is used for the `dc_current` table characterization and waveform measurements.

dc_current Group

The `dc_current` group specifies the input and output voltage values of a two-dimensional current table for a channel-connected block. Use the `index_1` and `index_2` attributes, respectively, to list the input and output voltage values in library voltage units. Specify the `values` attribute in the `dc_current` group to list the relative channel-connected block DC current values, in library current units, that are measured at the channel-connected block output node.

output_voltage_rise and output_voltage_fall Groups

The `output_voltage_rise` and `output_voltage_fall` groups specify `vector` groups that describe three-dimensional `output_voltage` tables for a channel-connected block whose output node's voltage values are rising or falling. The groups are defined inside the `ccsn_first_stage` and `ccsn_last_stage` groups.

Specify the following attributes in the `vector` group: The `index_1` attribute lists the `input_net_transition (slew)` values in library time units. The `index_2` attribute lists the `total_output_net_capacitance (load)` values in library capacitance units. The `index_3` attribute lists the sampling time values in library time units. The `values` attribute lists the voltage values, in library voltage units, that are measured at the channel-connected block output node.

propagated_noise_low and propagated_noise_high Groups

The `propagated_noise_low` and `propagated_noise_high` groups use `vector` groups to specify the three-dimensional `output_voltage` tables of the channel-connected block whose output node's voltage values are rising or falling. The groups are defined inside the `ccsn_first_stage` and `ccsn_last_stage` groups.

Specify the following attributes in the `vector` group: The `index_1` attribute lists the `input_noise_height` values in library voltage units. The `index_2` attribute lists the `input_noise_width` values in library time units. The `index_3` attribute lists the `total_output_net_capacitance` values in library capacitance units. The `index_4` attribute lists the sampling time values in library time units. The `values` attribute lists the voltage values, in library voltage units, that are measured at the channel-connected block output node.

when Attribute

The `when` attribute specifies the condition under which the channel-connected block data is applied. The attribute is defined in the `ccsn_first_stage` and `ccsn_last_stage` groups both at the pin level and the timing level.

CCS Noise Library Examples

Example 128 CCS Noise Library Example

```
library (CCS_noise) {

    technology ( cmos ) ;
    delay_model      : table_lookup;
    time_unit        : "1ps" ;
    leakage_power_unit : "1pW" ;
    voltage_unit     : "1V" ;
    current_unit     : "1uA" ;
    pulling_resistance_unit : "1kohm" ;
    capacitive_load_unit(1000.000,ff) ;

    nom_voltage      : 1.200;
    nom_temperature  : 25.000;
    nom_process       : 1.000;

    operating_conditions("OC1") {
        process : 1.000;
        temperature : 25.000;
        voltage : 1.200;
        tree_type : "balanced_tree";
    }
    default_operating_conditions:OC1;

    lu_table_template(del_0_5_7_t) {
        variable_1 : input_net_transition;
        index_1("10.000, 175.000, 455.000, 980.000, 2100.000");
        variable_2 : total_output_net_capacitance;
        index_2("0.000000, -0.004000, 0.007000, 0.019000, 0.040000, 0.075000,\n         0.175000");
    }

    lu_table_template(ccsn_dc_29x29) {
        variable_1 : input_voltage;
        variable_2 : output_voltage;
    }

    lu_table_template(ccsn_timing_lut_5) {
```

Chapter 13: Composite Current Source Signal Integrity Modeling

CCS Signal Integrity Modeling Overview

```
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 () {

```

Chapter 13: Composite Current Source Signal Integrity Modeling CCS Signal Integrity Modeling Overview

```
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 *
```

The following Liberty snippet shows the use of the optional `related_ccb_node` attribute in pin-based `ccsn_first_stage` and `ccsn_last_stage` groups.

Example 129 Using related_ccb_node Attribute in Pin-Based CCS Noise Models

```
cell("A") {
  ...
  pin(CK) {
    direction : "input" ;
    ...
    ccsn_first_stage() {
      related_ccb_node : "ckn:711" ;
      ...
    }
  }
  pin(EN) {
    direction : "input" ;
    ...
    ccsn_first_stage() {
      ...
    }
  }
  pin(Q) {
    direction : "output" ;
    ...
    ccsn_last_stage() {
      related_ccb_node : "i3:1623" ;
      ...
    }
  }
  ...
}
```

In the following Liberty snippet of a 2-input AND gate, the `related_ccb_node` attribute of the `ccsn_first_stage` and `ccsn_last_stage` groups are specified for the A→Y timing arc.

Example 130 Using related_ccb_node Attribute in Timing Arcs

```
cell (AND2) {
  ...
  pin (A) {
    ...
  }
  pin (B) {
    ...
  }
  pin (Y) {
    ...
    timing () {
      ...
      related_pin :A;
      ccsn_first_stage() {
        related_ccb_node : "net1:15";
        ...
      }
      ccsn_last_stage() {
```

```
    related_ccb_node : "net1:15";
    ...
}
}
...
}
```

Conditional Data Modeling in CCS Noise Models

The following attributes support conditional data modeling in pin-based CCS noise models:

- The `when` attribute in the `ccsn_first_stage` and `ccsn_last_stage` groups.
- The `mode` attribute.

The following syntax shows the `when` and `mode` support for pin-based CCS noise models:

```
cell(cell_name) {
    mode_definition (mode_name) {
        mode_value(namestring) {
            when : "Boolean expression";
            sdf_cond : "Boolean expression";
        } ...
    } ...
    pin(pin_name) {
        direction : input;
        /* The following syntax supports pin-based ccs noise */
        /* ccs noise first stage for Condition 1 */
        ccsn_first_stage() {
            is_needed :true | false;
            when : "Boolean expression";
            mode (mode_name, mode_value);
            ...
        }
        ...
        /* ccs noise first stage for Condition n */
        ccsn_first_stage() {
            is_needed :true | false;
            when : "Boolean expression";
            mode (mode_name, mode_value);
            ...
        }
        pin(pin_name) {
            direction : output;
            /* ccs noise last stage for Condition 1 */
            ccsn_last_stage() {
                is_needed : true | false;
                when : "Boolean expression";
                mode (mode_name, mode_value);
                ...
            }
            ...
        }
    }
}
```

```
/* ccs noise last stage for Condition n */
ccsn_last_stage() {
    is_needed : true | false;
    when : "Boolean expression";
    mode (mode_name, mode_value);
    ...
}
timing() {
    ...
/* following are arc-based ccs noise */
ccsn_first_stage() {
    is_needed : true | false;
    ...
}
...
ccsn_last_stage() {
    is_needed : true | false;
    ...
}
}
```

when Attribute

The `when` attribute is a conditional attribute that is supported in pin-based CCS noise models in the `ccsn_first_stage` and `ccsn_last_stage` groups.

mode Attribute

The pin-based `mode` attribute is provided in the `ccsn_first_stage` and `ccsn_last_stage` groups for conditional data modeling. If the `mode` attribute is specified, `mode_name` and `mode_value` must be predefined in the `mode_definition` group at the cell level.

Example

```
library (csm13os120_typ) {
    technology ( cmos );
    delay_model : table_lookup;
    lu_table_template(ccsn_dc_29x29) {
        variable_1 : input_voltage;
        variable_2 : output_voltage;
    }
    cell(inv0d0) {
        area : 0.75;
        mode_definition(rw) {
            mode_value(read) {
                when : "I";
                sdf_cond : "I == 1";
            }
            mode_value(write) {
                when : "!I";
            }
        }
    }
}
```

```
        sdf_cond : "I == 0";
    }
}
pin(I) {
    direction : input;
    max_transition : 2100.0;
    capacitance : 0.002000;
    fanout_load : 1;
    ...
}
pin(ZN) {
    direction : output;
    max_capacitance : 0.175000;
    max_fanout : 58;
    max_transition : 1400.0;
    function : "(I)";
/* pin-based CCS noise first stage for Condition 1 */
ccsn_first_stage () {
    ...
    when : "I"; /* or using mode as next commented line */
/* mode(rw, read); */
    dc_current(ccsn_dc_29x29) { ... }
    output_voltage_rise() { ... }
    output_voltage_fall() { ... }
    propagated_noise_low() { ... }
    propagated_noise_high() { ... }
} /* ccsn_last_stage */
/* pin-based CCS noise last stage for Condition 2 */
ccsn_last_stage () {
    ...
    when : "!I"; /* or using mode as next commented line */
/* mode(rw, read); */
    dc_current(ccsn_dc_29x29) { ... }
    output_voltage_rise() { ... }
    output_voltage_fall() { ... }
    propagated_noise_low() { ... }
    propagated_noise_high() { ... }
} /* ccsn_last_stage */

timing() {
    related_pin : "I";
    timing_sense : negative_unate;
    ...
} /* timing I -> Z */
} /* Z */
} /* cell(inv0d0) */
} /* library */
```

CCS Noise Modeling for Unbuffered Cells With a Pass Gate

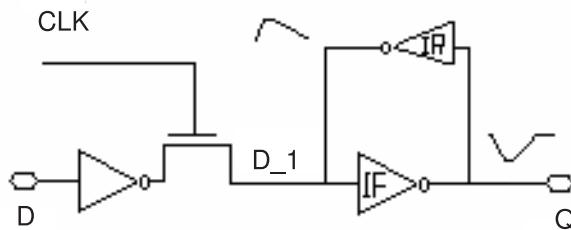
Unbuffered input and output latches are a special type of cell that has an internal memory node connected to an input or output pin. To increase the speed of the design and lower power consumption, these cells do not use inverters.

The major difference between an unbuffered output cell and an unbuffered input cell is as follows:

- Unbuffered Output Cell

An unbuffered output cell has the *feedback*, or back-driving path, from the unbuffered output pin to an internal node. In [Figure 111](#), Q is connected to internal node D_1 through the IR inverter.

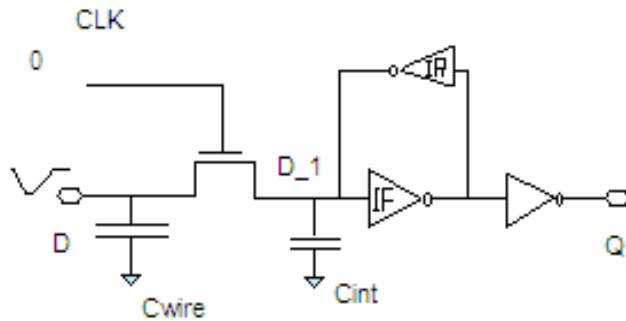
Figure 111 Unbuffered Output Latch



- Unbuffered Input Cell

The input pin of an unbuffered cell is not buffered and can be connected through a pass gate to the internal node. (A pass gate is a special gate that has an input and an output and a control input. If the control is set to true, the output is driven by the input. Otherwise, it is a floating output.) For example, in [Figure 112](#), D is connected to internal node D_1 through a pass gate.

Figure 112 Unbuffered Input Latch



To correctly model this category of cells, you must determine:

- If a pin is buffered or unbuffered.
- If a pin is implemented with a pass gate.
- If the `ccsn_*_stage` information models a pass gate.

Syntax for Unbuffered Output Latches

```
/* unbuffered output pin */
pin (pin_name) {
    direction : inout | output;
    is_unbuffered : true | false ;
    has_pass_gate : true | false ;
    ccsn_first_stage () {
        is_pass_gate : true | false;
        ...
    }
    ...
    ccsn_last_stage () {
        is_pass_gate : true | false;
        ...
    }
    ...
    timing() {
        ccsn_first_stage () {
            is_pass_gate : true | false;
            ...
        }
        ...
        ccsn_last_stage () {
            is_pass_gate : true | false;
            ...
        }
        ...
    }
}
```

```
        }
        pin (pin_name) {
            direction : input | inout;
            is_unbuffered : true | false ;
            has_pass_gate : true | false ;
            ccsn_first_stage () {
                is_pass_gate : true | false;
                ...
            }
            ...
            ccsn_last_stage () {
                is_pass_gate : true | false;
                ...
            }
            ...
            timing() {
                ccsn_first_stage () {
                    is_pass_gate : true | false;
                    ...
                }
                ...
                ccsn_last_stage () {
                    is_pass_gate : true | false;
                    ...
                }
                ...
            }
        }
    }
```

Pin-Level Attributes

The following attributes are pin-level attributes for unbuffered output latches.

is_unbuffered Attribute

The `is_unbuffered` simple Boolean attribute indicates that a pin is unbuffered. This optional attribute can be specified on the pins of any library cell. The default is `false`.

has_pass_gate Attribute

The `has_pass_gate` simple Boolean attribute can be defined in a pin group to indicate whether the pin is internally connected to at least one pass gate.

ccsn_first_stage Group

The `ccsn_first_stage` group specifies CCS noise for the first stage of the channel-connected block (CCB). When the `ccsn_first_stage` group is defined at the pin level, it can only be defined in an input pin or an inout pin.

The `ccsn_first_stage` group syntax models back-driving CCS noise propagation information from the output pin to the internal node.

is_pass_gate Attribute

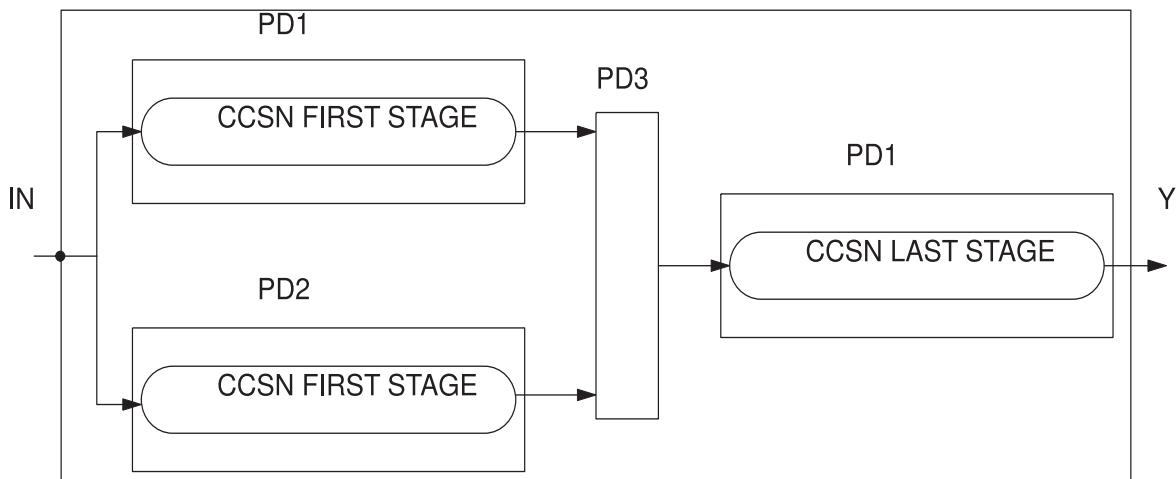
The `is_pass_gate` Boolean attribute is defined in a `ccsn_*_stage` group (such as `ccsn_first_stage`) to indicate whether the `ccsn_*_stage` information is modeled for a pass gate. The attribute is optional and its default is `false`.

CCS Noise Modeling for Multivoltage Designs

In multivoltage designs, a complex macro cell can have multiple power domains with different power supplies internal to the cell. The internal power supplies that provide power to some of the inputs and outputs of the channel-connected blocks (CCBs), cannot be modeled at the pin level. Therefore, pin-level attributes do not correctly describe the operation of the CCS noise stages for these channel-connected blocks. To correctly model the CCS noise stages, specify the internal power supplies in the `ccsn_first_stage` and `ccsn_last_stage` groups that are specified within the `pin` or `timing` groups.

Figure 113 shows a macro cell with three power domains, PD1, PD2, and PD3. The input pin, IN, is connected to the channel-connected blocks (not shown in the figure) for two CCS noise stages in PD1 and PD2. The outputs of the channel-connected blocks go to PD3. The CCS noise stage for the output pin, Y, is in PD1 and is driven by an internal stage in PD3. The example in Figure 113 shows that the channel-connected block inputs or outputs for the CCS stages do not always map to a pin.

Figure 113 A Complex Macro Cell With Multiple Power Domains



To model such CCS noise stages, use the `output_signal_level` and the `input_signal_level` attributes respectively within the `ccsn_first_stage` and `ccsn_last_stage` groups, as shown in [Example 131](#).

Example 131 Pin and Timing Based CCS Noise Model of Low-to-High Level-Shifter Cells

```
library (test) {
    technology(cmos);
    nom_voltage : 1.080000;
    voltage_unit : "1V";
    voltage_map(VDD1,1.080000);
    voltage_map(GND1,0.000000);
    voltage_map(VDD2,0.900000);
    ...
cell (LVL_SHIFT_L2H_1) {
    is_level_shifter : true;
    level_shifter_type : LH;
    input_voltage_range(0.900000,1.320000);
    output_voltage_range(0.900000,1.320000);
    pg_pin (VDD) {
        voltage_name : VDD1;
        ...
    }
    pg_in (VDDL) {
        voltage_name : VDD2;
        ...
    }
    pin (I) {
        direction : input;
        level_shifter_data_pin : true;
        related_ground_pin : VSS;
        related_power_pin : VDDL;
        ccsn_first_stage () /* pin-based model */
            is_needed : true;
            stage_type : both;
            ...
            output_signal_level : VDD1;
            /* Specifies that the power supply to the ccsn_first_stage output
               is VDD1. The power supply to the ccsn_first_stage input is
               specified at the pin level by the related_power_pin attribute
               as VDDL. */
            dc_current (ccsn_dc)...
    }
    pin (Z) {
        direction : output;
        power_down_function : "!VDD + !VDDL + VSS";
        function : "I";
        related_ground_pin : VSS;
        related_power_pin : VDD;
        ...
        ccsn_last_stage () /* pin-based model */
    }
}
```

```
    is_needed : "true";
    stage_type : "both";
    ...
    input_signal_level : VDD2;
    /* Specifies that the power supply to the ccsn_last_stage input
       is VDD2. The power supply to the ccsn_last_stage output is
       specified at the pin level by the related_power_pin attribute
       as VDD. */
    dc_current (ccsn_dc)...
}

...
}

cell (LVL_SHIFT_L2H_2) {
    is_level_shifter : true;
    level_shifter_type : LH;
    input_voltage_range(0.900000,1.320000);
    output_voltage_range(0.900000,1.320000);
    pg_pin (VDD) {
        voltage_name : VDD1;
        ...
    }
    pg_in (VDDL) {
        voltage_name : VDD2;
        ...
    }
    pin (I) {
        direction : input;
        level_shifter_data_pin : true;
        related_ground_pin : VSS;
        related_power_pin : VDDL;
    }
    pin (Z) {
        direction : output;
        power_down_function : "!VDD + !VDDL + VSS";
        function : "I";
        related_ground_pin : VSS;
        related_power_pin : VDD;
        ...
    }
    timing() {
        related_pin : I;
        ...
        ccsn_first_stage () {/* arc-based model */
            is_needed : true;
            stage_type : both;
            ...
            output_signal_level : VDD1;
            dc_current (ccsn_dc)...
        }
        ccsn_last_stage () {/* arc-based model */
            is_needed : "true";
            stage_type : "both";
            ...
        }
    }
}
```

```
        input_signal_level : VDD1;
        dc_current (ccsn_dc) ...
    }
}
...
}
...
}
...
}
```

Referenced CCS Noise Modeling

For CCS noise characterization, a circuit is partitioned into channel-connected blocks (CCBs). An input pin might simultaneously drive multiple channel-connected blocks (CCBs).

For different input states and capacitive loads, you can represent each channel-connected block using multiple `input_ccb` and `output_ccb` groups. The `input_ccb` and `output_ccb` groups have names and are defined in `pin` groups.

In each timing arc, you can use these names to reference the `input_ccb` or `output_ccb` groups that

- Contribute to the noise but do not propagate the noise in the timing arc; this also applies to pin-level `receiver_capacitance` groups
- Propagate the noise in the timing arc

Because you define the `input_ccb` and `output_ccb` groups only in the `pin` groups, this model eliminates multiple definitions of the same channel-connected block noise in different timing groups, such as for a `ccsn_last_stage` group representing an inverter in a conventional two-stage CCS noise model.

To better model noise, the `input_ccb` and `output_ccb` groups include the `output_voltage_rise` and `output_voltage_fall` groups. The `output_voltage_rise` and `output_voltage_fall` values are characterized with driver waveforms instead of ramp waveforms as in the conventional two-stage CCS noise model.

For more information about the conventional two-stage CCS noise model, see [CCS Signal Integrity Modeling Syntax on page 639](#).

Modeling Syntax

```
library (library_name) {
    cell (cell_name) {
        driver_waveform : "waveform_name";
        driver_waveform_rise : "waveform_name";
```

```
driver_waveform_fall "waveform_name";
...
pin (pin_name) {
    direction : input;
    ...
    input_ccb (input_ccb_name1) {
        when : logical_expression;
        mode(mode_name, mode_value);
        related_ccb_node : "spice_node_name1";
        output_voltage_rise {
            ...
        }
        ...
    }
    input_ccb (input_ccb_name2) {
        when : logical_expression
        mode(mode_name, mode_value);
        related_ccb_node : "spice_node_name2";
        output_voltage_rise {
            ...
        }
        ...
    }
    ...
}
...
receiver_capacitance (lu_template) {
    when : logical_expression;
    mode(mode_name, mode_value);
    active_input_ccb(input_ccb_name1, input_ccb_name2,...]);
} /* end receiver_capacitance group */
} /* end pin a group */
pin (pin_name) {
    direction : output;
    ...
    output_ccb(output_ccb_name1) {
        when : logical_expression;
        mode(mode_name, mode_value);
        related_ccb_node : "spice_node_name3";
    }
    timing() /* Arc has propagating noise (full arc-based) */
    ...
    active_input_ccb(input_ccb_name1, input_ccb_name2, ...);
    propagating_ccb(input_ccb_name2, output_ccb_name);
}
timing() /* Arc has no propagating noise because it involves
            three inverting stages)*/
...
active_input_ccb(input_ccb_name1, input_ccb_name2);
active_output_ccb(output_ccb_name);
}
} /* end pin group */
} /* end cell group */
} /* end library group */
```

Cell-Level Attributes

This section describes the optional cell-level attributes specific to the referenced CCS noise model.

driver_waveform Attribute

The `driver_waveform` attribute specifies a cell-specific driver waveform. The specified driver waveform must be a predefined `driver_waveform_name` attribute value in the library-level `normalized_driver_waveform` group table.

driver_waveform_rise and driver_waveform_fall Attributes

The `driver_waveform_rise` and `driver_waveform_fall` specify the rise and fall driver waveforms. These attributes override the `driver_waveform` attribute.

For more information about driver waveform modeling syntax, see [Driver Waveform Support on page 282](#).

Pin-Level Groups

This section describes the pin-level groups and attributes specific to the referenced CCS noise model.

input_ccb and output_ccb Groups

The `input_ccb` and `output_ccb` groups include all the attributes and subgroups of the `ccsn_first_stage` and `ccsn_last_stage` groups respectively.

Except for pins where the `input_ccb` and `output_ccb` groups are not required, you must name all these groups so that they can be referenced. For the exceptions, you must set the `is_needed` attribute to `false` in the `input_ccb` and `output_ccb` groups.

The name includes the pin identifier and the group identifier. The pin identifier is the same in all the `input_ccb` and `output_ccb` groups of the pin. The pin identifier enables homogeneous treatment of these groups in buses and bundles.

For example, in the `input_ccb("CCB_D1"){...}` group, the `CCB_D` part of the name indicates that the channel-connected block group belongs to pin D and 1 indicates that this is the first `input_ccb` group of pin D.

Note:

The library characterization tools can use an additional cell identifier in the `input_ccb` and `output_ccb` group names.

Conditional Data Modeling

Use the `mode` and `when` attributes in the `input_ccb` and `output_ccb` groups for conditional data modeling.

`related_ccb_node` Attribute

The optional `related_ccb_node` attribute specifies the SPICE node in the subcircuit netlist that is used for the `dc_current` table characterization. The attribute is defined in the `input_ccb` group of an input pin and the `output_ccb` group of an output pin.

`output_voltage_rise` and `output_voltage_fall` Groups

The `output_voltage_rise` and `output_voltage_fall` groups have the following differences from those in the CCS noise stage groups.

- The output waveform specified by the `index_3` and `values` attributes is characterized using a driver waveform and not a ramp waveform. So, the `index_1` or `input_net_transition` is a slew value based on slew trip points and library slew derate and not a rail-to-rail ramp transition time.
- The `index_1` (`input_net_transition`) `values` must match one of the `index_1` values of the driver waveform specified at the cell-level.
- The `index_2` (`total_output_net_capacitance`) `values` must match one of the `index_2` values of the `cell_rise` and `cell_fall` groups.
- The `values` at `index_1` and `index_2` of all the `output_voltage_rise` or `output_voltage_fall` groups in an `input_ccb` or `output_ccb` group must form a grid without duplicated or missing points.

For an `input_ccb` group that does not directly drive an output pin, the number of `output_voltage_rise` groups must be $M \times 1$. For an `output_ccb` or an `input_ccb` that directly drives an output pin, the number of groups must be $M \times N$.

M is the number of `index_1` values and N is the number of `index_2` values.

Note:

For libraries with 7×7 or 8×8 nonlinear delay model (NLDM) tables, both M and N can be 4.

Timing-Level Attributes

This section describes the timing-level attributes specific to the referenced CCS noise model.

active_input_ccb Attribute

The `active_input_ccb` attribute lists the `input_ccb` groups of the input pin that are active or switching in the timing arc or the receiver capacitance load but do not propagate the noise to the output pin. In the `timing` group, the input pin is specified by the `related_pin` attribute.

You can also specify this attribute in the `receiver_capacitance` group of the input pin.

active_output_ccb Attribute

The `active_output_ccb` attribute lists the `output_ccb` groups in the timing arc that drive the output pin, but do not propagate the noise. You must define both the `output_ccb` and `timing` groups in the same `pin` group.

propagating_ccb Attribute

The `propagating_ccb` attribute lists all the channel-connected block groups that propagate the noise in a particular timing arc, to the output pin.

In the list, the first name is the `input_ccb` group of the input pin (specified by the `related_pin` attribute in the `timing` group). The second name, if present, is for the `output_ccb` group of the output pin.

Note:

You can list at most two names. This attribute does not support timing arcs with three or more inverting stages. For such cases and complex circuits with converging paths, reference the `input_ccb` group name in the `active_input_ccb` attribute and the `output_ccb` group name in the `active_output_ccb` attribute.

Examples

The following example shows a simple noise model with two timing arcs sharing the same output channel-connected block.

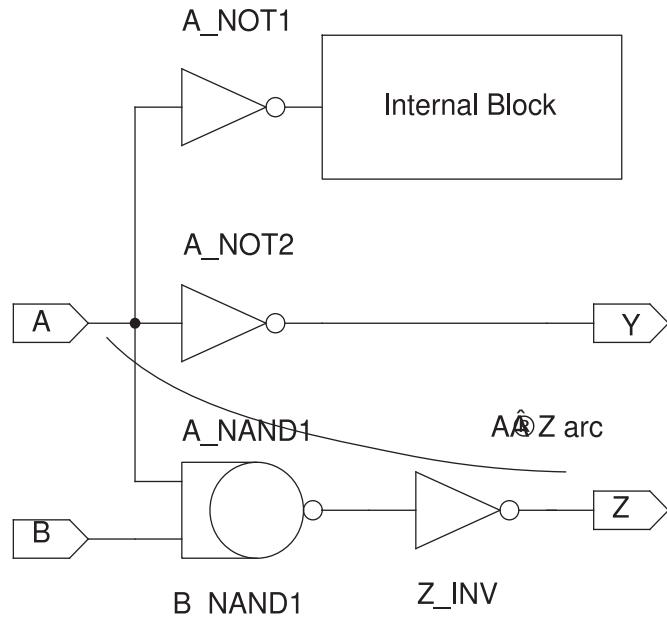
```
cell (AND2) {
    ...
    pin (A) {
        ...
        input_ccb("CCB_A") {
            related_ccb_node : "net1:15";
            ...
        }
    }
    pin (B) {
        ...
        input_ccb("CCB_B") {
```

Chapter 13: Composite Current Source Signal Integrity Modeling
Referenced CCS Noise Modeling

```
    related_ccb_node : "net1:15";
    ...
}
input_ccb("CCB_CP2") {
    related_ccb_node : "net3:3";
    ...
}
pin (Y) {
    ...
    output_ccb("CCB_Y") {
        related_ccb_node : "net1:15";
        ...
    }
    timing () {
        related_pin : A;
        propagating_ccb("CCB_A", "CCB_Y");
    }
    timing () {
        related_pin : B;
        propagating_ccb("CCB_B", "CCB_Y");
    }
    ...
}
}
```

The following figure is a schematic with the input, A, driving three logic gates. The relevant channel-connected blocks are shown.

Figure 114 A Schematic With Multiple Active Channel-Connected Blocks



The following table shows which of the four active channel-connected blocks propagate the noise in the $A \rightarrow Z$ timing arc.

Active channel-connected block	Propagates noise in $A \rightarrow Z$ arc?
A_NOT1	No
A_NOT2	No
A_NAND1	Yes
Z_INV	Yes

The following example shows how the blocks are referenced in the timing group.

```

pin(Z) {
    direction : output;
    timing () { /* A→Z arc */
        related_pin : A;
    ...
        active_input_ccb("A_NOT1", "A_NOT2");
        propagating_ccb("A_NAND1", "Z_INV");
    } /* end timing */
}
    
```

Chapter 13: Composite Current Source Signal Integrity Modeling
Referenced CCS Noise Modeling

}

14

Composite Current Source Power Modeling

This chapter provides an overview of composite current source (CCS) modeling to support advanced technologies.

It covers the syntax for CCS power modeling in the following sections:

- [Composite Current Source Power Modeling](#)
 - [Compact CCS Power Modeling](#)
 - [Composite Current Source Dynamic Power Examples](#)
-

Composite Current Source Power Modeling

The library nonlinear power model format captures leakage power numbers in multiple input combinations to generate a state-dependent table. It also captures dynamic power of various input transition times and output load capacitance to create the state-dependent and path-dependent internal energy data.

The composite current source (CCS) power modeling format extends current library models to include current-based waveform data to provide a complete solution that addresses static and dynamic power. It also addresses dynamic IR drop. The following are features of this approach as compared to the nonlinear power model:

- Creates a single unified power library format suitable for power optimization, power analysis, and rail analysis.
- Captures a supply current waveform for each power or ground pin.
- Provides finer time resolution.
- Offers full multivoltage support.
- Captures equivalent parasitic data to perform fast and accurate rail analysis.
- Reduces the characterization runtime.

Cell Leakage Current

Because CCS power is current-based data, leakage current on the power and ground pins is captured instead of leakage power as specified in the nonlinear power model format. For information about gate leakage, see [gate_leakage Group on page 671](#). The leakage current syntax is as follows:

Example 132 Leakage Current Syntax

```
cell(cell_name) {  
    ...  
    leakage_current() {  
        when : "Boolean expression";  
        pg_current(pg_pin_name) {  
            value : float;  
        }  
        ...  
    } /* 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.

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 133 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 */  
        ...  
    }  
}
```

Gate Leakage Modeling in Leakage Current

The syntax for these power models is described in the following section.

Syntax

```
cell(cell_name) {  
    ...  
    leakage_current() {  
        when : "Boolean expression";  
        pg_current(pg pin name) {  
            value : float;  
        }  
        ...  
        gate_leakage(input pin name) {  
            input_low_value : float;  
            input_high_value : float;  
        }  
        ...  
        }  
        ...  
        leakage_current() {  
            /* group without when statement */  
            /* default state */  
            ...  
        }  
    }  
}
```

gate_leakage Group

This group specifies the cell's gate leakage current on input or inout pins within the `leakage_current` group in a cell. For information about cell leakage, see [Cell Leakage Current on page 670](#).

The following information pertains to a `gate_leakage` group:

- Groups can be placed in any order if there are more than one `gate_leakage` groups within a `leakage_current` group.
- Leakage current of a cell is characterized with opened outputs, which means outputs of a modeling cell do not drive any other cells. Outputs are assumed to have zero static current during the measurement.
- A missing `gate_leakage` group is allowed for certain pins.
- Current conservation is applicable if it can be applied to higher error tolerance.

input_low_value Attribute

This attribute specifies gate leakage current on an input or inout pin when the pin is in a low state.

- A negative float value is required.
- The gate leakage current is measured from the power pin of the cell to the ground pin of its driver cell.
- The input pin is pulled low.
- The `input_low_value` attribute is not required for a `gate_leakage` group.
- Defaults to 0 if no `gate_leakage` group is specified for certain pins.
- Defaults to 0 if no `input_low_value` attribute is specified in the `gate_leakage` group.

input_high_value Attribute

This attribute specifies gate leakage current on an input or inout pin when the pin is in a high state.

- A positive float value is required.
- The gate leakage current is measured from the power pin of its driver cell to the ground pin of the cell.
- The input pin is pulled high.
- The `input_high_value` is not required for a `gate_leakage` group.
- Defaults to 0 if no `gate_leakage` groups is specified for certain pins.
- Defaults to 0 if no `input_high_value` is specified in the `gate_leakage` group.

Intrinsic Parasitic Models

The intrinsic parasitic model syntax consists of two parts: intrinsic resistance and intrinsic capacitance.

```
cell (cell_name) {
    mode_definition (mode_name) {
        mode_value (namestring) {
            when : "Boolean expression";
            sdf_cond : "Boolean expression";
        }
    ...
    intrinsic_parasitic() {
        mode (mode_name, mode_value);
        when : "Boolean expression";
    }
}
```

Chapter 14: Composite Current Source Power Modeling

Composite Current Source Power Modeling

```
intrinsic_resistance(pg_pin_name) {
    related_output : output_pin_name;
    value : float;
}
intrinsic_capacitance(pg_pin_name) {
    value : float;
}
}

intrinsic_parasitic() {
    /*without when statement */
    /* default state */
}
```

Example 134 Typical Intrinsic Parasitic Model Using Modes

```
library (csm13os120_typ) {
    technology ( cmos ) ;
    delay_model : table_lookup;
    lu_table_template(ccsn_dc_29x29) {
        variable_1 : input_voltage;
        variable_2 : output_voltage;
    }
    cell(inv0d0) {
        area : 0.75;
        pg_pin(V1) {
            voltage_name : VDD1;
            pg_type : primary_power;
        }
        pg_pin(G1) {
            voltage_name : GND1;
            pg_type : primary_ground;
        }
        mode_definition(rw) {
            mode_value(read) {
                when : "A1";
                sdf_cond : "A1 == 1";
            }
            mode_value(write) {
                when : "!A1";
                sdf_cond : "A1 == 0";
            }
        }
        pin(A1) {
            direction : input;
            capacitance : 0.1 ;
            related_power_pin : V1;
            related_ground_pin : G1;
        }
        pin(A2) {
            direction : input;
```

```
    capacitance : 0.1 ;
    related_power_pin : V1;
    related_ground_pin : G1;
}
pin(ZN) {
    direction : output;
    max_capacitance : 0.1;
    function : "!A1+A2";
    related_power_pin : V1;
    related_ground_pin : G1;
    timing() {
        timing_sense : "negative_unate"
        related_pin : "A1";
        ...
    }
}
pin(ZN1) {
    direction : output;
    max_capacitance : 0.1;
    function : "!A1";
    related_power_pin : V1;
    related_ground_pin : G1;
    timing() {
        timing_sense : "negative_unate"
        related_pin : "A1 A2";
    ...
    }
}
intrinsic_parasitic() {
    mode(rw, read);
    intrinsic_resistance(G1) {
        related_output : "ZN";
        value : 9.0;
    }
    intrinsic_capacitance(G1) {
        value : 8.2;
    }
}
} /* cell(inv0d0) */
} /* library
```

Voltage-Dependent Intrinsic Parasitic Models

Intrinsic parasitics are conventionally modeled as voltage-independent or steady-state values. However, intrinsic parasitics are voltage-dependent. To better represent intrinsic parasitics in a CCS power model, use a lookup table for intrinsic parasitics instead of a single steady-state value. You can use the steady-state value when your design requirements are not critical.

The lookup table is one-dimensional and consists of intrinsic parasitic values for different values of VDD. You can selectively add these values to any `intrinsic_resistance` or

`intrinsic_capacitance` subgroup. Use lookup tables when correct estimation of voltage drops is critical, such as power-switch designs.

The following are the advantages of using lookup tables.

- Accurate estimation of peak-inrush current and wake-up time
- Optimal power-up and power-down sequencing
- Optimal power-switch design, that is, minimum number of used and placed power-switch cells

Example 135 Syntax of Voltage-Dependent Intrinsic Parasitic Model With Lookup Tables

```
lu_table_template (template_name) {
variable_1 : pg_voltage | pg_voltage_difference ;
index_1 ("float, ... float");
}
cell (cell_name) {
...
intrinsic_parasitic() {
    when : "Boolean expression";
    intrinsic_resistance (pg_pin_name) {
        related_output : output_pin_name;
        value : float;
        reference_pg_pin : pg_pin_name;
        lut_values (template_name) {
            index_1 ("float, ... float");
            values ("float, ... float");
        }
    }
    intrinsic_capacitance(pg_pin_name) {
        value : float;
        reference_pg_pin : pg_pin_name;
        lut_values (template_name) {
            index_1 ("float, ... float");
            values ("float, ... float");
        }
    }
}
...
}
```

Example 136 Example of Voltage-Dependent Intrinsic Parasitic Model

```
library(example_library) {
    .....
    lu_table_template ( test_voltage ) {
        variable_1 : pg_voltage;
        index_1 ("0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0" );
    }
    cell (AND3) {
        .....
```

Chapter 14: Composite Current Source Power Modeling

Composite Current Source Power Modeling

```
intrinsic_parasitic() {
    when : "A1 & A2 & ZN";
    intrinsic_resistance(G1) {
        related_output : "ZN";
        value : 9.0;
        reference_pg_pin : G1;
        lut_values ( test_voltage ) {
            index_1 ( "0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9,
1.0" );
            values ( "0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9,
1.0" );
        }
    }
    intrinsic_capacitance(G2) {
        value : 8.2;
        reference_pg_pin : G2;
        lut_values ( test_voltage ) {
            index_1 ( "0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9,
1.0" );
            values ( "0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9,
1.0" );
        }
    }
    intrinsic_resistance(G1) {
        related_output : "ZN1";
        value : 62.2;
    }
}
intrinsic_parasitic() {
/* default state */
    intrinsic_resistance(G1) {
        related_output : "ZN";
        value : 9.0;
        reference_pg_pin : G1;
        lut_values ( test_voltage ) {
            index_1 ( "0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9,
1.0" );
            values ( "0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9,
1.0" );
        }
    }
    intrinsic_resistance(G1) {
        related_output : "ZN1";
        value : 9.0;
    }
    intrinsic_capacitance(G2) {
        value : 8.2;
        reference_pg_pin : G2;
        lut_values ( test_voltage ) {
            index_1 ( "0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9,
1.0" );
            values ( "0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9,
1.0" );
        }
    }
}
```

```
        }
    }
    .....
} /* end of cell AND3 */
}
```

Library-Level Group

The following library-level group models voltage-dependent intrinsic parasitics.

lu_table_template Group

This group defines the template for the `lut_values` group. The `lu_table_template` group includes a one-dimensional variable, `variable_1`. The valid values of the `variable_1` variable are `pg_voltage` and `pg_voltage_difference`. When the `variable_1` variable is set to the `pg_voltage` value, the values of the intrinsic capacitance or resistance directly vary with the power supply voltage of the cell. When the `variable_1` variable is set to the `pg_voltage_difference` value, the values of the intrinsic capacitance or resistance vary with the difference between the power supply voltage and the power pin voltage specified by the `reference_pg_pin` attribute.

Note:

The `reference_pg_pin` attribute specifies the reference pin for the `intrinsic_resistance` and `intrinsic_capacitance` groups. The reference pin must be a valid PG pin.

Pin-Level Group

The following pin-level group models voltage-dependency in intrinsic parasitics by using lookup tables.

lut_values Group

To use the lookup table for intrinsic parasitics, use the `lut_values` group. You can add the `lut_values` group to both the `intrinsic_resistance` and `intrinsic_capacitance` groups. The `lut_values` group uses the `variable_1` variable, which is defined within the `lu_table_template` group, at the library level.

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 parasitics modeling of the total capacitance in macros cells is not state dependent. This means that there is no state condition specified in the `intrinsic_parasitic` group.

Parasitics Modeling Syntax

```
cell (cell_name) {
    power_cell_type : enum(stdcell, macro);
    ...
    intrinsic_parasitic() {
        total_capacitance(pg pin name) {
            value : float;
        }
        ...
    }
    ...
}
```

Dynamic Power

Because CCS power is current-based data, instantaneous power data on the power or ground pin is captured instead of internal energy specified in the nonlinear power model format. The current-based data provides higher accuracy than the existing model.

In the CCS modeling format, instantaneous power data is specified as a table of current waveforms. The table is dependent on the transition time of a toggling input and the capacitance of the toggling outputs.

As the number of output pins increases in a cell, the number of waveform tables becomes large. However, the cell with multiple output pins (more than one output) does not need to be characterized for all possible output load combinations. Therefore, two types of methods can be introduced to simplify the captured data.

- Cross type - Only one output capacitance is swept, while all other output capacitances are held in a typical value or fixed value.
- Diagonal type - The capacitance to all the output pins is swept together by an identical value.

A table that is modeled based on these two types is defined as a sparse table. Otherwise it is defined as a dense table, meaning that all combinations of the output load variable are specified in tables.

Dynamic Power and Ground Current Table Syntax

You can use the following syntax for dynamic current:

```
pg_current_template(template_name_1) {
    variable_1 : input_net_transition;
    variable_2 : total_output_net_capacitance;
    variable_3 : time;
    index_1(float, ); /* optional */
    index_2(float, ); /* optional */
    index_3(float, ); /* optional */
}

pg_current_template(template_name_2) {
    variable_1 : input_net_transition;
    variable_2 : total_output_net_capacitance;
    variable_3 : total_output_net_capacitance;
    variable_4 : time;
    index_1(float, ); /* optional */
    index_2(float, ); /* optional */
    index_3(float, ); /* optional */
    index_4(float, ); /* optional */
}
```

Dynamic Power Modeling in Macro Cells

The extensions to CCS dynamic power format provides more accurate models for macro cells. The current dynamic power model only supports current waveforms for single-input events.

The model can also be applied to memory modeling with synchronous events, which are triggered by toggling either a single `read_enable` or `write_enable`.

However, for asynchronous event, the read access can be triggered by more than one bit of the address bus toggling. To support asynchronous memory access for macro cells, use `min_input_switching_count` and `max_input_switching_count`, in dynamic power as shown in the next section.

The following syntax for dynamic power format provides more accurate models for macro cells:

```
...
cell(cell_name) {
mode_definition(mode_name) {
    mode_value(namestring) {
        when : "Boolean expression";
        sdf_cond : "Boolean expression";
    }
}
...
power_cell_type : enum(stdcell, macro)
dynamic_current() {
    mode(mode_name, mode_value);
```

```
when : "Boolean expression";
related_inputs : "input_pin_name";
switching_group() {
    min_input_switching_count : integer;
    max_input_switching_count : integer;
    pg_current(pg_pin_name) {
        vector(template_name) {
            reference_time : float;
            index_1(float);
            index_2("float,...");
            values("float,...");
        } /* end vector group*/
    ...
} /* end pg_current group */
...
} /* end switching_group */
...
} /* end dynamic_current group */
...
} /* end cell group*/
...
```

The `min_input_switching_count` and `max_input_switching_count` attributes specify the number of bits in the input bus that are switching simultaneously while an asynchronous event occurs. A single switching bit can be defined by setting the same value in both attributes.

The following example shows that any three bits specified in `related_inputs` are switching simultaneously.

```
...
min_input_switching_count : 3;
max_input_switching_count : 3;
...
```

A range of switching bits can be defined by setting the minimum and maximum value. The following example shows that any 2, 3, 4 or 5 bits specified in `related_inputs` are switching simultaneously.

```
...
min_input_switching_count : 2;
max_input_switching_count : 5;
...
```

min_input_switching_count Attribute

This attribute specifies the minimum number of bits in an input bus that are switching simultaneously.

- The count must be integer.
- The count must greater than 0 and less than `max_input_switching_count`.

max_input_switching_count Attribute

This attribute specifies the maximum number of bits in an input bus that are switching simultaneously.

- The count must be integer.
- The count must greater than `min_input_switching_count`.
- The count must be less than the total number of bits listed in `related_inputs`.

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" )
```

Chapter 14: Composite Current Source Power Modeling

Composite Current Source Power Modeling

```
        index_2 ( "2.4, 2.6, 2.9, 4.0" )
        values ( "0.012, 0.109, 0.534, 0.746")
    }
    vector ( CCS_power_1 ) {
        reference_time : 0.01;
        index_1 ("0.08")
        index_2 ( "1.0, 1.6, 1.8, 1.9" )
        values ( "0.102, 0.209, 0.474, 0.992")
    }
    ...
}
/* pg_current */
...
} /* switching_group */
...
} /* dynamic_current */
...
...
intrinsic_parasitic() {
    total_capacitance(VDD) {
        value : 0.2;
    }
    ...
}
/* intrinsic_parasitic */
...
...
leakage_current() {
    when : WE;
    gate_leakage(data_in) {
        input_low_value : -0.3;
        input_high_value : 0.5;
    }
    ...
}
/* leakage_current */
...
} /* end of cell */
...
```

Conditional Data Modeling for Dynamic Current Example

```
library (csm13os120_typ) {
    technology ( cmos ) ;
    delay_model : table_lookup;
    lu_table_template(ccsn_dc_29x29) {
        variable_1 : input_voltage;
        variable_2 : output_voltage;
    }
    cell(inv0d0) {
        area : 0.75;
        pg_pin(V1) {
            voltage_name : VDD1;
            pg_type : primary_power;
        }
        pg_pin(G1) {
```

Chapter 14: Composite Current Source Power Modeling

Composite Current Source Power Modeling

```
voltage_name : GND1;
pg_type : primary_ground;
}
mode_definition(rw) {
    mode_value(read) {
        when : "A1";
        sdf_cond : "A1 == 1";
    }
    mode_value(write) {
        when : "!A1";
        sdf_cond : "A1 == 0";
    }
}
power_cell_type : stdcell;
dynamic_current() /* dense table */
mode(rw, read);
related_inputs : "A2";
related_outputs : "ZN ZN1";
switching_group() {
    output_switching_condition(rise rise);
    pg_current(V1) {
        vector(test_1) {
            reference_time : 23.7;
            index_1("0.8");
            index_2("0.7");
            index_3("10.4");
            index_4("8.2 8.5 9.1 9.4 9.8");
            values("0.7 34.6 3.78 92.4 100.1");
        }
        ...
    }
}
pin(A1) {
    direction : input;
    capacitance : 0.1 ;
    related_power_pin : V1;
    related_ground_pin : G1;
}
pin(A2) {
    direction : input;
    capacitance : 0.1 ;
    related_power_pin : V1;
    related_ground_pin : G1;
}
pin(ZN) {
    direction : output;
    max_capacitance : 0.1;
    function : "!A1+A2";
    related_power_pin : V1;
    related_ground_pin : G1;
    timing() {
        timing_sense : "negative_unate"
```

```
        related_pin : "A1";
        ...
    }
}
pin(ZN1) {
    direction : output;
    max_capacitance : 0.1;
    function : "!A1";
    related_power_pin : V1;
    related_ground_pin : G1;
    timing() {
        timing_sense : "negative_unate"
        related_pin : "A1 A2";
        ...
    }
}
/* cell(inv0d0) */
} /* library
```

Dynamic Current Syntax

The syntax in [Example 137](#) is used for instantaneous power data, which is captured at the cell level.

Example 137 Dynamic Current Syntax

```
cell(cell_name) {

    power_cell_type : enum(stdcell, macro)
    dynamic_current() {
        when : "Boolean expression";
        related_inputs : input_pin_name;
        related_outputs : output_pin_name;
        typical_capacitances(float, ); /* applied for cross type; */
        switching_group() {
            input_switching_condition(enum(rise, fall));
            output_switching_condition(enum(rise, fall));
            pg_current(pg_pin_name) {
                vector(template_name) {
                    reference_time : float;
                    index_output : output_pin_name; /* applied for
cross type; */
                    index_1(float);

                    index_n(float);
                    index_n+1(float, );
                    values(float, );
                } /* vector */
            } /* pg_current */
        }
    }
}
```

```
    } /* switching_group */  
  
} /* dynamic_current */  
  
} /* cell */
```

Note:

For the `input_switching_condition` and `input_switching_condition` groups, the `rise` and `fall` values can be separated by commas or space.

Compact CCS Power Modeling

CCS power compaction uses base curve technology to significantly reduce the library size of CCS power libraries. Greater control of the Liberty file size allows you to include additional data points and more accurately capture CCS power data for dynamic current waveforms.

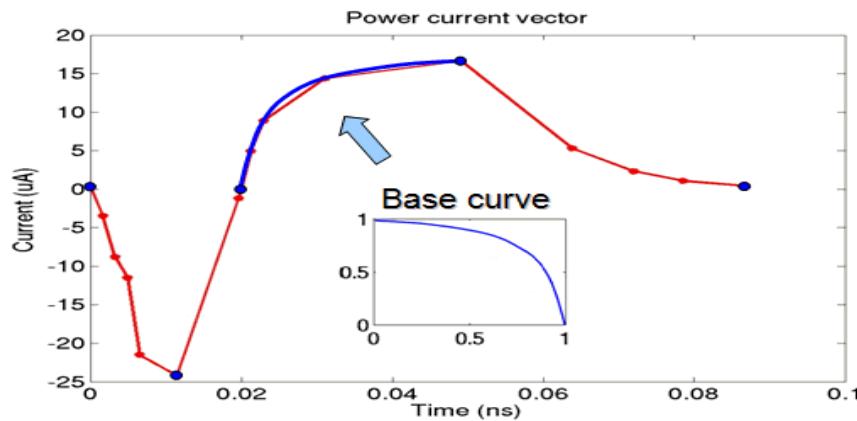
Base curve technology was first introduced for compact CCS timing. Each timing current waveform is split into two segments in the I-V domain, and the shape of each segment is modeled by a base curve that has a similar shape. This segmentation prevents direct modeling with the piecewise linear data points.

Compact CCS power modeling differs from compact CCS timing modeling in that power waveforms can include both positive sections and negative sections. They must be modeled in an $I(t)$ domain. In addition, the segmentation is more flexible. This is important because the current waveform shape might contain one or more bumps. The compact CCS power modeling segmentation points can be selected at:

- The point where the current waveform crosses zero
- The peak of a current bump

In [Figure 115](#), the red curve shows an example CCS power waveform in piecewise linear format with fifteen data points. Thirty float numbers must be stored in the library. This is an expensive storage cost for a single $I(t)$ waveform because libraries generally include a large number of $I(t)$ waveforms. In addition, the waveform shape is not smooth, despite the fifteen data points, due to the inefficiency of the piecewise linear representation. The current value error is larger than 5% in some regions of the waveform.

Figure 115 Power Current Vector



Segmenting the waveform and using base curve technology for each segment provides greater accuracy. In [Figure 115](#), the blue dots and blue curve show the waveform using base curve technology. The blue dots represent five segmentation points that divide the waveform into four sections. You can save the segmentation points as characteristic points and model the shape of each segment using a base curve. The blue curve is the third segment that can be represented by a base curve.

The following example describes the format for each current waveform:

$t_{start}, I_{start}, bcid_1, t_{ip1}, I_{ip1}, bcid_2, [t_{ip2}, I_{ip2}, bcid_3, \dots,] t_{end}, I_{end};$

The arguments are defined as follows:

t_{start}

The current start time.

I_{start}

The initial current.

t_{ip}

The time of an internal segmentation point.

I_{ip}

The current value of an internal segmentation point.

t_{end}

The time when transition ends and current value becomes stable.

I_{end}

The current value at the endpoint.

bcid

The ID of the base curve that models the shape between two neighboring points.

Compact CCS Power Syntax and Requirements

The expanded, or dynamic, CCS power model syntax provides an important reference and criteria for compact CCS power modeling. See [Dynamic Current Syntax on page 684](#) for the `dynamic_current` syntax and see [Dynamic Power and Ground Current Table Syntax on page 679](#) for the `pg_current_template` syntax.

The following requirements must be met in the `pg_current_template` group:

- The `last variable_*` value must be `time`. The `time` variable is required.
- The `input_net_transition` and `total_output_net_capacitance` values are available for all `variable_*` attributes except the `last variable_*`. They can be placed in any order except `last`.

The following conditions describe the `pg_current_template` group:

- There can be zero or one `input_net_transition` variable.
- There can be zero, one, or two `total_output_net_capacitance` variables.
- Each `vector` group in the `pg_current` group describes a current waveform that is compacted into a table in the compact CCS power model.

Similar to the compact CCS timing modeling syntax, compact CCS power modeling syntax uses the `base_curves` group to describe normalized base curves and the `compact_lut_template` group as the compact current waveform template. However, the `compact_lut_template` group attributes are extended from three dimensions to four dimensions when CCS power models need two `total_output_net_capacitance` attributes.

The compact CCS power modeling syntax is as follows:

```
library (my_library) {
    base_curves(bc_name) {
        base_curve_type : enum (ccs_half_curve, ccs_timing_half_curve);
        curve_x ("float, ..., float");
        curve_y ("integer, float, ..., float"); /* base curve #1 */
        curve_y ("integer, float, ..., float"); /* base curve #2 */
        ...
        curve_y ("integer, float, ..., float"); /* base curve #n */
    }
}
```

```
compact_lut_template (template_name) {
    base_curves_group : bc_name;
    variable_1 : input_net_transition | total_output_net_capacitance;
    variable_2 : input_net_transition | total_output_net_capacitance;
    variable_3 : input_net_transition | total_output_net_capacitance;
    variable_4 : curve_parameters;
    index_1 ("float, ..., float");
    index_2 ("float, ..., float");
    index_3 ("float, ..., float");
    index_4 ("string, ..., string");
}

...
cell(cell_name) {
    dynamic_current() {
        switching_group() {
            pg_current(pg_pin_name) {
                compact_ccs_power (template_name) {
                    base_curves_group : bc_name;
                    index_output : pin_name;
                    index_1 ("float, ..., float");
                    index_2 ("float, ..., float");
                    index_3 ("float, ..., float");
                    index_4 ("string, ..., string");
                    values ("float | integer, ..., float | integer");
                } /* end of compact_ccs_power */
            }
        }
    }
}
}
} /* end of cell */
}
} /* end of library*/
```

Library-Level Groups and Attributes

This section describes library-level groups and attributes used for compact CCS power modeling.

base_curves Group

The `base_curves` group contains the detailed description of normalized base curves. The `base_curves` group has the following attributes:

base_curve_type Complex Attribute

The `base_curve_type` attribute specifies the type of base curve. The `ccs_half_curve` value allows you to model compact CCS power and compact CCS timing data within

the same `base_curves` group. You must specify `ccs_half_curve` before specifying `ccs_timing_half_curve`.

curve_x Complex Attribute

The data array contains the x-axis values of the normalized base curve. Only one `curve_x` value is allowed for each `base_curves` group.

For a `ccs_timing_half_curve` base curve, the `curve_x` value must be between 0 and 1 and increase monotonically.

curve_y Complex Attribute

Each base curve consists of one `curve_x` and one `curve_y` attributes. You should define the `curve_x` base curve before `curve_y` for better clarity and easier implementation. The valid region for `curve_y` is [-30, 30] for compact CCS power.

There are two data sections in the `curve_y` complex attribute:

- The `curve_id` integer specifies the identifier of the base curve.
- The data array specifies the y-axis values of the normalized base curve.

compact_lut_template Group

The `compact_lut_template` group is a lookup table template used for compact CCS timing and power modeling.

The following requirements must be met for compact CCS power modeling:

- The last `variable_*` value must be `curve_parameters`.
- The `input_net_transition` and `total_output_net_capacitance` values are available for all `variable_*` attributes except the last `variable_*`. They can be placed in any order except last.

The following conditions describe the `pg_current_template` group:

- There can be zero or one `input_net_transition` variable.
- There can be zero, one, or two `total_output_net_capacitance` variables.
- The element type for all `index_*` values except the last one is a list of floating-point numbers.
- The element type for the last `index_*` value is a string.
- The only valid value for `index_*`, when it is last and when it is specified with `curve_parameters` as the last `variable_*`, is `init_time, init_current, bc_id1, point_time1, point_current1, bc_id2, [point_time2, point_current2, bc_id3, ...], end_time, end_current`.

The valid value in the last index is the pattern that all curve parameter series should follow. It is a pattern rather than a specified series because the table varies in size. Curve parameters define how to describe a current waveform. There should be at least two segments. The reference time for each current waveform is always zero. The negative time values, such as the values with corresponding parameters `init_time`, `point_time` and `end_time`, are permitted.

Note that this index is only for clarity. It is not used to determine the curve parameters. Curve parameters can be uniquely determined by the size of the values. A valid size can be represented as $(8+3i)$, where i is an integer and $i \geq 0$. The current waveform has $(i+2)$ segments.

Cell-Level Groups and Attributes

This section describes cell-level groups and attributes used for compact CCS power modeling.

compact_ccs_power Group

The `compact_ccs_power` group contains a detailed description for compact CCS power data. The `compact_ccs_power` group includes the following optional attributes: `base_curves_group`, `index_1`, `index_2`, `index_3` and `index_4`. The description for these attributes in the `compact_ccs_power` group is the same as in the `compact_lut_template` group. However, the attributes have a higher priority in the `compact_ccs_power` group. For more information, see [compact_lut_template Group on page 689](#).

The `index_output` attribute is also optional. It is used only on cross type tables.

values Attribute

The `values` attribute is required in the `compact_ccs_power` group. The data within the quotation marks (" "), or `line`, represent the current waveform for one index combination. Each value is determined by the corresponding curve parameter. In the following line,

"t0, c0, 1, t1, c1, 2, t2, c2, 3, t3, c3, 4, t4, c4"

the size is $14 = 8+3*2$. Therefore, the curve parameters are as follows:

```
"init_time, init_current, bc_id1, point_time1, point_current1, bc_id2, \
point_time2, point_current2, bc_id3, point_time3, point_current3,
bc_id4, \
end_time, end_current"
```

The elements in the `values` attribute are floating-point numbers for time and current and integers for the base curve ID. The number of current waveform segments can be different

for each slew and load combination, which means that each line size can be different. As a result, Liberty syntax supports tables with varying sizes, as shown:

```
compact_ccs_power (template_name) {
    ...
    index_1("0.1, 0.2"); /* input_net_transition */
    index_2("1.0, 2.0"); /* total_output_net_capacitance */
    index_3 ("init_time, init_current, bc_id1, point_time1,
    point_current,
    \
    bc_id2, [point_time2, point_current2, bc_id3, ...], \
    end_time, end_current"); /* curve_parameters */
    values
    ("t0, c0, 1, t1, c1, 2, t2, c2, 3, t3, c3, 4, t4, c4", \ /* segment=4 */
     "t0, c0, 1, t1, c1, 2, t2, c2", \ /* segment=2 */
     "t0, c0, 1, t1, c1, 2, t2, c2, 3, t3, c3", \ /* segment=3 */
     "t0, c0, 1, t1, c1, 2, t2, c2, 3, t3, c3"); /* segment=3 */
}
```

Composite Current Source Dynamic Power Examples

This section provides the following CCS dynamic power examples:

- [Design Cell With a Single Output Example](#)
- [Dense Table With Two Output Pins Example](#)
- [Cross Type With More Than One Output Pin Example](#)
- [Diagonal Type With More Than One Output Pin Example](#)

Design Cell With a Single Output Example

```
pg_current_template ( CCS_power_1 ) {
    variable_1 : input_net_transition ;
    variable_2 : total_output_net_capacitance ;
    variable_3 : time ;
}

cell (example) {
    dynamic_current() {
        when: D;
        related_inputs : CP;
        related_outputs : Q;
        switching_group ( ) {
            input_switching_condition(rise);
            output_switching_condition(rise);
            pg_current (VDD ) {
                vector ( CCS_power_1 ) {
                    reference_time : 0.01;
```

Chapter 14: Composite Current Source Power Modeling
Composite Current Source Dynamic Power Examples

```
        index_1 ( 0.01 )
        index_2 ( 1.0 )
        index_3 ( 0.000, 0.0873, 0.135, 0.764)
        values ( 0.002, 0.009, 0.134, 0.546)
    }
}
}
}
```

Dense Table With Two Output Pins Example

```
pg_current_template ( CCS_power_1 ) {
    variable_1 : input_net_transition ;
    variable_2 : total_output_net_capacitance ;
    variable_3 : total_output_net_capacitance ;
    variable_4 : time ;
}

cell ( example ) {
    dynamic_current() {
        related_inputs : "A" ;
        related_outputs : "Z" ;
        typical_capacitances(0.04);
        switching_group() {
            input_switching_condition(rise);
            output_switching_condition(rise);
            pg_current(VDD) {
                vector(ccsp_switching_load_time) {
                    reference_time : 0.0015 ;
                    index_1("0.0019");
                    index_2("0.001");
                    index_3("0, 0.006, 0.03, 0.07, 0.09, 0.1, 0.2, 0.3, 0.4,
                            0.5");
                    values("5e-06, 0.001, 0.02, 0.03, 0.05, 0.08, 0.09, 0.04,
                           0.009, 5.0e-06");
                }
            }
        }
    }
}
```

Cross Type With More Than One Output Pin Example

```
pg_current_template ( CCS_power_1 ) {
    variable_1 : input_net_transition ;
    variable_2 : total_output_net_capacitance ;
    variable_3 : time ;
}
```

Chapter 14: Composite Current Source Power Modeling

Composite Current Source Dynamic Power Examples

```
cell ( example )

dynamic_current() {
    when: D;
    related_inputs : CP;
    related_outputs : Q QN QN1 QN2;
    typical_capacitances(10.0 10.0 10.0 10.0);
    switching_group () {
        input_switching_condition(rise);
        output_switching_condition(rise, fall, fall, fall);
        pg_current (VSS) {
            vector ( CCS_power_1 ) {
                index_output : Q;
                reference_time : 0.01;
                index_1 ( 0.01 )
                index_2 ( 5.0 )
                index_3 ( 0.000, 0.0873, 0.135, 0.764)
                values ( 0.002, 0.009, 0.134, 0.546)
            }
            vector ( CCS_power_1 ) {
                index_output : QN;
                reference_time : 0.01;
                index_1 ( 0.01 )
                index_2 ( 1.0 )
                index_3 ( 0.000, 0.0873, 0.135, 0.764)
                values ( 0.002, 0.009, 0.134, 0.546)
            }
            vector ( CCS_power_1 ) {
                index_output : QN;
                reference_time : 0.01;
                index_1 ( 0.01 )
                index_2 ( 5.0 )
                index_3 ( 0.000, 0.0873, 0.135, 0.764)
                values ( 0.002, 0.009, 0.134, 0.546)
            }
        }
    }
}
```

Diagonal Type With More Than One Output Pin Example

```
pg_current_template ( CCS_power_1 ) {
    variable_1 : input_net_transition ;
    variable_2 : total_output_net_capacitance ;
    variable_3 : time ;
}
cell ( example )
dynamic_current() {
    when: D ;
    related_inputs : CP;
    related_outputs : Q QN QN1 QN2;
    switching_group () {
        input_switching_condition(rise);
```

Chapter 14: Composite Current Source Power Modeling
Composite Current Source Dynamic Power Examples

```
        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)
    }
}
```

15

On-Chip Variation (OCV) Modeling

On-chip variation (OCV) causes variation in the timing performance of each transistor in a design.

In advanced OCV models, these variations are modeled using derating factors based on the path depth and path distance.

In parametric OCV models, the variations are modeled using random variation and path distance of cells.

The OCV model definitions are covered in the following sections:

- [Advanced OCV Modeling](#)
-

Advanced OCV Modeling

In an advanced OCV model, the derating values are based on the path depth and path distance.

The modeling syntax is:

```
library (name) {
    ocv_table_template(ocv_template_name) {
        variable_1: path_depth|path_distance;
        variable_2: path_depth|path_distance;
        index_1 ("float..., float");
        index_2 ("float..., float");
    }
    ocv_derate(ocv_derate_group_name) {
        ocv_derate_factors(ocv_template_name) {
            rf_type: rise|fall|rise_and_fall;
            derate_type: early|late;
            path_type: clock|data|clock_and_data;
            index_1 ("float,..., float");
            index_2 ("float,..., float");
            values ( "float,..., float", \
                     ...,\ 
                     "float,..., float");
        }
    ...
}
```

```

}
default_ocv_derate_group: ocv_derate_group_name;
ocv_arc_depth: float;
...
cell (cell_name) {
    ocv_derate_group: ocv_derate_group_name;
    ocv_arc_depth: float;
    ...
    ocv_derate(ocv_derate_group_name) {
        ocv_derate_factors(ocv_template_name) {
            rf_type: rise|fall|rise_and_fall;
            derate_type: early|late;
            path_type: clock|data|clock_and_data;
            index_1 ("float,..., float");
            index_2 ("float,..., float");
            values ("float,..., float", \
                    ...,\ \
                    "float,..., float");
        }
    ...
}
pin | bus | bundle (name) {
    direction: inout | output;
    timing() {
        ocv_arc_depth: float;
        ...
    } /* end of timing */
} /* end of inout | output pin */
...
} /* end of cell */
...
} /* end of library */

```

Table 33 Association Between the Liberty Advanced OCV Syntax and the PrimeTime Advanced OCV File Format

Liberty syntax	PrimeTime advanced OCV file format
ocv_derate	group_name
ocv_table_template	table
path_depth	depth
path_distance	distance
ocv_derate_factors	table
rf_type	rf_type
derate_type	derate_type

Liberty syntax	PrimeTime advanced OCV file format
path_type	path_type
ocv_derate_group	group_name
default_ocv_derate_group	group_name
ocv_arc_depth	Not applicable

Library-Level Groups and Attributes

This section describes library-level groups and attributes for advanced OCV modeling that also apply to cells and timing arcs where specified.

ocv_table_template Group

The `ocv_table_template` group defines the template for an `ocv_derate_factors` group defined within the `ocv_derate` group.

The template definition includes two one-dimensional variables, `variable_1` and `variable_2`. These variables are used as indexes in the `ocv_derate_factors` group. You must specify `variable_1`. Specifying `variable_2` is optional.

Both `variable_1` and `variable_2` can have one of the following values:

- `path_depth`: The number of cells in a delay path.
- `path_distance`: The physical distance spanned by the path. Use the `distance_unit` attribute in the library to specify the path distance unit.

You can specify the following types of lookup tables within the `ocv_table_template` group:

- One-dimensional tables with the index consisting of `path_depth` or `path_distance` for a table
- Two-dimensional tables with one index consisting of `path_depth`, and the other index consisting of `path_distance`

This means that for a two-dimensional table, the values of `variable_1` and `variable_2` are different.

ocv_derate Group

The `ocv_derate` group contains a set of `ocv_derate_factors` groups. The `ocv_derate` group specifies a set of lookup tables with OCV derating factors for timing.

You must specify at least one `ocv_derate_factors` group within an `ocv_derate` group.

You can specify the `ocv_derate` group in the `library` and `cell` groups. To define multiple `ocv_derate` groups in a library, use different group names. If multiple `ocv_derate` groups have the same name, the group that is last specified overrides the previous ones.

`ocv_derate_factors` Group

The `ocv_derate_factors` group specifies a lookup table of the OCV derating factors. The lookup table can be one-dimensional, two-dimensional, or scalar. For a one-dimensional lookup table, the index consists of `path_depth` or `path_distance` values. For a two-dimensional lookup table, the indexes consist of the `path_depth` and `path_distance` values. Use the scalar to specify a single derating factor irrespective of the path depth and path distance.

Use the following attributes to define specific OCV derating factors in the `ocv_derate_factors` lookup table:

- `rf_type`

The `rf_type` attribute defines the type of delay specified in the `ocv_derate_factors` lookup table. Valid values are `rise`, `fall`, and `rise_and_fall`.

- `derate_type`

The `derate_type` attribute defines the type of arrival time specified in the `ocv_derate_factors` lookup table. The valid values are `early` and `late`.

- `path_type`

The `path_type` attribute defines the type of path specified in the `ocv_derate_factors` group. The valid values are `clock`, `data`, and `clock_and_data`.

These attributes do not have defaults. You must specify these attributes in the `ocv_derate_factors` group.

Applying OCV Derating Factors to Library Cells

To apply the set of derating factors defined within an `ocv_derate` group to a cell, use the cell-level `ocv_derate_group` attribute to specify the name of the `ocv_derate` group.

To apply the same set of derating factors to multiple cells in a library, specify the `ocv_derate` group at the library-level and define the name of the `ocv_derate` group with the `ocv_derate_group` attribute in the corresponding `cell` groups.

The set of derating factors defined within a cell-level `ocv_derate` group can be applied only to the particular cell.

If you do not specify the `ocv_derate_group` attribute, the default `ocv_derate` group defined at the library level applies to the cell.

default_ocv_derate_group Attribute

The `default_ocv_derate_group` attribute specifies the name of the default `ocv_derate` group for a library. The `default ocv_derate` group applies to the cell groups where the `ocv_derate` group name is not defined using the cell-level `ocv_derate_group` attribute.

ocv_arc_depth Attribute

The optional `ocv_arc_depth` attribute specifies the effective logic depth of a cell or a timing arc. The default is 1.0.

The path depth is used to determine the OCV derating factors from the lookup tables in the `ocv_derate_factors` group.

You can define the `ocv_arc_depth` attribute in the library, cell, or timing groups. If you define this attribute in different groups, the attribute value in the timing group overrides the value defined in the cell group, and the value defined in the cell group overrides the value defined in the library group.

Cell-Level Attribute

This section describes a cell-level attribute for advanced OCV modeling.

ocv_derate_group Attribute

The `ocv_derate_group` attribute specifies the name of the `ocv_derate` group that applies to a cell.

Advanced OCV Library Example

Example 138 A Typical Advanced OCV Library

```
library (OCV_lib) {
    ocv_table_template(2D_ocv_template) {
        variable_1: path_depth;
        variable_2: path_distance;
        index_1 ("1,2,3");
        index_2 ("1,2,3");
    }
    ocv_table_template(1D_ocv_template1) {
        variable_1: path_depth;
        index_1 ("1,2,3");
    }
    ocv_table_template(1D_ocv_template2) {
        variable_1: path_distance;
        index_1 ("1,2,3");
    }
    ocv_derate(advanced_ocv) {
```

Chapter 15: On-Chip Variation (OCV) Modeling
Advanced OCV Modeling

```
ocv_derate_factors(scalar) {
    rf_type: rise_and_fall;
    derate_type: early;
    path_type: clock_and_data;
    values ( "0.100");
}
ocv_derate_factors(scalar) {
    rf_type: rise_and_fall;
    derate_type: late;
    path_type: clock_and_data;
    values ( "0.200");
}
ocv_arc_depth: 1.0;
default_ocv_derate_group: advanced_ocv;
cell (cell1) {
    ocv_arc_depth: 2.0;
    ocv_derate_group: aocv1;
    ocv_derate(aocv1) {
        ocv_derate_factors (2D_ocv_template) {
            rf_type: rise_and_fall;
            derate_type: early;
            path_type: clock_and_data;
            index_1 ("1,2,3");
            index_2 ("4, 5");
            values ( "0.1, 0.2, 0.3",
                     "0.2, 0.3, 0.4");
        }
        ocv_derate_factors (2D_ocv_template) {
            rf_type: rise_and_fall;
            derate_type: late;
            path_type: clock_and_data;
            index_1 ("1,2,3");
            index_2 ("100, 200");
            values ( "0.1, 0.2, 0.3",
                     "0.2, 0.3, 0.4");
        }
    }
    ocv_derate(aocv2){
        ocv_derate_factors(scalar) {
            rf_type: rise_and_fall;
            derate_type: late;
            path_type: clock_and_data;
            values ( "0.900");
        }
        ocv_derate_factors(scalar) {
            rf_type: rise_and_fall;
            derate_type: early;
            path_type: clock_and_data;
            values ( "0.800");
        }
    }
...
}
```

```
pin | bus | bundle (name) {
    direction: inout | output;
    timing() {
        ocv_arc_depth: 3.0;
        ...
    } /* end of timing */
}
...
} /* end of cell */
cell (cell1) {
    ocv_derate_group: advanced_ocv;
}
...
} /* end of cell */
cell (cell3) {
    /* use advanced_ocv as default_ocv_derate_group
       defined in library */
}
...
} /* end of cell */
...
} /* end of library */
```

LVF Models For Cell Delay, Transition, and Constraint

In a parametric OCV (POCV) model, the random variation is specific to a cell or a timing arc, unlike an advanced OCV model where a specific derating factor applies to multiple cells or an entire library. The Liberty variation format (LVF) is used to represent the parametric OCV (POCV) library data, such as the slew-load table per delay timing arc. The following Liberty variation format (LVF) syntax consists of groups for sigma variation cell delay, output transition or slew, and constraint tables that are load and input-slew dependent. The variation values are used during parametric OCV analysis.

Syntax

The parametric OCV Liberty variation format (LVF) modeling syntax for cell delay, transition, and constraint is:

```
library (name) {
    lu_table_template(lu_template_name) {
        variable_1: input_net_transition;
        variable_2: total_output_net_capacitance;
        variable_3: related_out_total_output_net_capacitance;
        index_1 ("float, ..., float");
        index_2 ("float, ..., float");
        index_3 ("float, ..., float");
    }
    lu_table_template(constraint_lu_template_name) {
        variable_1: related_pin_transition;
    }
}
```

Chapter 15: On-Chip Variation (OCV) Modeling
LVF Models For Cell Delay, Transition, and Constraint

```
variable_2: constrained_pin_transition;
variable_3: related_out_total_output_net_capacitance | \
            related_out_output_net_length | \
            related_out_output_net_wire_cap | \
            related_out_output_net_pin_cap;
index_1 ("float, ..., float");
index_2 ("float, ..., float");
index_3 ("float, ..., float");
}
ocv_table_template(ocv_template_name) {
    variable_1: path_distance;
    index_1 ("float, ..., float");
}
ocv_derate(ocv_derate_group_name) {
    ocv_derate_factors(ocv_template_name) {
        rf_type: rise | fall | rise_and_fall;
        derate_type: early | late;
        path_type: clock | data | clock_and_data;
        index_1 ("float, ..., float");
        values ( "float, ..., float");
    }
    ...
}
default_ocv_derate_distance_group: ocv_derate_group_name;
...
cell (cell_name) {
    ocv_derate_distance_group: ocv_derate_group_name;
    ...
    pin | bus | bundle (name) {
        direction: inout | output;
        timing() {
            ...
            ocv_sigma_cell_rise(lu_template_name){
                sigma_type: early | late | early_and_late;
                index_1 ("float, ..., float");
                index_2 ("float, ..., float");
                values ( "float, ..., float", \
                    ...,\ \
                    "float, ..., float");
            }
            ocv_sigma_cell_fall(lu_template_name){
                sigma_type: early | late | early_and_late;
                index_1 ("float, ..., float");
                index_2 ("float, ..., float");
                values ( "float, ..., float", \
                    ...,\ \
                    "float, ..., float");
            }
            ocv_sigma_rise_transition(lu_template_name){
                sigma_type: early | late | early_and_late;
                index_1 ("float, ..., float");
                index_2 ("float, ..., float");
                index_3 ("float, ..., float");
            }
        }
    }
}
```

```
        values ( "float, ..., float", \
                  ...,
                  "float, ..., float");
    }
    ocv_sigma_fall_transition(lu_template_name) {
        sigma_type: early | late | early_and_late;
        index_1 ("float, ..., float");
        index_2 ("float, ..., float");
        index_3 ("float, ..., float");
        values ( "float, ..., float", \
                  ...,
                  "float, ..., float");
    }
    ...
} /* end of timing */
...
} /* end of pin */
...
pin | bus | bundle (name) {
    direction: input | inout;
    timing() {
        ...
        ocv_sigma_rise_constraint(constraint_lu_template_name) {
            index_1 ("float, ..., float");
            index_2 ("float, ..., float");
            values ( "float, ..., float", \
                      ...,
                      "float, ..., float");
        }
        ocv_sigma_fall_constraint(constraint_lu_template_name) {
            index_1 ("float, ..., float");
            index_2 ("float, ..., float");
            values ( "float, ..., float", \
                      ...,
                      "float, ..., float");
        }
        ...
    } /* end of timing */
}
...
} /* end of pin */
...
} /* end of cell */
...
} /* end of library */
```

Note:

For input pins, you can specify only the `ocv_sigma_rise_constraint` and `ocv_sigma_fall_constraint` tables in the corresponding timing groups.

For output pins, you can specify only the `ocv_sigma_cell_rise`, `ocv_sigma_cell_fall`, `ocv_sigma_rise_transition`, and `ocv_sigma_fall_transition` tables in the corresponding timing groups.

For inout pins, you can specify the `ocv_sigma_cell_rise`, `ocv_sigma_cell_fall`, `ocv_sigma_rise_transition`, and `ocv_sigma_fall_transition`, `ocv_sigma_rise_constraint`, and `ocv_sigma_fall_constraint` tables in the corresponding timing groups.

Do not specify the `sigma_type` attribute in the `ocv_sigma_rise_constraint` and `ocv_sigma_fall_constraint` groups.

Table 34 Association Between the Liberty Parametric OCV Syntax and the PrimeTime Parametric OCV File Format

Liberty syntax	PrimeTime parametric OCV file format
<code>ocv_derate</code>	<code>group_name</code>
<code>ocv_table_template</code>	<code>table</code>
<code>path_distance</code>	<code>distance</code>
<code>ocv_derate_factors</code>	<code>table</code>
<code>rf_type</code>	<code>rf_type</code>
<code>derate_type</code>	<code>derate_type</code>
<code>path_type</code>	<code>path_type</code>
<code>ocv_derate_distance_group</code>	<code>group_name</code>
<code>default_ocv_derate_distance_group</code>	<code>group_name</code>

Library-Level Groups and Attributes

This section describes library-level groups and attributes for parametric OCV modeling that also apply to cells and timing arcs where specified.

lu_table_template Group

The `lu_table_template` group defines the template for the following groups specified in the timing group:

- `ocv_sigma_cell_rise`
- `ocv_sigma_cell_fall`

- `ocv_sigma_rise_transition`
- `ocv_sigma_fall_transition`
- `ocv_sigma_rise_constraint`
- `ocv_sigma_fall_constraint`

For the `ocv_sigma_cell_rise`, `ocv_sigma_cell_fall`, `ocv_sigma_rise_transition`, and `ocv_sigma_fall_transition` groups, the template definition includes the `variable_1`, `variable_2`, `variable_3`, `index_1`, `index_2`, and `index_3` attributes. The `variable_1`, `variable_2` and `variable_3` attributes specify the index variables for the lookup tables (LUTs) in these groups. The `index_1`, `index_2`, and `index_3` attributes specify the numerical values of the variables.

The values of `variable_1`, `variable_2`, and `variable_3` attributes are `input_net_transition`, `total_output_net_capacitance`, and `related_out_total_output_net_capacitance`.

You can specify the following types of lookup tables within these groups:

- One-dimensional tables with a single index, such as `input_net_transition`
- Two-dimensional tables with the indexes, `input_net_transition` and `total_output_net_capacitance`
- Three-dimensional tables with the indexes, `input_net_transition`, `total_output_net_capacitance`, and `related_out_total_output_net_capacitance`
- A scalar that has a single variation value irrespective of the `input_net_transition`, `total_output_net_capacitance`, and `related_out_total_output_net_capacitance` values

For the `ocv_sigma_rise_constraint` and `ocv_sigma_fall_constraint` groups, the template definition includes the `variable_1`, `variable_2`, `variable_3`, `index_1`, `index_2`, and `index_3` attributes. The `variable_1`, `variable_2` and `variable_3` attributes specify the index variables for the lookup tables of the `ocv_sigma_rise_constraint` and `ocv_sigma_fall_constraint` groups. The `index_1`, `index_2`, and `index_3` attributes specify the numerical values of these variables.

The values of `variable_1` and `variable_2` are `related_pin_transition` and `constrained_pin_transition`, respectively. The values of `variable_3` can be `related_out_total_output_net_capacitance`, `related_out_output_net_length`, `related_out_net_wire_cap`, or `related_out_output_net_pin_cap`. You can specify the following types of lookup tables within these groups:

- One-dimensional tables with a single index, such as `related_pin_transition`

- Two-dimensional tables with the indexes, `related_pin_transition` and `constrained_pin_transition`
- Three-dimensional tables with the indexes, `related_pin_transition`, `constrained_pin_transition`, and a valid value of `variable_3`
- A scalar that has a single variation value irrespective of the `related_pin_transition`, `constrained_pin_transition`, and `variable_3` values

ocv_table_template Group

The `ocv_table_template` group defines the template for an `ocv_derate_factors` group defined within the `ocv_derate` group.

The template definition includes a one-dimensional variable, `variable_1`, that is used as an index in the `ocv_derate_factors` group. Valid value of `variable_1` is `path_distance` and represents the physical distance spanned by the path. Use the `distance_unit` attribute in the library to specify the path distance unit.

ocv_derate Group

The `ocv_derate` group contains a set of `ocv_derate_factors` groups. The `ocv_derate` group specifies a set of lookup tables with OCV derating factors for timing.

You must specify at least one `ocv_derate_factors` group within an `ocv_derate` group.

You can specify the `ocv_derate` group in the `library` and `cell` groups. To define multiple `ocv_derate` groups in a library, use different group names. If multiple `ocv_derate` groups have the same name, the group that is last-specified overrides the previous ones.

ocv_derate_factors Group

The `ocv_derate_factors` group specifies a lookup table of the OCV derating factors. The lookup table can be one-dimensional or scalar. For a one-dimensional lookup table, the index consists of `path_distance` values. Use the scalar to specify a single derating factor irrespective of the path distance.

Use the following attributes to define specific OCV derating factors in the `ocv_derate_factors` lookup table:

- `rf_type`

The `rf_type` attribute defines the type of delay specified in the `ocv_derate_factors` lookup table. Valid values are `rise`, `fall`, and `rise_and_fall`.

- `derate_type`

The `derate_type` attribute defines the type of arrival time specified in the `ocv_derate_factors` lookup table. The valid values are `early` and `late`.

- `path_type`

The `path_type` attribute defines the type of path specified in the `ocv_derate_factors` group. The valid values are `clock`, `data`, and `clock_and_data`.

These attributes do not have defaults. You must specify these attributes in the `ocv_derate_factors` group.

Applying Parametric OCV Derating Factors to Library Cells

To apply the set of derating factors defined within an `ocv_derate` group to a cell, use the cell-level `ocv_derate_distance_group` attribute to specify the name of the `ocv_derate` group.

To apply the same set of derating factors to multiple cells in a library, specify the `ocv_derate` group at the library-level and define the name of the `ocv_derate` group with the `ocv_derate_distance_group` attribute in the corresponding `cell` groups.

The set of derating factors defined within a cell-level `ocv_derate` group can be applied only to the particular cell.

If you do not specify the `ocv_derate_distance_group` attribute, the default `ocv_derate` group defined at the library level applies to the cell.

default_ocv_derate_distance_group Attribute

The `default_ocv_derate_group` attribute specifies the name of the default `ocv_derate` group for a library. The default `ocv_derate` group applies to the `cell` groups where the `ocv_derate` group name is not defined using the cell-level `ocv_derate_distance_group` attribute.

Cell-Level Attribute

This section describes a cell-level attribute for parametric OCV modeling.

ocv_derate_distance_group Attribute

The `ocv_derate_distance_group` attribute specifies the name of the `ocv_derate` group that applies to a cell.

Timing Arc Level Groups and Attributes

This section describes the timing arc-level groups for modeling delay variations. Each of the groups specify a lookup table. The lookup table can be one-dimensional, two-dimensional, three-dimensional, or scalar.

Define the template of the lookup table in the library-level `lu_table_template` group. To use the template, specify the name of the `lu_table_template` group as the arc-level group name.

ocv_sigma_cell_rise Group

The `ocv_sigma_cell_rise` group specifies a lookup table for the rise delay variation. In the lookup table, each absolute rise-delay variation offset from the corresponding nominal rise delay is specified at one sigma (σ), where sigma is the standard deviation of the rise delay distribution.

Note:

The nominal rise delay value is specified by the `cell_rise` group within the `timing` group. For more information, see [Chapter 7, Timing Arcs](#).

During parametric OCV analysis, the delay variation and the nominal rise delay are used to calculate the rise delay at one sigma (σ):

```
rise delay(±σ) = nominal rise delay ± ocv_sigma_cell_rise value
```

ocv_sigma_cell_fall Group

The `ocv_sigma_cell_fall` group specifies a lookup table for the fall delay variation. In the lookup table, each absolute fall-delay variation offset from the corresponding nominal fall delay is specified at one sigma (σ), where sigma is the standard deviation of the fall delay distribution.

Note:

The nominal fall delay value is specified by the `cell_fall` group within the `timing` group. For more information, [Chapter 7, Timing Arcs](#).

During OCV analysis, the delay variation and the nominal fall delay are used to calculate the fall delay at one sigma (σ):

```
fall delay(±σ) = nominal fall delay ± ocv_sigma_cell_fall value
```

ocv_sigma_rise_transition Group

The `ocv_sigma_rise_transition` group specifies a lookup table of the rise transition variation values. In the lookup table, each absolute rise-transition variation offset from the corresponding nominal rise transition is specified at one sigma (σ), where sigma is the standard deviation of the rise transition distribution.

Note:

The nominal rise transition value is specified by the `rise_transition` group within the `timing` group. For more information, see [Chapter 7, Timing Arcs](#).

During parametric OCV analysis, the rise transition variation and the nominal rise transition are used to calculate the rise transition at one sigma (σ):

```
rise transition(±σ) = nominal rise transition ± ocv_sigma_rise_transition value
```

ocv_sigma_fall_transition Group

The `ocv_sigma_fall_transition` group specifies a lookup table for the fall transition variation. In the lookup table, each absolute fall-transition variation offset from the nominal fall transition is specified at one sigma (σ), where sigma is the standard deviation of the fall transition distribution.

Note:

The nominal fall transition value is specified by the `fall_transition` group within the `timing` group. For more information, [Chapter 7, Timing Arcs](#)".

During parametric OCV analysis, the transition variation and the nominal fall transition are used to calculate the fall delay at one sigma (σ):

```
fall transition(±σ) = nominal fall transition ± ocv_sigma_fall_transition value
```

sigma_type Attribute

The optional `sigma_type` attribute defines the type of arrival time specified in the `ocv_sigma_cell_rise`, `ocv_sigma_cell_fall`, `ocv_sigma_rise_transition`, and `ocv_sigma_fall_transition` lookup tables. The values are `early`, `late`, and `early_and_late`. The default is `early_and_late`.

During parametric OCV analysis, when you specify the `sigma_type` attribute in these groups, the following values at $±\sigma$ are calculated as:

```
rise delay(+σ) = nominal cell rise + ocv_sigma_cell_riselate
rise delay(-σ) = nominal cell rise - ocv_sigma_cell_riseearly
fall delay(+σ) = nominal cell fall + ocv_sigma_cell_falllate
fall delay(-σ) = nominal cell fall - ocv_sigma_cell_fallearly

rise transition(+σ) = nominal rise transition + ocv_sigma_rise_transitionlate
rise transition(-σ) = nominal rise transition - ocv_sigma_rise_transitionearly
fall transition(+σ) = nominal fall transition + ocv_sigma_fall_transitionlate
fall transition(-σ) = nominal fall transition - ocv_sigma_fall_transitionearly
```

ocv_sigma_rise_constraint Group

The `ocv_sigma_rise_constraint` group specifies a lookup table of the rise constraint variation values. In the lookup table, each absolute rise-constraint variation offset from the nominal rise constraint is specified at one sigma (σ), where sigma is the standard deviation of the rise constraint distribution.

Note:

The nominal rise constraint value is specified by the `rise_constraint` group within the `timing` group. For more information, see [Chapter 7, Timing Arcs](#)".

During OCV analysis, the rise constraint variation and the nominal rise constraint are used to calculate the rise constraint at one sigma (σ):

```
rise constraint(±σ) = nominal rise constraint ± ocv_sigma_rise_constraint value
```

ocv_sigma_fall_constraint Group

The `ocv_sigma_fall_constraint` group specifies a lookup table for the fall constraint variation. In the lookup table, each absolute fall-constraint variation offset from the nominal fall constraint is specified at one sigma (σ), where sigma is the standard deviation of the fall constraint distribution.

Note:

The nominal fall constraint value is specified by the `fall_constraint` group within the `timing` group. For more information, [Chapter 7, Timing Arcs](#)".

During parametric OCV (POCV) analysis, the delay variation and the nominal fall delay are used to calculate the fall delay at one sigma (σ):

```
fall constraint(±σ) = fall_constraint ± ocv_sigma_fall_constraint value
```

Parametric OCV Library Example

Example 139 A Library Model for OCV Delays, Transitions, and Constraints

```
library (lib1) {
    lu_table_template(2D_lu_template) {
        variable_1: input_net_transition;
        variable_2: total_output_net_capacitance;
        index_1 ("1,2,3");
        index_2 ("1,2,3");
    }
    lu_table_template(3D_lu_template) {
        variable_1: input_net_transition;
        variable_2: total_output_net_capacitance;
        variable_3: related_out_total_output_net_capacitance;
        index_1 ("1,2,3");
        index_2 ("1,2,3");
        index_3 ("1,2,3");
    }
    lu_table_template(2D_constraint_lu_template) {
        variable_1: related_pin_transition;
        variable_2: constrained_pin_transition;
        index_1 ("1,2,3");
        index_2 ("1,2,3");
    }
    ocv_table_template(1D_ocv_template2) {
        variable_1: path_distance;
        index_1 ("1,2,3");
    }
}
```

Chapter 15: On-Chip Variation (OCV) Modeling
LVF Models For Cell Delay, Transition, and Constraint

```
}

ocv_derate(location_based_ocv) {
    ocv_derate_factors(1D_ocv_template2) {
        rf_type: rise_and_fall;
        derate_type: early;
        path_type: clock_and_data;
        index_1 ("1, 2");
        values ( "5.0, 6.0");
    }
}

ocv_derate(locv1) {
    ocv_derate_factors(1D_ocv_template2) {
        rf_type: rise_and_fall;
        derate_type: early;
        path_type: clock_and_data;
        index_1 ("1, 2");
        values ( "5.0, 6.0");
    }
}

default_ocv_derate_distance_group: location_based_ocv;
cell (cell1) {
    ocv_derate_distance_group: locv1;
    ...
    pin (pin1) {
        direction: output;
        timing() {
            ...
            related_pin : "rpin1" ;
            timing_type : rising_edge ;
            cell_rise( 2D_lu_template ) {
                index_1 ( "0.1, 0.2, 0.3");
                index_2 ( "0.4, 0.5, 0.6");
                values ( "0.1, 0.2, 0.3",
                         "0.2, 0.3, 0.4",
                         "0.3, 0.4, 0.5");
            }
            cell_fall( scalar ) {
                values ("0.100");
            }
            rise_transition( 2D_lu_template ) {
                index_1 ( "0.1, 0.2, 0.3");
                index_2 ( "0.4, 0.4, 0.5");
                values ( "0.1, 0.2, 0.3",
                         "0.3, 0.4, 0.5",
                         "0.5, 0.6, 0.7");
            }
            fall_transition( scalar ) {
                values ("0.100");
            }
            ocv_sigma_cell_rise( 2D_lu_template ) {
                sigma_type: early;
                index_1 ( "0.3, 0.4, 0.5");
                index_2 ( "0.1, 0.2, 0.3");
            }
        }
    }
}
```

Chapter 15: On-Chip Variation (OCV) Modeling
LVF Models For Cell Delay, Transition, and Constraint

```
values ( "0.1, 0.2, 0.3", \
         "0.2, 0.3, 0.4", \
         "0.3, 0.4, 0.5");
}
ocv_sigma_cell_rise( 2D_lu_template ){
    sigma_type: late;
    index_1 ( "0.3, 0.4, 0.5");
    index_2 ( "0.1, 0.2, 0.3");
    values ( "0.1, 0.2, 0.3", \
             "0.2, 0.3, 0.4", \
             "0.3, 0.4, 0.5");
}
ocv_sigma_cell_fall( scalar ){
    sigma_type: early;
    values ("0.1");
}
ocv_sigma_cell_fall( scalar ){
    sigma_type: late;
    values ("0.1");
}
ocv_sigma_rise_transition( 3D_lu_template ){
    sigma_type: early;
    index_1 ( "0.1, 0.2, 0.3");
    index_2 ( "0.4, 0.5, 0.6");
    index_3 ( "0.7, 0.8");
    values ("0.1, 0.2, 0.3", \
            "0.2, 0.3, 0.4", \
            "0.3, 0.4, 0.5", \
            "0.4, 0.5, 0.6", \
            "0.5, 0.6, 0.7", \
            "0.6, 0.7, 0.8");
}
ocv_sigma_rise_transition( 3D_lu_template ){
    sigma_type: late;
    index_1 ( "0.1, 0.2, 0.3");
    index_2 ( "0.4, 0.5, 0.6");
    index_3 ( "0.7, 0.8");
    values ("0.1, 0.2, 0.3", \
            "0.2, 0.3, 0.4", \
            "0.3, 0.4, 0.5", \
            "0.4, 0.5, 0.6", \
            "0.5, 0.6, 0.7", \
            "0.6, 0.7, 0.8");
}
ocv_sigma_fall_transition( scalar ){
    sigma_type: early_and_late;
    values ( "0.200");
}
...
} /* end of timing */
}
...
pin (pin2) {
```

```
direction: input | inout;
timing() {
    ...
    ocv_sigma_rise_constraint( 2D_lu_constraint_template ) {
        index_1 ( "0.3, 0.4, 0.5");
        index_2 ( "0.1, 0.2, 0.3");
        values ( "0.1, 0.2, 0.3", \
                  "0.2, 0.3, 0.4", \
                  "0.3, 0.4, 0.5");
    }
    ocv_sigma_fall_constraint( 2D_lu_constraint_template ) {
        index_1 ( "0.3, 0.4, 0.5");
        index_2 ( "0.1, 0.2, 0.3");
        values ( "0.1, 0.2, 0.3", \
                  "0.2, 0.3, 0.4", \
                  "0.3, 0.4, 0.5");
    }
    ...
} /* end of timing */
...
} /* end of pin */
...
} /* end of cell */
cell (cell2) {
/* use location_based_ocv as
   default_ocv_derate_distance_group defined in library */
...
} /* end of cell */
...
} /* end of library */
```

LVF Retain Arc Models

A retain arc models the time during which an output port retains its current logical value after a voltage rise or fall at a related input port.

For information about the nominal retain arc tables that store the retain arc delay and transition time, see [Chapter 7, Timing Arcs.](#)

The parametric OCV (POCV) models for retain arcs are stored in the Liberty variation format (LVF) in library source files.

Syntax

The following is the modeling syntax of LVF retain arc models.

```
library (name) {
    lu_table_template(lu_template_name) {
```

Chapter 15: On-Chip Variation (OCV) Modeling
LVF Retain Arc Models

```
variable_1: input_net_transition;
variable_2: total_output_net_capacitance;
variable_3: related_out_total_output_net_capacitance;
index_1 (float,, float);
index_2 (float,, float);
index_3 (float,, float);
}

cell (name) {
    pin|bus|bundle(name) {
        direction: inout|output;
        timing() {
            ocv_sigma_retaining_rise (lu_template_name) {
                sigma_type: early|late|early_and_late;
                index_1 (float,, float);
                index_2 (float,, float);
                index_3 (float,, float);
                values (float,, float,
                        ,
                        \,
                        float,, float);
            }
            ocv_sigma_retaining_fall(lu_template_name) {
                sigma_type: early|late|early_and_late;
                index_1 (float,, float);
                index_2 (float,, float);
                index_3 (float,, float);
                values (float,, float,
                        ,
                        \,
                        float,, float);
            }
            ocv_sigma_retain_rise_slew (lu_template_name) {
                sigma_type: early|late|early_and_late;
                index_1 (float,, float);
                index_2 (float,, float);
                index_3 (float,, float);
                values (float,, float,
                        ,
                        \,
                        float,, float);
            }
            ocv_sigma_retain_fall_slew(lu_template_name) {
                sigma_type: early|late|early_and_late;
                index_1 (float,, float);
                index_2 (float,, float);
                index_3 (float,, float);
                values (float,, float,
                        ,
                        \,
                        float,, float);
            }
        } /* end of timing */
    } /* end of inout/output pin */
} /* end of cell */
} /* end of library */
```

Library-Level Groups

This section describes library-level groups to model retain arc timing variation.

lu_table_template Group

The `lu_table_template` group defines the template for the following timing-level retain arc groups:

- `ocv_sigma_retaining_rise`
- `ocv_sigma_retaining_fall`
- `ocv_sigma_retain_rise_slew`
- `ocv_sigma_retain_fall_slew`

The template definition includes the `variable_1`, `variable_2`, `variable_3`, `index_1`, `index_2`, and `index_3` attributes. The `variable_1`, `variable_2`, and `variable_3` attributes specify the variables that are used as indexes in these groups. The `index_1`, `index_2`, and `index_3` attributes specify the numerical values of the variables. The values of `variable_1`, `variable_2`, and `variable_3` are `input_net_transition`, `total_output_net_capacitance`, and `related_out_total_output_net_capacitance` respectively. You can define a scalar, one-dimensional, two-dimensional, or three-dimensional lookup table using the `lu_table_template` group.

Timing Arc Level Groups

This section describes timing-level groups to model retain arc timing delay and transition variation. These groups are lookup tables that can be scalar, one, two, or three-dimensional.

Define these groups in the `timing` group together with the nominal retain arc tables.

ocv_sigma_retaining_rise and ocv_sigma_retaining_fall Groups

The `ocv_sigma_retaining_rise` and `ocv_sigma_retaining_fall` groups are retain arc delay lookup tables that specify the absolute variation offset from the nominal retain arc table values at one sigma (σ), where sigma is the standard deviation of the delay distribution.

During parametric OCV analysis, the retain arc values at $\pm\sigma$ are calculated using the nominal and variation values as:

```
retaining_rise(±σ) = retaining_rise ± ocv_sigma_retaining_rise  
retaining_fall(±σ) = retaining_fall ± ocv_sigma_retaining_fall
```

For more information about the nominal retain arc delay groups, see [Chapter 7, Timing Arcs](#).

ocv_sigma_retain_rise_slew and ocv_sigma_retain_fall_slew Groups

The `ocv_sigma_retain_rise_slew` and `ocv_sigma_retain_fall_slew` groups are retain arc slew lookup tables that specify the absolute variation offset values from the nominal retain arc table values at one sigma (σ), where sigma is the standard deviation of the transition distribution.

During parametric OCV analysis, the retain arc values at $\pm\sigma$ are calculated using the variation values as:

```
retain_rise_slew( $\pm\sigma$ ) = retain_rise_slew ±  
                           ocv_sigma_retain_rise_slew  
  
retain_fall_slew( $\pm\sigma$ ) = retain_fall_slew ±  
                           ocv_sigma_retain_fall_slew
```

For more information about the nominal retain arc slew groups, see [Chapter 7, Timing Arcs](#).

sigma_type Attribute

The optional `sigma_type` attribute defines the type of arrival time specified in the `ocv_sigma_retaining_rise`, `ocv_sigma_retaining_fall`, `ocv_sigma_retain_rise_slew`, and `ocv_sigma_retain_fall_slew` lookup tables. The values are `early`, `late`, and `early_and_late`. The default is `early_and_late`.

During OCV analysis, if the `sigma_type` attribute is specified in these groups, the following values at $\pm\sigma$ are calculated as:

```
retaining_rise( $+\sigma$ ) = retaining_rise + ocv_sigma_retaining_riselate  
retaining_rise( $-\sigma$ ) = retaining_rise - ocv_sigma_retaining_riseearly  
retaining_fall( $+\sigma$ ) = retaining_fall + ocv_sigma_retaining_falllate  
retaining_fall( $-\sigma$ ) = retaining_fall - ocv_sigma_retaining_fallearly  
  
retain_rise_slew( $+\sigma$ ) = retain_rise_slew + ocv_sigma_retain_rise_slewlate  
retain_rise_slew( $-\sigma$ ) = retain_rise_slew - ocv_sigma_retain_rise_slewearly  
retain_fall_slew( $+\sigma$ ) = retain_fall_slew + ocv_sigma_retain_fall_slewlate  
retain_fall_slew( $-\sigma$ ) = retain_fall_slew - ocv_sigma_retain_fall_slewearly
```

Library Example

The following example is a library with LVF retain arc models.

```
library (lib1) {  
    lu_table_template(2D_lu_template) {  
        variable_1: input_net_transition;
```

```
variable_2: total_output_net_capacitance;
index_1 (1,2,3);
index_2 (1,2,3);
}
lu_table_template(1D_lu_template) {
    variable_1: total_output_net_capacitance;
    index_1 (1,2,3);
}
cell (cell1) {

    pin(pin1) {
        direction: output;
        timing() {

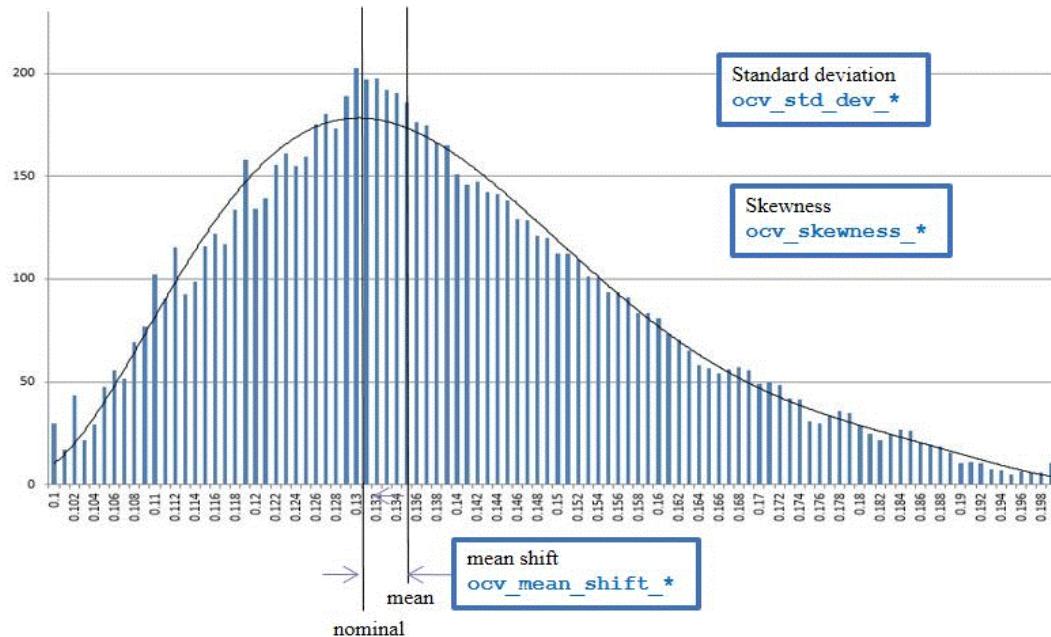
            related_pin : "CLKIN" ;
            timing_type : rising_edge ;
            cell_rise( 2D_lu_template ){
                ...
            }
            cell_fall( 2D_lu_template ){
                ...
            }
            rise_transition( 2D_lu_template ){
                ...
            }
            fall_transition( 2D_lu_template ){
                ...
            }
            retaining_rise( 2D_lu_template ){
                index_1( "0.10000, 0.20000, 0.30000");
                index_2( "0.40000, 0.60000, 0.60000");
                values ( "0.10000, 0.20000, 0.30000", \
                          "0.40000, 0.50000, 0.60000", \
                          "0.70000, 0.80000, 0.90000");
            }
            retaining_fall ( 1D_lu_template ){
                values ( "0.10000, 0.20000, 0.30000");
            }
            retain_rise_slew( 2D_lu_template ){
                index_1 ( "0.10000, 0.20000, 0.30000");
                index_2 ( "0.40000, 0.60000, 0.60000");
                values ( "0.10000, 0.20000, 0.30000", \
                          "0.40000, 0.50000, 0.60000", \
                          "0.70000, 0.80000, 0.90000");
            }
            retain_fall_slew( 1D_lu_template ){
                values ( "0.10000, 0.20000, 0.30000");
            }
            ocv_sigma_retaining_rise( 2D_lu_template ){
                sigma_type: early;
                index_1 ( "0.10000, 0.20000, 0.30000");
                index_2 ( "0.40000, 0.50000, 0.60000");
                values ( "0.10000, 0.20000, 0.30000", \
                          "0.40000, 0.50000, 0.60000", \
                          "0.70000, 0.80000, 0.90000");
            }
        }
    }
}
```

```
        "0.40000, 0.50000, 0.60000",\
        "0.70000, 0.80000, 0.90000");
    }
    ocv_sigma_retaining_rise( 2D_lu_template ){
        sigma_type: late;
        index_1 ( "0.10000, 0.20000, 0.30000");
        index_2 ( "0.40000, 0.50000, 0.60000");
        values ( "0.10000, 0.20000, 0.30000",\
                  "0.40000, 0.50000, 0.60000",\
                  "0.70000, 0.80000, 0.90000");
    }
    ocv_sigma_retaining_fall ( 1D_lu_template ){
        sigma_type: early;
        values ( "0.10000, 0.20000, 0.30000");
    }
    ocv_sigma_retaining_fall ( 1D_lu_template ){
        sigma_type: late;
        values ( "0.10000, 0.20000, 0.30000");
    }
    ocv_sigma_retain_rise_slew( 2D_lu_template ){
        sigma_type: early_and_late;
        index_1 ( "0.10000, 0.20000, 0.30000");
        index_2 ( "0.40000, 0.50000, 0.60000");
        values ( "0.10000, 0.20000, 0.30000",\
                  "0.40000, 0.50000, 0.60000",\
                  "0.70000, 0.80000, 0.90000");
    }
    ocv_sigma_retain_fall_slew ( 1D_lu_template ){
        values ( "0.10000, 0.20000, 0.30000");
    }
} /* end of timing group */
} /* end of pin group */
} /* end of cell group */
} /* end of library group */
```

LVF Moment-Based Models For Ultra-Low Voltage Designs

Accurate models of ultra-low voltage libraries might require support for asymmetric, biased, or non-Gaussian distributions of timing variation. To capture the shape of the biased timing variation distribution, moments-based Liberty variation format (LVF) OCV models are supported.

Figure 116 Biased Timing Variation Distribution With Positive Skew and Shifted Mean



As shown in [Figure 116](#), the moment-based LVF syntax is based on the mean, standard deviation, and skewness of the timing variation distribution.

Mean

Mean is the weighted average of the possible values, using their probabilities as weights or the continuous analog thereof. The mean of a random variable X is defined, as shown:

$$E[X] = x_1 p_1 + x_2 p_2 + \dots + x_k p_k$$

where X can take values x_1 with probability p_1 , x_2 with probability p_2 , up to x_k with probability p_k .

Standard Deviation

Standard deviation is a measure of the variation or dispersion of the distribution. It is defined as the square root of the variance, as shown:

$$(E[(X - E[X])^2])^{1/2}$$

Skewness

Skewness is a measure of the asymmetry of the distribution of a random variable X about its mean. The definition follows the Pearson's coefficient of skewness. It is defined, as shown:

$$(E[(X-E[X])^3])^{1/3}$$

The following sections describe the moment-based LVF OCV models of random variation in cell delay, transition, and constraints.

Syntax

The following shows the moment-based LVF OCV modeling syntax.

```
library (lib_name) {
    lu_table_template (lu_template_name) {
        variable_1: input_net_transition;
        variable_2: total_output_net_capacitance;
        variable_3: related_out_total_output_net_capacitance;
        index_1 ("float,..., float");
        index_2 ("float,..., float");
        index_3 ("float,..., float");
    }
    ...
    cell (cell_name) {
        ...
        pin | bus | bundle (name) {
            direction: inout | output;
            timing() {
                ...
                /* delay nominal values */
                cell_rise (lu_template_name) {
                    ...
                }
                cell_fall (lu_template_name) {
                    ...
                }
                /* delay variation values */
                ocv_std_dev_cell_rise(lu_template_name) {
                    index_1 ("float,..., float");
                    index_2 ("float,..., float");
                    index_3 ("float,..., float");
                    values ("float,..., float", \
                            ...,\n                            "float,..., float");
                }
                ocv_std_dev_cell_fall(lu_template_name) {
                    index_1 ("float,..., float");
                }
            }
        }
    }
}
```

Chapter 15: On-Chip Variation (OCV) Modeling
LVF Moment-Based Models For Ultra-Low Voltage Designs

```
index_2 ("float,..., float");
index_3 ("float,..., float");
values  ("float,..., float", \
         ...,\ 
           "float,..., float");
}
ocv_std_dev_retaining_rise(lu_template_name) {
    index_1 ("float,..., float");
    index_2 ("float,..., float");
    index_3 ("float,..., float");
    values  ("float,..., float", \
             ...,\ 
               "float,..., float");
}
ocv_std_dev_retaining_fall(lu_template_name) {
    index_1 ("float,..., float");
    index_2 ("float,..., float");
    index_3 ("float,..., float");
    values  ("float,..., float", \
             ...,\ 
               "float,..., float");
}
ocv_mean_shift_cell_rise(lu_template_name) {
    index_1 ("float,..., float");
    index_2 ("float,..., float");
    index_3 ("float,..., float");
    values  ("float,..., float", \
             ...,\ 
               "float,..., float");
}
ocv_mean_shift_cell_fall(lu_template_name) {
    index_1 ("float,..., float");
    index_2 ("float,..., float");
    index_3 ("float,..., float");
    values  ("float,..., float", \
             ...,\ 
               "float,..., float");
}
ocv_mean_shift_retaining_rise(lu_template_name) {
    index_1 ("float,..., float");
    index_2 ("float,..., float");
    index_3 ("float,..., float");
    values  ("float,..., float", \
             ...,\ 
               "float,..., float");
}
ocv_mean_shift_retaining_fall(lu_template_name) {
    index_1 ("float,..., float");
    index_2 ("float,..., float");
    index_3 ("float,..., float");
    values  ("float,..., float", \
             ...,\ 
               "float,..., float");
}
```

```
        "float,..., float");
}
ocv_skewness_cell_rise(lu_template_name) {
    index_1 ("float,..., float");
    index_2 ("float,..., float");
    index_3 ("float,..., float");
    values ( "float,..., float", \
             ...,\n
             "float,..., float");
}
ocv_skewness_cell_fall(lu_template_name) {
    index_1 ("float,..., float");
    index_2 ("float,..., float");
    index_3 ("float,..., float");
    values ( "float,..., float", \
             ...,\n
             "float,..., float");
}
ocv_skewness_retaining_rise(lu_template_name) {
    index_1 ("float,..., float");
    index_2 ("float,..., float");
    index_3 ("float,..., float");
    values ( "float,..., float", \
             ...,\n
             "float,..., float");
}
ocv_skewness_retaining_fall(lu_template_name) {
    index_1 ("float,..., float");
    index_2 ("float,..., float");
    index_3 ("float,..., float");
    values ( "float,..., float", \
             ...,\n
             "float,..., float");
}

/* transition nominal values */
rise_transition (lu_template_name) {
    ...
}
fall_transition (lu_template_name) {
    ...
}
/* transition variation values */
ocv_std_dev_rise_transition(lu_template_name) {
    ...
}
ocv_std_dev_fall_transition(lu_template_name) {
    ...
}
ocv_std_dev_retain_rise_slew(lu_template_name) {
    ...
}
ocv_std_dev_retain_fall_slew(lu_template_name) {
```

```
    ...
}

ocv_mean_shift_rise_transition(lu_template_name) {
    ...
}

ocv_mean_shift_fall_transition(lu_template_name) {
    ...
}

ocv_mean_shift_retain_rise_slew(lu_template_name) {
    ...
}

ocv_mean_shift_retain_fall_slew(lu_template_name) {
    ...
}

ocv_skewness_rise_transition(lu_template_name) {
    ...
}

ocv_skewness_fall_transition(lu_template_name) {
    ...
}

ocv_skewness_retain_rise_slew(lu_template_name) {
    ...
}

ocv_skewness_retain_fall_slew(lu_template_name) {
    ...
}

} /* end of timing */
} /* end of inout | output pin */

...
pin | bus | bundle (name) {
    direction: inout | input;
    timing() {

        ...
        /* constraint nominal values */
        rise_constraint(lu_template_name) {
            ...
        }

        fall_constraint(lu_template_name) {
            ...
        }

        /* constraint variation values */
        ocv_std_dev_rise_constraint(lu_template_name) {
            ...
        }

        ocv_std_dev_fall_constraint(lu_template_name) {
            ...
        }

        ocv_mean_shift_rise_constraint(lu_template_name) {
            ...
        }

        ocv_mean_shift_fall_constraint(lu_template_name) {
            ...
        }
    }
}
```

```
ocv_skewness_rise_constraint(lu_template_name) {  
    ...  
}  
ocv_skewness_fall_constraint(lu_template_name) {  
    ...  
}  
...  
} /* end of timing */  
} /* end of inout | input pin */  
...  
} /* end of cell */  
...  
} /* end of library */
```

Library-Level Groups

This section describes library-level groups to model asymmetric timing variation distributions of delays, transitions, and constraints.

lu_table_template Group

The `lu_table_template` group defines the template for the following timing-level groups:

- `ocv_std_dev_*`
- `ocv_mean_shift_*`
- `ocv_skewness_*`

The template definition is same as that of the [lu_table_template Group on page 715](#) described in “[LVF Retain Arc Models](#).”

Timing Arc Level Groups

This section describes timing-level groups to model asymmetric timing variation distributions of delays, transitions, and constraints. These groups are lookup tables that can be scalar, one, two, or three-dimensional.

Define these groups in the `timing` group together with the nominal retain arc tables.

These groups are defined under the `timing` group together with the nominal tables.

ocv_std_dev_* Groups

The `ocv_std_dev_cell_rise`, `ocv_std_dev_cell_fall`,
`ocv_std_dev_rise_transition`, `ocv_std_dev_fall_transition`,
`ocv_std_dev_retaining_rise`, `ocv_std_dev_retaining_fall`,
`ocv_std_dev_retain_rise_slew`, `ocv_std_dev_retain_fall_slew`,

`ocv_std_dev_rise_constraint`, and `ocv_std_dev_fall_constraint` look up tables store the values of standard deviation of the timing variation distribution.

`ocv_mean_shift_*` Groups

The `ocv_mean_shift_cell_rise`, `ocv_mean_shift_cell_fall`,
`ocv_mean_shift_rise_transition`, `ocv_mean_shift_fall_transition`,
`ocv_mean_shift_retaining_rise`, `ocv_mean_shift_retaining_fall`,
`ocv_mean_shift_retain_rise_slew`, `ocv_mean_shift_retain_fall_slew`,
`ocv_mean_shift_rise_constraint`, and `ocv_mean_shift_fall_constraint` lookup tables specify the offset value from the mean of the timing variation distribution to the nominal value.

The mean values of the timing variation distribution are calculated as:

```
mean rise delay =      nominal rise delay +  
                      ocv_mean_shift_cell_rise  
  
mean fall delay =      nominal fall delay +  
                      ocv_mean_shift_cell_fall  
  
mean rise transition = nominal rise transition +  
                      ocv_mean_shift_rise_transition  
  
mean fall transition = nominal fall transition +  
                      ocv_mean_shift_fall_transition  
  
mean retaining rise =  nominal retaining rise +  
                      ocv_mean_shift_retaining_rise  
  
mean retaining fall =  nominal retaining fall +  
                      ocv_mean_shift_retaining_fall  
  
mean retain rise slew = nominal retain rise slew +  
                      ocv_mean_shift_retain_rise_slew  
  
mean retain fall skew = nominal retain fall slew +  
                      ocv_mean_shift_retain_fall_slew  
  
mean rise constraint = nominal rise constraint +  
                      ocv_mean_shift_rise_constraint  
  
mean fall constraint = nominal fall constraint +  
                      ocv_mean_shift_fall_constraint
```

`ocv_skewness_*` Groups

These `ocv_skewness_cell_rise`, `ocv_skewness_cell_fall`,
`ocv_skewness_rise_transition`, `ocv_skewness_fall_transition`,
`ocv_skewness_retaining_rise`, `ocv_skewness_retaining_fall`,
`ocv_skewness_retain_rise_slew`, `ocv_skewness_retain_fall_slew`,

`ocv_skewness_rise_constraint`, `ocv_skewness_fall_constraint` **lookup tables** specify the skewness values of the timing variation distribution.

Library Example

The following example shows a library with moment-based LVF models of a biased timing variation distribution.

```
library (lib1) {
    lu_table_template(2D_lu_template) {
        variable_1: input_net_transition;
        variable_2: total_output_net_capacitance;
        index_1 ("1,2,3");
        index_2 ("1,2,3");
    }
    cell (cell1) {
        ...
        pin(pin1) {
            direction: output;
            timing() {
                ...
                related_pin : "CLKIN" ;
                timing_type : rising_edge ;

                cell_rise( 2D_lu_template ){
                    index_1 ( "0.3, 0.4, 0.6");
                    index_2 ( "0.01, 0.02, 0.03");
                    values ( "0.04, 0.05, 0.06",\
                            "0.07, 0.08, 0.09",\
                            "0.1, 0.2, 0.3");
                }
                cell_fall ( scalar ){
                    values ("0.095");
                }
                rise_transition( 2D_lu_template ){
                    index_1 ( "0.3, 0.4, 0.6");
                    index_2 ( "0.01, 0.02, 0.03");
                    values ( "0.04, 0.05, 0.06",\
                            "0.07, 0.08, 0.09",\
                            "0.1, 0.2, 0.3");
                }
                fall_transition ( scalar ){
                    values ("0.095");
                }
                ocv_std_dev_cell_rise( 2D_lu_template ){
                    index_1 ( "0.3, 0.4, 0.6");
                    index_2 ( "0.01, 0.02, 0.03");
                    values ( "0.04, 0.05, 0.06",\
                            "0.07, 0.08, 0.09",\
                            "0.1, 0.2, 0.3");
                }
            }
        }
    }
}
```

Chapter 15: On-Chip Variation (OCV) Modeling
LVF Moment-Based Models For Ultra-Low Voltage Designs

```
ocv_std_dev_cell_fall ( scalar ){
    values ("0.005");
}
ocv_std_dev_rise_transition( 2D_lu_template ){
    index_1 ("0.3, 0.4, 0.6");
    index_2 ("0.01, 0.02, 0.03");
    values ( "0.04, 0.05, 0.06",\
              "0.07, 0.08, 0.09",\
              "0.1, 0.2, 0.3");
}
ocv_std_dev_fall_transition ( scalar ){
    values ("0.005");
}
ocv_std_dev_retaining_rise( 2D_lu_template ){
    index_1 ("0.3, 0.4, 0.6");
    index_2 ("0.01, 0.02, 0.03");
    values ( "0.04, 0.05, 0.06",\
              "0.07, 0.08, 0.09",\
              "0.1, 0.2, 0.3");
}
ocv_std_dev_retaining_fall ( scalar ){
    values ("0.005");
}
ocv_std_dev_retain_rise_slew( 2D_lu_template ){
    index_1 ("0.3, 0.4, 0.6");
    index_2 ("0.01, 0.02, 0.03");
    values ( "0.04, 0.05, 0.06",\
              "0.07, 0.08, 0.09",\
              "0.1, 0.2, 0.3");
}
ocv_std_dev_retain_fall_slew ( scalar ){
    values ("0.005");
}
ocv_mean_shift_cell_rise( 2D_lu_template ){
    index_1 ("0.3, 0.4, 0.6");
    index_2 ("0.01, 0.02, 0.03");
    values ( "0.04, 0.05, 0.06",\
              "0.07, 0.08, 0.09",\
              "0.1, 0.2, 0.3");
}
ocv_mean_shift_cell_fall( scalar ){
    values ("0.015");
}
ocv_mean_shift_rise_transition( 2D_lu_template ){
    index_1 ("0.3, 0.4, 0.6");
    index_2 ("0.01, 0.02, 0.03");
    values ( "0.04, 0.05, 0.06",\
              "0.07, 0.08, 0.09",\
              "0.1, 0.2, 0.3");
}
ocv_mean_shift_fall_transition ( scalar ){
    values ("0.005");
}
```

Chapter 15: On-Chip Variation (OCV) Modeling
LVF Moment-Based Models For Ultra-Low Voltage Designs

```
ocv_mean_shift_retaining_rise( 2D_lu_template ){
    index_1 ( "0.3, 0.4, 0.6");
    index_2 ( "0.01, 0.02, 0.03");
    values ( "0.04, 0.05, 0.06",\
              "0.07, 0.08, 0.09",\
              "0.1, 0.2, 0.3");
}
ocv_mean_shift_retaining_fall ( scalar ){
    values ("0.005");
}
ocv_mean_shift_retain_rise_slew( 2D_lu_template ){
    index_1 ( "0.3, 0.4, 0.6");
    index_2 ( "0.01, 0.02, 0.03");
    values ( "0.04, 0.05, 0.06",\
              "0.07, 0.08, 0.09",\
              "0.1, 0.2, 0.3");
}
ocv_mean_shift_retain_fall_slew ( scalar ){
    values ("0.005");
}
ocv_skewness_cell_rise( 2D_lu_template ){
    index_1 ( "0.3, 0.4, 0.6");
    index_2 ( "0.01, 0.02, 0.03");
    values ( "0.04, 0.05, 0.06",\
              "0.07, 0.08, 0.09",\
              "0.1, 0.2, 0.3");
}
ocv_skewness_cell_fall( scalar ){
    values ("0.015");
}
ocv_skewness_rise_transition( 2D_lu_template ){
    index_1 ( "0.3, 0.4, 0.6");
    index_2 ( "0.01, 0.02, 0.03");
    values ( "0.04, 0.05, 0.06",\
              "0.07, 0.08, 0.09",\
              "0.1, 0.2, 0.3");
}
ocv_skewness_fall_transition( scalar ){
    values ("0.015");
}
ocv_skewness_retaining_rise( 2D_lu_template ){
    index_1 ( "0.3, 0.4, 0.6");
    index_2 ( "0.01, 0.02, 0.03");
    values ( "0.04, 0.05, 0.06",\
              "0.07, 0.08, 0.09",\
              "0.1, 0.2, 0.3");
}
ocv_skewness_retaining_fall ( scalar ){
    values ("0.005");
}
ocv_skewness_retain_rise_slew( 2D_lu_template ){
    index_1 ( "0.3, 0.4, 0.6");
    index_2 ( "0.01, 0.02, 0.03");
```

Chapter 15: On-Chip Variation (OCV) Modeling
LVF Moment-Based Models For Ultra-Low Voltage Designs

```
values ( "0.04, 0.05, 0.06",\
         "0.07, 0.08, 0.09",\
         "0.1, 0.2, 0.3");
}
ocv_skewness_retain_fall_slew ( scalar ){
    values ("0.005");
}
...
} /* end of timing */
...
} /* end of pin */
...
} /* end of cell */
...
} /* end of library */
```

16

Defining I/O Pads

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

To model I/O pads, you must understand the following concepts covered in this chapter:

- [Special Characteristics of I/O Pads](#)
 - [Identifying Pad Cells](#)
 - [Defining Units for Pad Cells](#)
 - [Describing Input Pads](#)
 - [Describing Output Pads](#)
 - [Modeling Wire Load for Pads](#)
 - [Programmable Driver Type Support in I/O Pad Cell Models](#)
 - [Pad Cell Examples](#)
-

Special Characteristics of I/O Pads

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

Pads typically have longer delays and higher drive capabilities than the cells in an integrated circuit's core. Because of their higher drive, CMOS pads sometimes exhibit noise problems. Slew-rate control is available on output pads to help alleviate this problem.

Pads are fixed resources—a limited number are available on each die size. In addition, special types, such as clock pads, might be more limited than others. These limits must be modeled.

A distinguishing feature of pad cells is the voltage level at which input pads transfer logic 0 or logic 1 signals to the core or at which output pad drivers communicate logic values from the core.

Integrated circuits that communicate with one another must have compatible voltage levels at their pads. Because pads communicate with the world outside the integrated circuit, you must describe the pertinent units of peripheral library properties, such as external load, drive capability, delay, current, power, and resistance. This description makes it easier to design chips from multiple technologies.

Some libraries create logical pads out of multiple cells; such logical pads must be modeled so that they are just as easy to insert automatically on a design as a single-cell pad.

You must capture all these properties in the library to make it possible for the integrated circuit designer to insert the correct pads during synthesis.

Identifying Pad Cells

Use the attributes described in the following sections to specify I/O pads and pad pin behaviors.

is_pad Attribute

After you identify a cell as a pad cell, you must indicate which pin represents the pad. The `is_pad` simple attribute must be used on at least one pin of a cell with a `pad_cell` attribute.

The `direction` attribute indicates whether the pad is an input, output, or bidirectional pad.

Syntax

```
is_pad : true | false ;
```

Example

```
cell(INBUF) {
  ...
  pin(PAD) {
    direction : input ;
    is_pad : true ;
  }
}
```

driver_type Attribute

A `driver_type` attribute defines two types of signal modifications: transformation and resolution.

- Transformation specifies an actual signal transition from 0/1 to L/H/Z. This signal transition performs a function on an input signal and requires only a straightforward mapping.

- Resolution resolves the value Z on an existing circuit node without actually performing a function and implies a constant (0/1) signal source as part of the resolution.

Syntax

```
driver_type : pull_up | pull_down | open_drain | open_source |
bus_hold
| resistive | resistive_0 | resistive_1 ;
pull_up
```

The pin is connected to power through a resistor. If it is a three-state output pin, it is in the Z state and its function is evaluated as a resistive 1 (H). If it is an input or inout pin and the node to which it is connected is in the Z state, it is considered an input pin at logic 1 (H). For a `pull_up` cell, the pin constantly stays at logic 1 (H).

`pull_down`

The pin is connected to ground through a resistor. If it is a three-state output pin, it is in the Z state and its function is evaluated as a resistive 0 (L). If it is an input or inout pin and the node to which it is connected is in the Z state, it is considered an input pin at logic 0 (L). For a `pull_down` cell, the pin constantly stays at logic 0 (L).

`open_drain`

The pin is an output pin without a pull-up transistor. Use this driver type only for off-chip output or inout pins representing pads. The pin goes to high impedance (Z) when its function is evaluated as logic 1.

Note:

An n-channel open-drain pad is flagged with `open_drain`, and a p-channel open-drain pad is flagged with `open_source`.

`open_source`

The pin is an output pin without a pull-down transistor. Use this driver type only for off-chip output or inout pins representing pads. The pin goes to high impedance (Z) when its function is evaluated as logic 0.

`bus_hold`

The pin is a bidirectional pin on a bus holder cell. The pin holds the last logic value present at that pin when no other active drivers are on the associated net. Pins with this driver type cannot have `function` or `three_state` attributes.

`resistive`

The pin is an output pin connected to a controlled pull-up or pull-down resistor with a control port EN. When EN is disabled, the pull-up or pull-down resistor is

turned off and has no effect on the pin. When EN is enabled, a functional value of 0 evaluated at the pin is turned into a weak 0, a functional value of 1 is turned into a weak 1, but a functional value of Z is not affected.

`resistive_0`

The pin is an output pin connected to power through a pull-up resistor that has a control port EN. When EN is disabled, the pull-up resistor is turned off and has no effect on the pin. When EN is enabled, a functional value of 1 evaluated at the pin turns into a weak 1, but a functional value of 0 or Z is not affected.

`resistive_1`

The pin is an output pin connected to ground through a pull-down resistor that has a control port EN. When EN is disabled, the pull-down resistor is turned off and has no effect on the pin. When EN is enabled, a functional value of 0 evaluated at the pin turns into a weak 0, but a functional value of 1 or Z is not affected.

[Table 35](#) lists the driver types, their signal mappings, and the applicable pin types.

Table 35 Driver Types

Driver type	Description	Signal mapping	Applicable pin types
<code>pull_up</code>	Resolution	01Z -> 01H	in, out
<code>pull_down</code>	Resolution	01Z -> 01L	in, out
<code>bus_hold</code>	Resolution	01Z -> 01S	inout
<code>open_drain</code>	Transformation	01Z -> 0ZZ	out
<code>open_source</code>	Transformation	01Z -> Z1Z	out
<code>resistive</code>	Transformation	01Z -> LHZ	out
<code>resistive_0</code>	Transformation	01Z -> 0HZ	out
<code>resistive_1</code>	Transformation	01Z -> L1Z	out

In [Table 35](#), the `pull_up`, `pull_down`, and `bus_hold` driver types define a resolution scheme. The remaining driver types define transformations.

The following example describes an output pin with a pull-up resistor and the bidirectional pin on a `bus_hold` cell.

Example

```
cell (bus) {  
    pin(Y) {  
        direction : output ;  
        driver_type : pull_up ;  
        pulling_resistance : 10000 ;  
        function : "IO" ;  
        three_state : "OE" ;  
    }  
}  
cell (bus_hold) {  
    pin(Y) {  
        direction : inout ;  
        driver_type : bus_hold ;  
    }  
}
```

Defining Units for Pad Cells

To process pads for full-chip synthesis, specify the units of time, capacitance, resistance, voltage, current, and power:

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

All these attributes are defined at the library level, as described in [Input-Capacitance Characterization Attributes on page 764](#). These values are required.

Capacitance

The capacitive_load_unit attribute defines the capacitance associated with a standard load. If you already represent capacitance values in terms of picofarads or femtofarads, use this attribute to define your base unit. If you represent capacitance in terms of the standard load of an inverter, define the exact capacitance for that inverter—for example, 0.101 pF.

Example

```
capacitive_load_unit( 0.1,ff );
```

Resistance

You must supply a `pulling_resistance` attribute for pull-up and pull-down devices on pads and identify the unit to use with the `pulling_resistance_unit` attribute.

Example

```
pulling_resistance_unit : "1kohm";
```

Voltage

You can use the `input_voltage` and `output_voltage` groups to define a set of input or output voltage ranges for your pads. To define the units of voltage you use for these groups, use the `voltage_unit` attribute. All the attributes defined inside `input_voltage` and `output_voltage` groups are scaled by the value defined for `voltage_unit`. In addition, the `voltage` attribute in the `operating_conditions` groups also represents its values in these units.

Example

```
voltage_unit : "1V";
```

Current

You can define the drive current that can be generated by an output pad and also define the pulling current for a pull-up or pull-down transistor under nominal voltage conditions. Define all current values with the library-level `current_unit` attribute.

Example

```
current_unit : "1uA";
```

Describing Input Pads

To represent input pads in your logic library, you must describe the input voltage characteristics and indicate whether hysteresis applies.

The input pad properties are described in the next section. Examples at the end of this chapter describe a standard input buffer, an input buffer with hysteresis, and an input clock buffer.

hysteresis Attribute

Specify the `hysteresis` attribute for an input pad for a long transition time or when the pad is driven by a noisy line.

The default is `false`. When `true`, the `vil` and `vol` voltage ratings are actual transition points.

Example

```
hysteresis : true;
```

Pads with hysteresis can have derating factors that are different from the core cells. Use the `scaled_cell` group to describe the timing of cells with hysteresis. This construct provides derating capabilities and minimum, typical, or maximum timing for cells.

Describing Output Pads

To represent output pads in your logic library, you must describe the output voltage characteristics and the drive-current rating of output and bidirectional pads. Additionally, you must include information about the slew rate of the pad. These output pad properties are described in the sections that follow.

Examples at the end of this chapter show a standard output buffer and a bidirectional pad.

Drive Current

Output and bidirectional pads in a technology can have different drive-current capabilities. To define the drive current supplied by the pad buffer, use the `drive_current` attribute on an output or bidirectional pad or auxiliary pad pin. The value is in units consistent with the `current_unit` attribute you defined.

Example

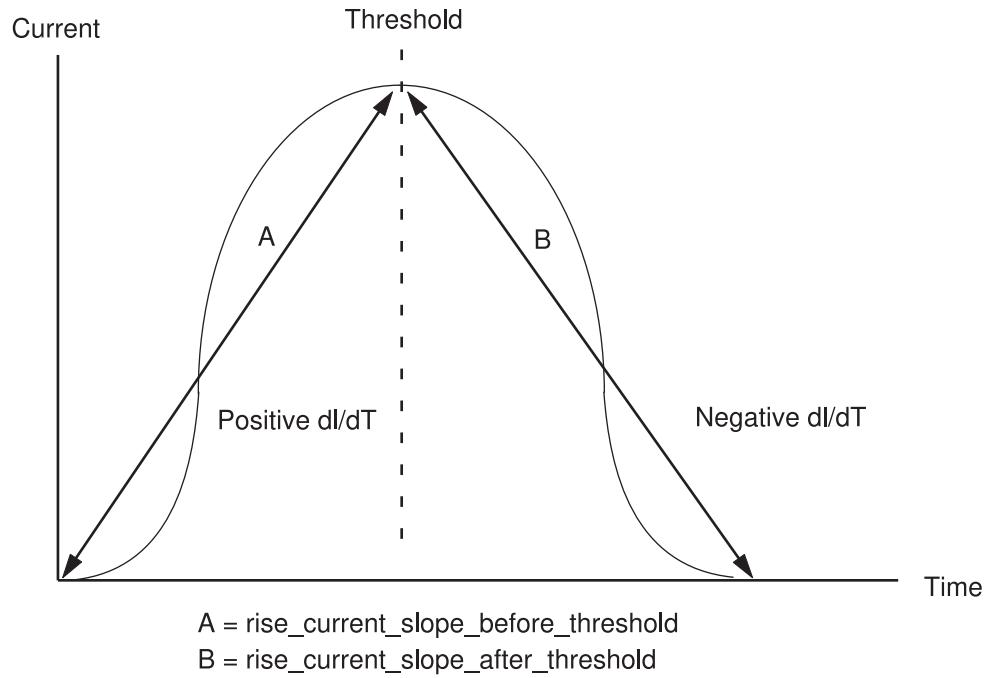
```
pin(PAD) {  
    direction : output;  
    is_pad : true;  
    drive_current : 1.0;  
}
```

Slew-Rate Control

The `slew_control` attribute accepts one of four possible enumerations: none, low, medium, and high; the default is none. Increasing the slew control level slows down the transition rate. This method is the coarsest way to measure the level of slew-rate control associated with an output pad.

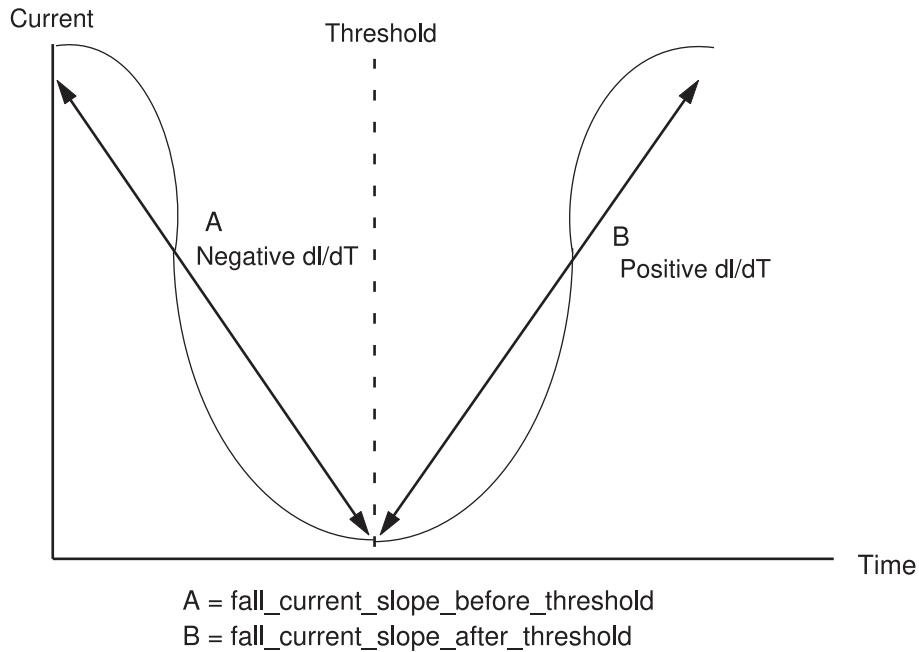
In [Figure 117](#), A is a positive number and a linear approximation of the change in current as a function of time from the beginning of the rising transition to the threshold. B is a negative number and a linear approximation of the current change over time from the threshold to the end of transition.

Figure 117 Slew-Rate Attributes—Rising Transitions



For falling transitions, the graph is reversed and A becomes negative and B positive as shown in [Figure 118](#).

Figure 118 Slew-Rate Attributes—Falling Transitions



Example 140 Slew-Rate Control Attributes

```
pin(PAD) {  
    is_pad : true;  
    direction : output;  
    output_voltage : GENERAL;  
    slew_control : high;  
    rise_current_slope_before_threshold : 0.18;  
    rise_time_before_threshold : 0.8;  
    rise_current_slope_after_threshold : -0.09;  
    rise_time_after_threshold : 2.4;  
    fall_current_slope_before_threshold : -0.14;  
    fall_time_before_threshold : 0.55;  
    fall_current_slope_after_threshold : 0.07;  
    fall_time_after_threshold : 1.8;  
    ...  
}
```

Modeling Wire Load for Pads

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

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

You can define the `wire_load` group, which you want to use for wire load estimation, on the pad ring. Add a level of hierarchy by placing the pads in the top level and by placing all the core circuitry at a lower level.

```
dc_shell> set_wire_load Pad_WireLoad
```

A different model is defined for the core hierarchical level.

Programmable Driver Type Support in I/O Pad Cell Models

To support pull-up and pull-down circuit structures, the Liberty models for I/O pad cells support pull-up and pull-down driver information using the `driver_type` attribute with the `pull_up` or `pull_down` values.

Liberty syntax also supports conditional (programmable) pull-up and pull-down driver information for I/O pad cells. The programmable pin syntax has also been extended to other `driver_type` attribute values, such as `bus_hold`, `open_drain`, `open_source`, `resistive`, `resistive_0`, and `resistive_1`.

Syntax

The following syntax supports programmable driver types in I/O pad cell models. Unlike the nonprogrammable driver type support, the programmable driver type support allows you to specify more than one driver type within a pin.

```
pin (pin_name) { /* programmable driver type pin */
    ...
    pull_up_function : "function string";
    pull_down_function : "function string";
    bus_hold_function : "function string";
    open_drain_function : "function string";
    open_source_function : "function string";
    resistive_function : "function string";
    resistive_0_function : "function string";
    resistive_1_function : "function string";
    ...
}
```

Programmable Driver Type Functions

The driver type function attributes in [Table 36](#) help model the programmable driver types. The same rules that apply to nonprogrammable driver types also apply to these functions.

Table 36 Programmable Driver Type Functions

Programmable driver type	Applicable on pin types
pull_up_function	Input, output and inout
pull_down_function	Input, output and inout
bus_hold_function	Inout
open_drain_function	Output and inout
open_source_function	Output and inout
resistive_function	Output and inout
resistive_0_function	Output and inout
resistive_1_function	Output and inout

With the exception of `pull_up_function` and `pull_down_function`, if any of the driver type functions in [Table 36](#) is specified on an inout pin, it is used only for output pins.

The following rules apply to programmable driver type functions (as well as nonprogrammable driver types in I/O pad cell models):

- The attribute can be applied to pad cell only.
- Only the input and inout pin can be specified in the function string.
- The function string is a Boolean function of input pins.

The following rules apply to an inout pin:

- If `pull_up_function` or `pull_down_function` and `open_drain_function` are specified within the same inout pin, `pull_up_function` or `pull_down_function` is used for the input pins.
- If `bus_hold_function` is specified on an inout pin, it is used for input and output pins.

Example

Example 141 Model of a Programmable Driver Type in an I/O Pad Cell

```
library(cond_pull_updown_example) {
delay_model : table_lookup;

time_unit : 1ns;
voltage_unit : 1V;
capacitive_load_unit (1.0, pf);
current_unit : 1mA;
```

Chapter 16: Defining I/O Pads
Programmable Driver Type Support in I/O Pad Cell Models

```
cell(conditional_PU_PD) {
    dont_touch : true ;
    dont_use : true ;
    pad_cell : true ;
    pin(IO) {
        drive_current : 1 ;
        min_capacitance : 0.001 ;
        min_transition : 0.0008 ;
        is_pad : true ;
        direction : inout ;
        max_capacitance : 30 ;
        max_fanout : 2644 ;
        function : "(A*ETM')+(TA*ETM)" ;
        three_state : "(TEN*ETM')+(EN*ETM)" ;
        pull_up_function : "(!P1 * !P2)" ;
        pull_down_function : "( P1 * P2)" ;
        capacitance : 2.06649 ;
        timing() {
            related_pin : "IO A ETM TEN TA" ;
            cell_rise(scalar) {
                values("0" ) ;
            }
            rise_transition(scalar) {
                values("0" ) ;
            }
            cell_fall(scalar) {
                values("0" ) ;
            }
            fall_transition(scalar) {
                values("0" ) ;
            }
        }
        timing() {
            timing_type : three_state_disable;
            related_pin : "EN ETM TEN" ;
            cell_rise(scalar) {
                values("0" ) ;
            }
            rise_transition(scalar) {
                values("0" ) ;
            }
            cell_fall(scalar) {
                values("0" ) ;
            }
            fall_transition(scalar) {
                values("0" ) ;
            }
        }
    }
    pin(ZI) {
        direction : output;
```

Chapter 16: Defining I/O Pads Pad Cell Examples

```
function      : "IO" ;
timing() {
    related_pin : "IO" ;
    cell_rise(scalar) {
        values("0" ) ;
    }
    rise_transition(scalar) {
        values("0" ) ;
    }
    cell_fall(scalar) {
        values("0" ) ;
    }
    fall_transition(scalar) {
        values("0" ) ;
    }
}
pin(A) {
    direction : input;
    capacitance : 1.0;
}
pin(EN) {
    direction : input;
    capacitance : 1.0;
}
pin(TA) {
    direction : input;
    capacitance : 1.0;
}
pin(TEN) {
    direction : input;
    capacitance : 1.0;
}
pin(ETM) {
    direction : input;
    capacitance : 1.0;
}
pin(P1) {
    direction : input;
    capacitance : 1.0;
}
pin(P2) {
    direction : input;
    capacitance : 1.0;
}
} /* End cell conditional_PU_PD */
} /* End Library */
```

Pad Cell Examples

These are examples of input, clock, output, and bidirectional pad cells.

Input Pads

Example 142 Standard Input Buffer

```
library (example1) {
    date : "August 14, 2015" ;
    revision : 2015.05;
    ...
    time_unit : "1ns";
    voltage_unit : "1V";
    current_unit : "1uA";
    pulling_resistance_unit : "1kohm";
    capacitive_load_unit( 0.1,ff );
    ...
    define_cell_area(bond_pads,pad_slots);
    define_cell_area(driver_sites,pad_driver_sites);
    ...
    input_voltage(CMOS) {
        vil : 1.5;
        vih : 3.5;
        vimin : -0.3;
        vimax : VDD + 0.3;
    }
    ...
    /***** INPUT PAD*****/
    cell(INBUF) {
        area : 0.000000;
        pad_cell : true;
        bond_pads : 1;
        driver_sites : 1;
        pin(PAD ) {
            direction : input;
            is_pad : true;
            input_voltage : CMOS;
            capacitance : 2.500000;
            fanout_load : 0.000000;
        }
        pin(Y ) {
            direction : output;
            function : "PAD";
            timing() {
                cell_rise(scalar) {
                    values( " 3.07 ");
                }
                rise_transition(scalar) {
                    values( " 0.50 ");
                }
                cell_fall(scalar) {
                    values( " 2.95 ");
                }
                fall_transition(scalar) {
                    values( " 0.50 ");
                }
            }
        }
    }
}
```

Chapter 16: Defining I/O Pads
Pad Cell Examples

```
        }
        related_pin :"PAD";
    }
}
}
```

Example 143 Input Buffer With Hysteresis

```
library (example1) {
    date : "August 14, 2015" ;
    revision : 2015.05;
    ...
    time_unit : "1ns";
    voltage_unit : "1V";
    current_unit : "1uA";
    pulling_resistance_unit : "1kohm";
    capacitive_load_unit( 0.1,ff );
    ...
    input_voltage(CMOS_SCHMITT) {
        vil : 1.0;
        vih : 4.0;
        vimin : -0.3;
        vimax : VDD + 0.3;
    }
    ...
/*INPUT PAD WITH HYSTERESIS*/
cell(INBUFH) {
    area : 0.000000;
    pad_cell : true;
    pin(PAD ) {
        direction : input;
        is_pad : true;
        hysteresis : true;
        input_voltage : CMOS_SCHMITT;
        capacitance : 2.500000;
        fanout_load : 0.000000;
    }
    pin(Y ) {
        direction : output;
        function : "PAD";
        timing() {
            cell_rise(scalar) {
                values( " 3.07 ");
            }
            rise_transition(scalar) {
                values( " 0.50 ");
            }
            cell_fall(scalar) {
                values( " 2.95 ");
            }
            fall_transition(scalar) {
                values( " 0.50 ");
            }
        }
    }
}
```

Chapter 16: Defining I/O Pads Pad Cell Examples

```
        related_pin :"PAD";
    }
}
}
```

Example 144 Input Clock Buffer

```
library (example1) {
  date : "August 12, 2015" ;
  revision : 2015.05;
  ...
  time_unit : "1ns";
  voltage_unit : "1V";
  current_unit : "1uA";
  pulling_resistance_unit : "1kohm";
  capacitive_load_unit( 0.1,ff );
  ...
  input_voltage(CMOS) {
    vil : 1.5;
    vih : 3.5;
    vimin : -0.3;
    vimax : VDD + 0.3;
  }
  ...
  /***** CLOCK INPUT BUFFER *****/
  cell(CLKBUF) {
    area : 0.000000;
    pad_cell : true;
    pad_type : clock;
    pin(PAD ) {
      direction : input;
      is_pad : true;
      input_voltage : CMOS;
      capacitance : 2.500000;
      fanout_load : 0.000000;
    }
    pin(Y ) {
      direction : output;
      function : "PAD";
      max_fanout : 2000.000000;
      timing() {
        cell_rise(scalar) {
          values( " 5.70 ");
        }
        rise_transition(scalar) {
          values( " 0.009921 ");
        }
        cell_fall(scalar) {
          values( " 6.90 ");
        }
        fall_transition(scalar) {
          values( " 0.010238 ");
        }
      }
    }
  }
}
```

```
        related_pin :"PAD";
    }
}
}
```

Output Pads

Example 145 Standard Output Buffer

```
library (example1) {
  date : "August 12, 2015" ;
  revision : 2015.05;
  ...
  time_unit : "1ns";
  voltage_unit : "1V";
  current_unit : "1uA";
  pulling_resistance_unit : "1kohm";
  capacitive_load_unit( 0.1,ff );
  ...
  output_voltage(GENERAL) {
    vol : 0.4;
    voh : 2.4;
    vomin : -0.3;
    vomax : VDD + 0.3;
  }
  /***** OUTPUT PAD *****/
  cell(OUTBUF) {
    area : 0.000000;
    pad_cell : true;
    pin(D) {
      direction : input;
      capacitance : 1.800000;
    }
    pin(PAD) {
      direction : output;
      is_pad : true;
      drive_current : 2.0;
      output_voltage : GENERAL;
      function : "D";
      timing() {
        cell_rise(scalar) {
          values( " 8.487 " );
        }
        rise_transition(scalar) {
          values( " 0.16974 " );
        }
        cell_fall(scalar) {
          values( " 9.347 " );
        }
        fall_transition(scalar) {
          values( " 0.18696 " );
        }
      }
    }
  }
}
```

```
        }
        related_pin :"D";
    }
}
}
```

Bidirectional Pad

Example 146 Bidirectional Pad Cell With Three-State Enable

```
library (example1) {
    date : "August 12, 2015" ;
    revision : 2015.08;
    ...
    time_unit : "1ns";
    voltage_unit : "1V";
    current_unit : "1uA";
    pulling_resistance_unit : "1kohm";
    capacitive_load_unit( 0.1,ff );
    ...
    output_voltage(GENERAL) {
        vol : 0.4;
        voh : 2.4;
        vomin : -0.3;
        vomax : VDD + 0.3;
    }
    /***** BIDIRECTIONAL PAD *****/
    cell(BIBUF) {
        area : 0.000000;
        pad_cell : true;
        pin(E D ) {
            direction : input;
            capacitance : 1.800000;
        }
        pin(Y ) {
            direction : output;
            function : "PAD";
            driver_type : "open_source pull_up";
            pulling_resistance : 10000;
            timing() {
                cell_rise(scalar) {
                    values( " 3.07 ");
                }
                rise_transition(scalar) {
                    values( " 0.50 ");
                }
                cell_fall(scalar) {
                    values( " 2.95 ");
                }
                fall_transition(scalar) {
                    values( " 0.50 ");
                }
            }
        }
    }
}
```

Chapter 16: Defining I/O Pads
Pad Cell Examples

```
        }
        related_pin :"PAD";
    }
}
pin(PAD ) {
    direction : inout;
    is_pad : true;
    drive_current : 2.0;
    output_voltage : GENERAL;
    input_voltage : CMOS;
    function : "D";
    three_state : "E";
    timing() {
        cell_rise(scalar) {
            values( " 8.483 ");
        }
        rise_transition(scalar) {
            values( " 0.34686 ");
        }
        cell_fall(scalar) {
            values( " 19.065 ");
        }
        fall_transition(scalar) {
            values( " 0.3813 ");
        }
    }
    related_pin :"E";
}
timing() {
    cell_rise(scalar) {
        values( " 17.466 ");
    }
    rise_transition(scalar) {
        values( " 0.16974 ");
    }
    cell_fall(scalar) {
        values( " 9.348 ");
    }
    fall_transition(scalar) {
        values( " 0.18696 ");
    }
    related_pin :"D";
}
}
```

Cell with contention_condition and x_function

Example 147 Cell With contention_condition and x_function Attributes

```
default_fanout_load : 0.1;
default_inout_pin_cap : 0.1;
```

Chapter 16: Defining I/O Pads Pad Cell Examples

```
default_input_pin_cap : 0.1;
default_output_pin_cap : 0.1;

capacitive_load_unit(1, pf);
pulling_resistance_unit : 1ohm;

voltage_unit : 1V;
current_unit : 1mA;
time_unit : 1ps;

cell (cell_a) {
    area : 1;
    contention_condition : "!ap & an";

    pin (ap, an) {
        direction : input;
        capacitance : 1;
    }
    pin (io) {
        direction : output;
        function : "!ap & !an";
        three_state : "ap & !an";
        x_function : "!ap & an";
        timing() {
            related_pin : "ap an";
            timing_type : three_state_disable;
            cell_rise(scalar) {
                values( " 0.10 ");
            }
            rise_transition(scalar) {
                values( " 0.01 ");
            }
            cell_fall(scalar) {
                values( " 0.10 ");
            }
            fall_transition(scalar) {
                values( " 0.01 ");
            }
        }
        timing() {
            related_pin : "ap an";
            timing_type : three_state_enable;
            cell_rise(scalar) {
                values( " 0.10 ");
            }
            rise_transition(scalar) {
                values( " 0.01 ");
            }
            cell_fall(scalar) {
                values( " 0.10 ");
            }
            fall_transition(scalar) {
                values( " 0.01 ");
            }
        }
    }
}
```

Chapter 16: Defining I/O Pads Pad Cell Examples

```
        }
    }
}
pin (z) {
    direction : output;
    function : "!ap & !an";
    x_function : "!ap & an | ap & !an";
}
}
```

See [contention_condition Attribute on page 64](#) and the “[x_function Attribute](#)” description in [Describing Clock Pin Functions on page 96](#) for more information about these attributes.

A

Library Characterization Configuration

Library information is generated by characterizing the behavior of the library cells under specific conditions. You can specify these conditions in one or more of the `char_config` groups.

Specifying the library characterization settings includes the following concepts and tasks explained in this chapter:

- [The `char_config` Group](#)
 - [Common Characterization Attributes](#)
 - [NLPM Characterization Attributes](#)
 - [CCS Timing Characterization Attributes](#)
 - [Input-Capacitance Characterization Attributes](#)
-

The `char_config` Group

The `char_config` group represents library characterization configuration and is a group of attributes that specify the settings to characterize a library. The library characterization settings include general and specific settings. The general settings are for common tasks, such as characterizing delays, input waveforms, output loads, and handling simulation results. The specific settings include settings for specific characterization models, such as delay, slew, constraint, power, and capacitance models.

Without the appropriate settings, library data can be misinterpreted. This can result in significant differences between the library data and SPICE simulation results. These settings are also critical for accurate recharacterization of the library.

You can define the `char_config` group within the `library`, `cell`, `pin`, and `timing` groups. You should use only one `char_config` group within each of these groups.

Library Characterization Configuration Syntax

[Example 148](#) shows the general syntax for library characterization configuration.

Example 148 General Syntax for Library Characterization Configuration

```
library (library_name) {
    char_config() {
        /* characterization configuration attributes */
    }
    ...
    cell (cell_name) {
        char_config() {
            /* characterization configuration attributes */
        }
        ...
        pin(pin_name) {
            char_config() {
                /* characterization configuration attributes */
            }
            timing() {
                char_config() {
                    /* characterization configuration attributes */
                }
            } /* end of timing */
            ...
        } /* end of pin */
        ...
    } /* end of cell */
    ...
} /* end of library */
```

The `char_config` group includes simple, and complex characterization configuration attributes.

These characterization configuration attributes are divided into the following categories:

- Common configuration
- Three-state
- Composite current source (CCS) timing
- Input capacitance

[Example 149](#) shows the use of these attributes to document the library characterization settings.

Example 149 Library Characterization Configuration Example

```
library (library_test) {
    lu_table_template(waveform_template) {
        variable_1 : input_net_transition;
        variable_2 : normalized_voltage;
        index_1 ("0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7");
        index_2 ("0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9");
    }
}
```

Appendix A: Library Characterization Configuration

The char_config Group

```
normalized_driver_waveform (waveform_template) {
    driver_waveform_name : input_driver;
    values ("0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09", \
            "0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19", \
            ...
            "0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9");
}
normalized_driver_waveform (waveform_template) {
    driver_waveform_name : input_driver_cell_test;
    values ("0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09", \
            "0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19", \
            ...
            "0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9");
}
normalized_driver_waveform (waveform_template) {
    driver_waveform_name : input_driver_rise;
    values ("0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09", \
            "0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19", \
            ...
            "0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9");
}
normalized_driver_waveform (waveform_template) {
    driver_waveform_name : input_driver_fall;
    values ("0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09", \
            "0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19", \
            ...
            "0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9");
}
char_config() {
/* library level default attributes*/
    driver_waveform(all, input_driver);
    input_stimulus_transition(all, 0.1);
    input_stimulus_interval(all, 100.0);
    unrelated_output_net_capacitance(all, 1.0);
    default_value_selection_method(all, any);
    merge_tolerance_abs( nldm, 0.1) ;
    merge_tolerance_abs( constraint, 0.1) ;
    merge_tolerance_abs( capacitance, 0.01) ;
    merge_tolerance_abs( nlpm, 0.05) ;
    merge_tolerance_rel( all, 2.0) ;
    merge_selection( all, max) ;

    internal_power_calculation : exclude_switching_on_rise ;
    ccs_timing_segment_voltage_tolerance_rel: 1.0 ;
    ccs_timing_delay_tolerance_rel: 2.0 ;
    ccs_timing_voltage_margin_tolerance_rel: 1.0 ;
    receiver_capacitance1_voltage_lower_threshold_pct_rise : 20.0;
    receiver_capacitance1_voltage_upper_threshold_pct_rise : 50.0;
    receiver_capacitance1_voltage_lower_threshold_pct_fall : 50.0;
    receiver_capacitance1_voltage_upper_threshold_pct_fall : 80.0;
    receiver_capacitance2_voltage_lower_threshold_pct_rise : 20.0;
    receiver_capacitance2_voltage_upper_threshold_pct_rise : 50.0;
    receiver_capacitance2_voltage_lower_threshold_pct_fall : 50.0;
```

Appendix A: Library Characterization Configuration

Common Characterization Attributes

```
receiver_capacitance2_voltage_upper_threshold_pct_fall : 80.0;
capacitance_voltage_lower_threshold_pct_rise : 20.0;
capacitance_voltage_lower_threshold_pct_fall : 50.0;
capacitance_voltage_upper_threshold_pct_rise : 50.0;
capacitance_voltage_upper_threshold_pct_fall : 80.0;
...
}
...
cell (cell_test) {
    char_config() {
        /* input driver for cell_test specifically */
        driver_waveform (all, input_driver_cell_test);
        /* specific default arc selection method for constraint */
        default_value_selection_method (constraint, max);
        default_value_selection_method_rise(nldm_transition, min);
        default_value_selection_method_fall(nldm_transition, max);
        internal_power_calculation : exclude_switching_on_rise;
        ...
    }
    ...
    pin(pin1) {
        char_config() {
            driver_waveform_rise(delay, input_driver_rise);
            internal_power_calculation : exclude_switching_on_rise;
        }
        ...
    }
    timing() {
        char_config() {
            driver_waveform_rise(constraint, input_driver_rise);
            driver_waveform_fall(constraint, input_driver_fall);
            /* specific ccs segmentation tolerance for this timing arc */
            ccs_timing_segment_voltage_tolerance_rel: 2.0 ;
        }
        /* timing */
    } /* pin */
} /* cell */
} /* lib */
```

Common Characterization Attributes

To specify the common characterization settings, set the common configuration attributes. All common configuration attributes of the `char_config` group are complex. A complex characterization configuration attribute has the following syntax:

Syntax

```
complex_attribute_name ( char_model, value ) ;
```

The first argument of the complex attribute is the characterization model. The second argument is a value of this attribute, such as a waveform name, a specific characterization

Appendix A: Library Characterization Configuration

Common Characterization Attributes

method, a numerical value of a model parameter, or other values. Use the syntax to apply the attribute value to a specific characterization model. You can specify multiple complex attributes in the `char_config` group. You can also specify a single complex attribute multiple times for different characterization models.

However, when you specify the same attribute in multiple `char_config` groups at different levels, such as at the library, cell, pin, and timing levels, the attribute specified at the lower level gets priority over the ones specified at the higher levels. For example, the pin-level `char_config` group attributes have higher priority over the library-level `char_config` group attributes. [Table 37](#) lists the valid characterization models of the `char_config` group and their corresponding descriptions.

Table 37 Valid Characterization Models of the char_config Group

Characterization model	Description
all	Default model. The <code>all</code> model has the lowest priority among the characterization models. Any other characterization model overrides the <code>all</code> model.
ndlsm	Nonlinear delay model (NLDM)
nldm_delay	Specific NLDMs with higher priority over the default NLDM
nldm_transition	
capacitance	Capacitance model
constraint	Constraint model
constraint_setup	Specific constraint models with higher priority over the general constraint model
constraint_hold	
constraint_recovery	
constraint_removal	
constraint_skew	
constraint_min_pulse_width	
constraint_no_change	
constraint_non_seq_setup	
constraint_non_seq_hold	
constraint_minimum_period	
nlpm	Nonlinear power model (NLPM)

Characterization model	Description
nlpm_leakage	Specific NLPMs with higher priority over the general NLPM
nlpm_input	
nlpm_output	

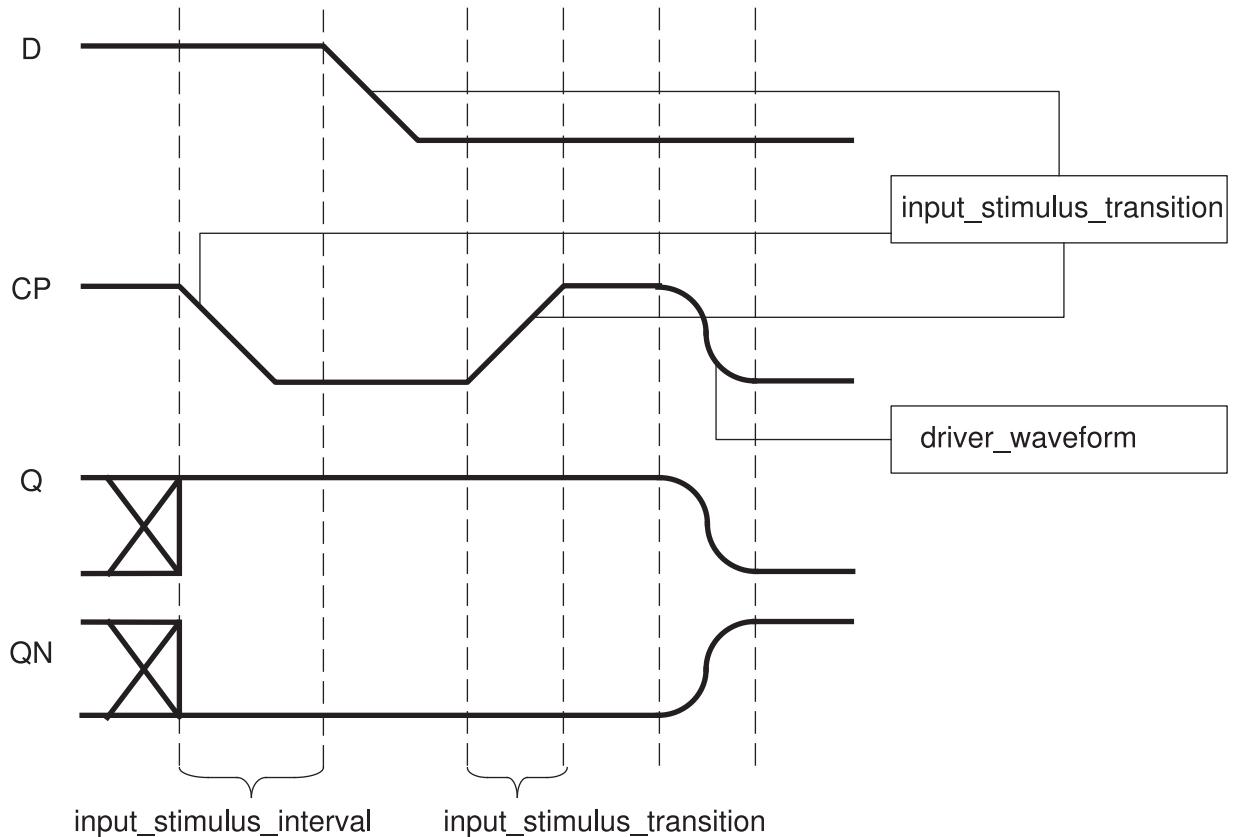
[Example 150](#) shows the syntax of the characterization configuration complex attributes.

Example 150 Syntax of Common Configuration Complex Attributes in the char_config Group

```
char_config() {
    driver_waveform(char_model, waveform_name);
    driver_waveform_rise(char_model, waveform_name);
    driver_waveform_fall(char_model, waveform_name);
    input_stimulus_transition(char_model, float);
    input_stimulus_interval(char_model, float);
    unrelated_output_net_capacitance(char_model, float);
    default_value_selection_method(char_model, method);
    default_value_selection_method_rise(char_model, method);
    default_value_selection_method_fall(char_model, method);
    merge_tolerance_abs(char_model, float) ;
    merge_tolerance_rel(char_model, float) ;
    merge_selection(char_model, method) ;
    ...
}
```

[Figure 119](#) illustrates the use of `driver_waveform`, `input_stimulus_transition`, and `input_stimulus_interval` attributes for the delay arc characterization of a D flip-flop.

Figure 119 Delay Arc Characterization



In Figure 119, an input stimulus with multiple transitions characterizes the delay arc, CP to Q or QN. The `input_stimulus_transition` and `input_stimulus_interval` attributes define and control the transitions and corresponding intervals, respectively. The `driver_waveform` attribute controls the last transition of the clock pulse, CP.

driver_waveform Attribute

The `driver_waveform` attribute defines the driver waveform to characterize a specific characterization model.

You can define the `driver_waveform` attribute within the `char_config` group at the library, cell, pin, and timing levels. If you define the `driver_waveform` attribute within the `char_config` group at the library level, the library-level `normalized_driver_waveform` group is ignored when the `driver_waveform_name` attribute is not defined.

If you do not define this attribute in the `char_config` group, the ramp waveform is used by default.

driver_waveform_rise and driver_waveform_fall Attributes

The `driver_waveform_rise` and `driver_waveform_fall` attributes define a specific rising and falling driver waveform, respectively, for a specific characterization model.

You can define the `driver_waveform_rise` and `driver_waveform_fall` attributes within the `char_config` group at the library, cell, pin, and timing levels. If you define the `driver_waveform_rise` and `driver_waveform_fall` attributes within the `char_config` group at the library level, the library-level `normalized_driver_waveform` group is ignored when the `driver_waveform_name` attribute is not defined.

If you do not define these attributes in the `char_config` group, the ramp waveform is used by default.

input_stimulus_transition Attribute

The `input_stimulus_transition` attribute specifies the transition time for all the input-signal edges except the arc input pin's last transition, during generation of the input stimulus for simulation. For example, in figure in [Common Characterization Attributes](#), the last transition of the clock pulse, CP, uses the `driver_waveform` attribute, and not the `input_stimulus_transition` attribute.

The time units of the `input_stimulus_transition` attribute are specified by the library-level `time_unit` attribute.

You must define this attribute.

input_stimulus_interval Attribute

The `input_stimulus_interval` attribute specifies the time-interval between the input-signal toggles to generate the input stimulus for a characterization cell. The time units of this attribute are specified by the library-level `time_unit` attribute.

You must define the `input_stimulus_interval` attribute.

unrelated_output_net_capacitance Attribute

The `unrelated_output_net_capacitance` attribute specifies a load value for an output pin that is not a related output pin of the characterization model. The valid value is a floating-point number, and is defined by the library-level `capacitive_load_unit` attribute.

If you do not specify this attribute for the `nldm_delay` and `nlpm_output` characterization models, the unrelated output pins use the load value of the related output pin. However, you must specify this attribute for any other characterization model.

default_value_selection_method Attribute

The `default_value_selection_method` attribute defines the method of selecting a default for

- The delay arc from state-dependent delay arcs.
- The constraint arc from state-dependent constraint arcs.
- Pin-based minimum pulse-width constraints from simulated results with side pin combinations.
- Internal power arcs from multiple state-dependent `internal_power` groups.
- The `cell_leakage_power` attribute from the state-dependent values in leakage power models.
- The input-pin capacitance from capacitance values for input-slew values used for timing characterization.

default_value_selection_method_rise and default_value_selection_method_fall Attributes

Use the `default_value_selection_method_rise` and `default_value_selection_method_fall` attributes when the selection methods for rise and fall are different.

You must define either the `default_value_selection_method` attribute, or the `default_value_selection_method_rise` and `default_value_selection_method_fall` attributes. Table 38 lists the valid selection methods for the `default_value_selection_method`, `default_value_selection_method_rise`, `default_value_selection_method_fall`, and `merge_selection` attributes and their descriptions.

Table 38 Valid Common Configuration Selection Methods

Method	Description
any	Selects a random value from the state-dependent data
min	Selects the minimum value from the state-dependent data at each index point.
max	Selects the maximum value from the state-dependent data at each index point.
average	Selects an average value from the state-dependent data at each index point.

Method	Description
min_mid_table	When the state-dependent data is a lookup table (LUT), this method selects the minimum value from the LUT. The minimum value is selected by comparing the middle value in the LUT, with each of the table-values. Note: The middle value corresponds to an index value. If the number of index values is odd, then the middle value is taken as the median value. However, if the number of index values is even, then the smaller of the two values is selected as the middle value.
max_mid_table	When the state-dependent data is a lookup table (LUT), this method selects the maximum value from the LUT. The maximum value is selected by comparing the middle value in the LUT, with each of the table-values. Note: The middle value corresponds to an index value. If the number of index values is odd, then the middle value is taken as the median value. However, if the number of index values is even, then the smaller of the two values is selected as the middle value.
follow_delay	Selects the value from the state-dependent data for delay selection. This method is valid only for the nldm_transition characterization model, that is, the follow_delay method applies specifically to default transition-table selection and not any other default selection.

merge_tolerance_abs and merge_tolerance_rel Attributes

The `merge_tolerance_abs` and `merge_tolerance_rel` attributes specify the absolute and relative tolerances, respectively, to merge arc simulation results. Specify the absolute tolerance value in the corresponding library unit, and the relative tolerance value in percent, for example, 10.0 for 10 percent.

If you specify both the `merge_tolerance_abs` and `merge_tolerance_rel` attributes, the results are merged if either or both the tolerance conditions are satisfied. If you do not specify any of these attributes, data including identical data is not merged.

merge_selection Attribute

The `merge_selection` attribute specifies the method of selecting the merged data. When multiple sets of state-dependent data are merged, the attribute selects a

particular set of the state-dependent data to represent the merged data. See the table in [default_value_selection_method_rise](#) and [default_value_selection_method_fall](#) Attributes for the valid methods and their descriptions of the `merge_selection` attribute.

You must define the `merge_selection` attribute if you have defined either of the `merge_tolerance_abs` or `merge_tolerance_rel` attributes.

NLPM Characterization Attributes

The NLPM `internal_power` group of the output pin includes the `rise_power` and `fall_power` lookup tables that store the respective values. The `internal_power` lookup table values account for the switching energy. This is done using one of the following two methods:

- Deducting CV^2 from only the `rise_power` total energy values
- Deducting $CV^2/2$ from each of the `rise_power` and `fall_power` total energy values

The total energy is the energy dissipated when the pin makes a transition excluding the static leakage energy. CV^2 is the switching energy dissipated by the capacitive load on a net when the net makes a logical transition.

To specify the method to deduct the switching energy from the `internal_power` table values, set the `internal_power_calculation` attribute. Specify this attribute only if the `internal_power` group exists in the library. You can define the `internal_power_calculation` attribute within the `char_config` group at the library, cell, and pin levels.

[Example 151](#) example shows the syntax of the `internal_power_calculation simple` attribute.

Example 151 Syntax of NLPM Characterization Simple Attribute

```
char_config() {
    internal_power_calculation : exclude_switching_on_rise | \
        exclude_switching_on_rise_and_fall | include_switching];
    ...
}
```

The `internal_power_calculation` attribute has the following values:

- `exclude_switching_on_rise`
Indicates that the switching energy is deducted only from the `rise_power` table values.
- `exclude_switching_on_rise_and_fall`
Indicates that the switching energy is deducted from both the `rise_power` and `fall_power` table values

- `include_switching`

Indicates that the switching energy is not deducted from the table values in the `internal_power` group

The attribute does not have a default.

The following example shows the use of the `internal_power_calculation` attribute.

```
library (lib1) {
    ...
    char_config () {
        ...
        internal_power_calculation : exclude_switching_on_rise;
        ...
    }
    cell (cell1) {
        char_config () {
            ...
            internal_power_calculation : exclude_switching_on_rise_and_fall;
            ...
        }
        pin(pin1) {
            char_config () {
                ...
                internal_power_calculation : include_switching;
                ...
            }
            ...
        } /* end of pin */
        ...
    } /* end of cell */
    ...
} /* end of library */
```

CCS Timing Characterization Attributes

To specify the CCS timing characterization settings, set the simple attributes that define CCS timing generation. [Example 152](#) shows the syntax of these simple attributes.

Example 152 Syntax of CCS Timing Simple Attributes

```
char_config() {
    ccs_timing_segment_voltage_tolerance_rel: float;
    ccs_timing_delay_tolerance_rel: float;
    ccs_timing_voltage_margin_tolerance_rel: float;
    receiver_capacitance1_voltage_lower_threshold_pct_rise : float;
    receiver_capacitance1_voltage_upper_threshold_pct_rise : float;
    receiver_capacitance1_voltage_lower_threshold_pct_fall : float;
    receiver_capacitance1_voltage_upper_threshold_pct_fall : float;
    receiver_capacitance2_voltage_lower_threshold_pct_rise : float;
```

```
    receiver_capacitance2_voltage_upper_threshold_pct_rise : float;
    receiver_capacitance2_voltage_lower_threshold_pct_fall : float;
    receiver_capacitance2_voltage_upper_threshold_pct_fall : float;
    ...
}
```

You must define all these attributes if the library includes a CCS model.

ccs_timing_segment_voltage_tolerance_rel Attribute

The `ccs_timing_segment_voltage_tolerance_rel` attribute specifies the maximum permissible voltage difference between the simulation waveform and the CCS waveform to select the CCS model point. The floating-point value is specified in percent, where 100.0 represents 100 percent maximum permissible voltage difference.

ccs_timing_delay_tolerance_rel Attribute

The `ccs_timing_delay_tolerance_rel` attribute specifies the acceptable difference between the CCS waveform delay and the delay measured from simulation. The floating-point value is specified in percent, where 100.0 represents 100 percent acceptable difference.

ccs_timing_voltage_margin_tolerance_rel Attribute

The `ccs_timing_voltage_margin_tolerance_rel` attribute specifies the voltage tolerance to determine whether a signal has reached the rail-voltage value. The floating-point value is specified as a percentage of the rail voltage, such as 96.0 for 96 percent of the rail voltage.

CCS Receiver Capacitance Attributes

The following attributes specify the current integration limits, as a percentage of the voltage, to calculate the CCS receiver capacitances:

- `receiver_capacitance1_voltage_lower_threshold_pct_rise`
- `receiver_capacitance1_voltage_upper_threshold_pct_rise`
- `receiver_capacitance1_voltage_lower_threshold_pct_fall`
- `receiver_capacitance1_voltage_upper_threshold_pct_fall`
- `receiver_capacitance2_voltage_lower_threshold_pct_rise`
- `receiver_capacitance2_voltage_upper_threshold_pct_rise`

- receiver_capacitance2_voltage_lower_threshold_pct_fall
- receiver_capacitance2_voltage_upper_threshold_pct_fall

The floating-point values of these attributes can vary from 0.0 to 100.0.

Input-Capacitance Characterization Attributes

To specify input-capacitance characterization settings, set the simple attributes that define input-capacitance measurement. [Example 153](#) shows the syntax of these simple attributes.

Example 153 Syntax of Input-Capacitance Characterization Simple Attributes

```
char_config() {
    capacitance_voltage_lower_threshold_pct_rise : float;
    capacitance_voltage_lower_threshold_pct_fall : float;
    capacitance_voltage_upper_threshold_pct_rise : float;
    capacitance_voltage_upper_threshold_pct_fall : float;
    ...
}
```

Each floating-point threshold value is specified as a percentage of the supply voltage, and can vary from 0.0 to 100.0.

You must define all the simple attributes mentioned in [Example 153](#), in the `char_config` group for input-capacitance characterization.

capacitance_voltage_lower_threshold_pct_rise and capacitance_voltage_lower_threshold_pct_fall Attributes

The `capacitance_voltage_lower_threshold_pct_rise` and `capacitance_voltage_lower_threshold_pct_fall` attributes specify the lower-threshold value of a rising and falling voltage waveform, respectively, for calculating the NLDM input-pin capacitance.

capacitance_voltage_upper_threshold_pct_rise and capacitance_voltage_upper_threshold_pct_fall Attributes

The `capacitance_voltage_upper_threshold_pct_rise` and `capacitance_voltage_upper_threshold_pct_fall` attributes specify the upper-threshold value of a rising and falling voltage waveform, respectively, for calculating the NLDM input-pin capacitance.