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

Version E-2010.12, December 2010

SYNOPSYS®

Copyright Notice and Proprietary Information

Right to Copy Documentation

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

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

Destination Control Statement

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

Disclaimer

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

Registered Trademarks (®)

Synopsys, AEON, AMPS, Astro, Behavior Extracting Synthesis Technology, Cadabra, CATS, Certify, CHIPit, CODE V, CoMET, Confirma, Design Compiler, DesignWare, EMBED-IT!, Formality, Galaxy Custom Designer, Global Synthesis, HAPS, HapsTrak, HDL Analyst, HSIM, HSPICE, Identify, Leda, LightTools, MAST, METeor, ModelTools, NanoSim, NOVeA, OpenVera, ORA, PathMill, Physical Compiler, PrimeTime, SCOPE, Simply Better Results, SiVL, SNUG, SolvNet, Sonic Focus, STAR Memory System, Syndicated, Synplicity, Synplify, Synplify Pro, Synthesis Constraints Optimization Environment, TetraMAX, the Synplicity logo, UMRBus, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

Trademarks (™)

AFGen, Apollo, ARC, ASAP, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, BEST, Columbia, Columbia-CE, Cosmos, CosmosLE, CosmosScope, CRITIC, CustomExplorer, CustomSim, DC Expert, DC Professional, DC Ultra, Design Analyzer, Design Vision, DesignerHDL, DesignPower, DFTMAX, Direct Silicon Access, Discovery, Eclypse, Encore, EPIC, Galaxy, HANEX, HDL Compiler, Hercules, Hierarchical Optimization Technology, High-performance ASIC Prototyping System, HSIM^{plus}, i-Virtual Stepper, IICE, in-Sync, iN-Tandem, Intelli, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Macro-PLUS, Magellan, Mars, Mars-Rail, Mars-Xtalk, Milkyway, ModelSource, Module Compiler, MultiPoint, ORAengineering, Physical Analyst, Planet, Planet-PL, Polaris, Power Compiler, Raphael, RippledMixer, Saturn, Scirocco, Scirocco-i, SiWare, Star-RCXT, Star-SimXT, StarRC, System Compiler, System Designer, Taurus, TotalRecall, TSUPREM-4, VCSi, VHDL Compiler, VMC, and Worksheet Buffer are trademarks of Synopsys, Inc.

Service Marks (^{SM})

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

SystemC is a trademark of the Open SystemC Initiative and is used under license.
ARM and AMBA are registered trademarks of ARM Limited.
Saber is a registered trademark of SabreMark Limited Partnership and is used under license.
All other product or company names may be trademarks of their respective owners.

Contents

6. Advanced Composite Current Source Modeling

7. Nonlinear Signal Integrity Modeling

9. Composite Current Source Power Modeling

Preface

This preface includes the following sections:

What's New in This Release

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

To see the *Library Compiler Release Notes*,

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

   https://solvnet.synopsys.com/DownloadCenter

   If prompted, enter your user name and password. If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.
2. Select Library Compiler, and then select a release in the list that appears.

About This Guide

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

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

Following is a description of each volume's contents:

- *Library Compiler Technology and Symbol Libraries Reference Manual* provides information to be used with synthesis, test, and power tools.
- *Library Compiler VHDL Libraries Reference Manual* provides information to be used with simulation tools.
- *Library Compiler Physical Libraries Reference Manual* provides information required for floorplanning, RC estimation and extraction, placement, and routing.
- *Library Compiler Methodology and Modeling Functionality in Technology Libraries User Guide* describes the Library Compiler software and explains how to build libraries and define cells.

- *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide* describes how to model power, timing, optimization, and a physical library for Library Compiler.
- *Library Compiler Physical Libraries User Guide* describes how to develop physical libraries.

Audience

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

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

Related Publications

For additional information about Library Compiler, see the documentation on SolvNet at the following address:

https://solvnet.synopsys.com/DocsOnWeb

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

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

Conventions

The following conventions are used in Synopsys documentation.

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

Customer Support

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

Accessing SolvNet

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

To access SolvNet, go to the following address:

https://solvnet.synopsys.com

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

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

Contacting the Synopsys Technical Support Center

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

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

1

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

The power a circuit dissipates falls into two broad categories:

- Static power
- Dynamic power

Static Power

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

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

Dynamic Power

Dynamic power is the power dissipated when a circuit is active. A circuit is active anytime the voltage on a net changes due to some stimulus applied to the circuit. Because voltage on a net can change without necessarily resulting in a logic transition, dynamic power can result even when a net does not change its logic state.

The dynamic power of a circuit is composed of

- Internal power
- Switching power

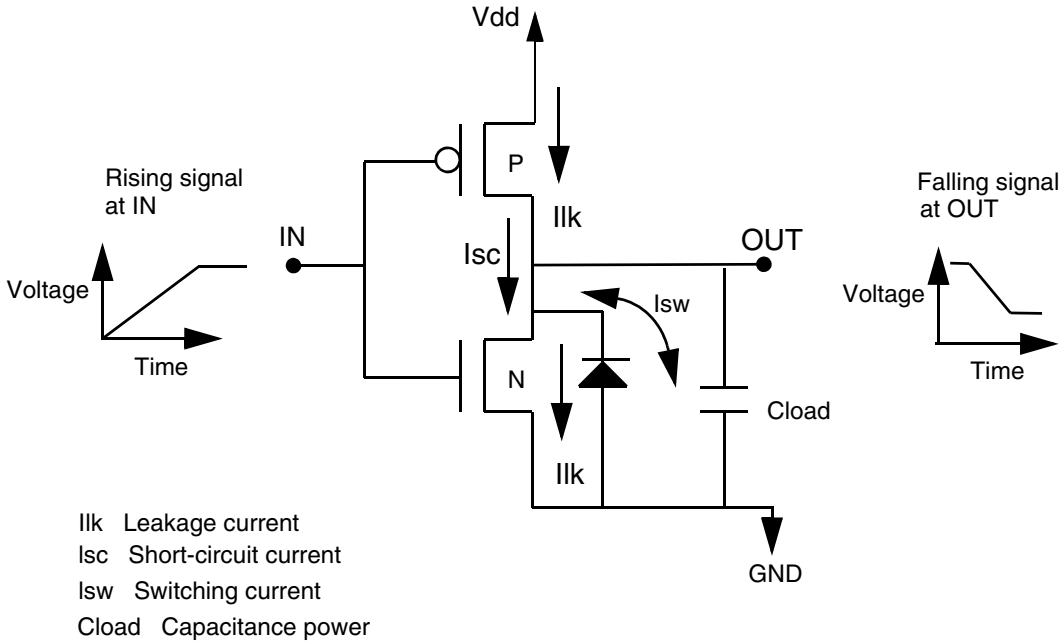
Internal Power

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

Figure 1-1 illustrates components of power dissipation and shows the cause of short-circuit power. In this figure, there is a slow rising signal at the gate input IN. As the signal makes a transition from low to high, the N-type transistor turns on and the P-type transistor turns off.

However, during signal transition, both the P- and N-type transistors can be on simultaneously for a short time. During this time, current flows from Vdd to GND, resulting in short-circuit power.

Figure 1-1 Components of Power Dissipation



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

In most simple library cells, internal power is due primarily to short-circuit power. For this reason, the terms *internal power* and *short-circuit power* are often considered synonymous.

Note:

A transition implies either a rising or a falling signal; therefore, if the power characterization involves running a full-cycle simulation, which includes both rising and falling signals, then you must average the energy dissipation measurement by dividing by 2.

Switching Power

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

$$(1/2)CV^2 \cdot TR_i$$

where TR_i is the toggle-rate activity. This equation is applied to each net in the design.

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

Figure 1-1 shows how the capacitance (Cload) is charged and discharged as the N or P transistor turns on.

Switching power accounts for 70 to 90 percent of the power dissipation of an active CMOS circuit.

Note:

Two measures of power dissipation are useful to designers: average power and peak power. Average power is the power dissipated by a circuit over a representative set of input stimuli. Peak power is the maximum power dissipated by a circuit over all the input stimuli. Peak power is normally associated with a particular stimulus.

Switching Activity

Switching activity is a metric used to measure the number of transitions (0-to-1 and 1-to-0) for every net in a circuit when input stimuli are applied. Switching activity is the average activity of the circuit with a set of input stimuli.

A circuit with higher switching activity is likely to dissipate more power than a circuit with lower switching activity. Switching activity is also correlated with the random-pattern testability of the circuit. A circuit with higher switching activity implies that a randomly selected input pattern might have better coverage.

For more information about switching activity and power consumption, see the *Power Compiler Reference Manual*.

Modeling Cells for Power

The three components of power dissipation are

- Leakage power
- Internal (short-circuit) power
- Switching power

As this equation shows, leakage power is summed over all cells in the design to yield the design's total leakage power (total static power dissipation):

$$P_{LeakageTotal} = \sum_{\forall cells(i)} P_{CellLeakage_i}$$

$P_{LeakageTotal}$

Total leakage power dissipation of the design.

$P_{CellLeakage}$

Leakage power dissipation of the cell.

As the following equation shows, the internal power of the cells and the switching power of the nets are used to compute the design's total dynamic power dissipation:

$$P_{Dynamic} = \underbrace{\sum_{\forall cells(i)} \left(P_{Cellinternal_i} \times E_i \right)}_{\text{Internal power}} + \underbrace{\frac{Vdd^2}{2} \sum_{\forall nets(i)} \left(C_{Load_i} \times E_i \right)}_{\text{Switching power}}$$

$P_{Dynamic}$

Total dynamic power dissipation of the design.

$P_{Cellinternal}$

Internal power of each cell.

V_{dd}

Supply voltage.

C_{Load}

Capacitive load of each net.

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

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

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

Modeling for Leakage Power

Regardless of the physical reasons for leakage power, library developers can annotate gates with the approximate total leakage power dissipated by the gate.

Leakage power characterization requires measuring the supply current (I_{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

$$P_{leakage} = VDD \times I_{supply}$$

Alternatively, leakage power can also be characterized with two simulation runs: one for output high and one for output low. The average of these two measurements is then used as the cell's leakage power. This method reduces characterization effort at the expense of accuracy.

Representing Leakage Power Information

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 value for both leakage and power density

cell_leakage_power Simple Attribute

This attribute specifies the leakage power of a cell. If this attribute is missing or negative, the value of the `default_cell_leakage_power` attribute is assigned.

Syntax

```
cell_leakage_power : value ;
```

Note:

You must define this attribute for cells with state-dependent leakage power to provide the leakage power value for those states where the state-specific leakage power has not been specified using the `leakage_power` group. Otherwise, Library Compiler issues an error.

Using the leakage_power Group for a Single Value

This group specifies the leakage power of a cell when the leakage power depends on the logical condition of that cell. This type of leakage power is called state-dependent. To model state-dependent leakage power, use the following attributes:

- mode
- when
- value

Syntax

```
leakage_power ( ) {    mode (mode_name, mode_value);    when : "Boolean
expression";    value: float;}
```

mode Complex Attribute

You define the `mode` attribute within a `leakage_power` group. A `mode` attribute pertains to an individual cell. The cell is active when `mode` is instantiated with a name and a value. You can specify multiple instances of the `mode` attribute but only one instance for each cell.

Syntax

```
mode (mode_name, mode_value);
```

Library Compiler issues an error message if the *mode_name* and *mode_value* strings are not already defined by a `mode_definition` group in the cell.

Example

```
cell (my_cell) {
  mode_definition (<mode_name>) {
    mode_value(namestring) {
      when : <boolean expression>;
```

```
        sdf_cond : <boolean expression>;
      } ...
    ...
    leakage_power (namestring) {
       mode (mode_name, mode_value);
       /*when : <boolean expression>;*/
       value : <float>;
       ...
    }/* end leakage_power() */
    leakage_current() { /* without the when statement */
       /* default state */
       ...
    }
  }/* end pin B */
  }/* end cell */
```

Example 1-1 shows a `mode` instance description.

Example 1-1 A mode Instance Description

```
library (my_library) {
  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";
         }
     }
     leakage_power () {
         mode(rw, read);
         value : 2;
     }
     leakage_power () {
         mode(rw, write);
         value : 2.2;
         }
     pin(I) {
         direction : input;
         max_transition : 2100.0;
         capacitance : 0.002000;
         fanout_load : 1;
         ...
     }
     ...
  } /* cell(inv0d0) */
```

```
} /* library
```

when Simple Attribute

This attribute specifies the state-dependent condition that determines whether the leakage power is accessed.

Syntax

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

Boolean expression

The name or names of pins, buses, and bundles with their corresponding Boolean operators. Table 1-1 lists valid operators.

Table 1-1 Valid Boolean Operators

Operator	Description
’	invert previous expression
!	invert following expression
^	logical XOR
*	logical AND
&	logical AND
space	logical AND
+	logical OR
|	logical OR
1	signal tied to logic 1
0	signal tied to logic 0

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

You must define mutually exclusive conditions for state-dependent leakage power and internal power. Mutually exclusive means that only one condition defined in the `when` attribute can be met at any given time.

You can reference buses and bundles in the `when` attribute in the `leakage_power` group, as shown here:

Syntax

```
when : "bus_name";
```

Example

```
cell(Z){
   ...
   leakage_power () {
      when : "A";
      ...
   }
}
```

value Simple Attribute

This attribute represents the leakage power for a given state of a cell. The value for this attribute is a floating-point number.

Example

```
cell (my cell) {
   ...
   leakage_power () {
      when : "! A";
      value : 2.0;
   }
   cell_leakage_power : 3.0;
}
```

Using the leakage_power Group for a Polynomial

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

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

The Library Compiler syntax for modeling power is shown below.

Syntax

```
library (poly_sample) {
   delay_model : polynomial ; /* polynomial template */
   power_supply () {
      ...
   } /* end power_supply */
```

```
power_poly_template(poly_template_name_id) {
   variables (variable_1, variable_2, ..., variable_n) ;
   variable_1_range (min_float, max_float) ;
   variable_2_range (min_float, max_float) ;
   ...
   variable_n_range (min_float, max_float) ;
   mapping (voltage1, power_rail_name_id) ;
   mapping (voltage2, power_rail_name_id) ;
   domain (domain_name_id) {
   variables (variable_1, variable_2, ..., variable_n) ;
   variable_1_range (min_float, max_float) ;
   variable_2_range (min_float, max_float) ;
   ...
 variable_n_range (min_float, max_float) ;
   mapping (voltage1, power_rail_name_id) ;
   mapping (voltage2, power_rail_name_id) ;
} /* end power_poly_template */
 ...
}
cell(name) {
   leakage_power() {
      when : "Boolean expression" ;
      power(poly_template_name_id) {
         /* variable ranges are optional for overwriting */
         variable_1_range (min_float, max_float) ;
         variable_2_range (min_float, max_float) ;
      ......
         variable_n_range (min_float, max_float) ;
         orders ("int , ..., int") ;
         coefs("float, ..., float") ;
         ...
      }
   }
}
cell(name) {
   /* piecewise polynomial */
   leakage_power() {
      when : "Boolean expression" ;
      power(poly_template_name_id) {
         domain (domain_name_id) {
         /* variable ranges are optional for overwriting */
         variable_1_range (min_float, max_float) ;
         variable_2_range (min_float, max_float) ;
         ...
         variable_n_range (min_float, max_float) ;
         orders ("int , ..., int") ;
         coefs ("float, ..., float") ;
      } /* end power */
    ...
```

```
      } /* end leakage_power */
   } / end cell */
} /* end library */
```

Specifying leakage power in a power Group

The `power` group is defined within a `leakage_power` group in a `cell` group at the library level, as shown here:

```
library (name) {
   cell (name) {
      leakage_power () {
         power (template name) {
         ... power template description ...
         }
      }
   }
}
```

Complex Attributes

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

Group

```
domain
```

orders Attribute

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

coefs Attribute

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

domain Group

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

leakage_power_unit Simple Attribute

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

Table 1-2 Valid Unit Values and Mathematical Equivalents

Text entry	Mathematical equivalent
1mW	1mW
100uW	100 micro W
10uW	10 micro W
1uW	1 micro W
100nW	100nW
10nW	10nW
1nW	1nW
100pW	100pW
10pW	10pW
1pW	1pW

Example

```
leakage_power_unit : 100uW;
```

If this attribute is missing, the leakage-power values are expressed without units.

default_cell_leakage_power Simple Attribute

The `default_cell_leakage_power` attribute indicates the default leakage power for those cells for which you have not set the `cell_leakage_power` attribute. This attribute must be a nonnegative floating-point number. If you do not define the `default_cell_leakage_power` attribute, it defaults to 0.0.

Example

```
default_cell_leakage_power : 0.5;
```

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

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

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

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

default_leakage_power_density Simple Attribute

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

Example

```
default_leakage_power_density : 0.5;
```

Library Compiler looks for a definition for both the `default_leakage_power_density` and `default_cell_leakage_power` attributes. For details about the `default_cell_leakage_power` attribute, see “default_cell_leakage_power Simple Attribute” on page 1-14.

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

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

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

Example

```
library(leakage) {
   delay_model : table_lookup;

   /* unit attributes */
   time_unit : "1ns";
   voltage_unit : "1V";
   current_unit : "1mA";
   pulling_resistance_unit : "1kohm";
   leakage_power_unit : "1pW";
   capacitive_load_unit (1.0,pf);

   /* power supply definition */
   power_supply() {
      default_power_rail : VDD;
      power_rail (VDD1, 1.0);
      power_rail (VDD2, 2.0);
   }
   cell (NAND2) {
      rail_connection (P1,VDD1);
      rail_connection (P2,VDD2);

      cell_leakage_power : 1.0 ;
      leakage_power() {
         power_level : "VDD1";
         when : "!A1 !A2" ;
         value : 1.5 ;
      }
      leakage_power() {
         power_level : "VDD1";
         when : "!A1 A2" ;
         value : 2.0 ;
      }
      leakage_power() {
         power_level : "VDD1";
         when : "A1 !A2" ;
         value : 3.0 ;
      }
      leakage_power() {
         power_level : "VDD1";
         when : "A1 A2" ;
         value : 4.0 ;
      }
      leakage_power() {
```

```
      power_level : "VDD2";
      when : "!A1 !A2" ;
      value : 3.5 ;
   }
   leakage_power() {
      power_level : "VDD2";
      when : "!A1 A2" ;
      value : 3.0 ;
   }
   leakage_power() {
      power_level : "VDD2";
      when : "A1 !A2" ;
      value : 4.0 ;
   }

   leakage_power() {
      power_level : "VDD2";
      when : "A1 A2" ;
      value : 5.0 ;
   }
   area : 1.0 ;
   pin(A1) {
      direction : input;
      capacitance : 0.1 ;
   }
   pin(A2) {
      direction : input;
      capacitance : 0.1 ;
   }
   pin(ZN) {
      direction : output;
      max_capacitance : 0.1;
      function :  "(A1*A2)'";
      timing() {
         timing_sense : "negative_unate"
         related_pin : "A1"
         cell_rise( scalar ) {
            values("0.0");
         }
         rise_transition( scalar ) {
            values("0.0");
         }
         cell_fall( scalar ) {
            values("0.0");
         }
         fall_transition( scalar ) {
            values("0.0"); }
         }
         internal_power() {
            related_pin : " A1 "
            power_level : "VDD1";
            rise_power( scalar ) {
               values("0.0");
```

```
            }
            fall_power( scalar ) {
               values("0.0");
            }
         }  /* end of internal power */
         internal_power() {
            related_pin : " A1 "
            power_level : "VDD2";
            rise_power( scalar ) {
               values("0.0");
            }
            fall_power( scalar ) {
               values("0.0");
            }
         }  /* end of internal power */
      } /* end of timing for related pin A1 */
      timing() {
         timing_sense : "negative_unate"
         related_pin : "A2"
         cell_rise( scalar ) {
            values("0.0"); }
         rise_transition( scalar ) {
            values("0.0"); }
         cell_fall( scalar ) {
            values("0.0");
         }
         fall_transition( scalar ) {
            values("0.0");
         }
         internal_power() {
            related_pin : " A2 "
            power_level : "VDD1";
            rise_power( scalar ) {
               values("0.0");
            }
            fall_power( scalar ) {
               values("0.0");
            }
         }  /* end of internal power */
         internal_power() {
            related_pin : " A2 "
            power_level : "VDD2";
            rise_power( scalar ) {
               values("0.0");
            }
            fall_power( scalar ) {
               values("0.0");
            }
         } /* end of internal power */
      } /* end of timing for related pin A2 */
   } /* end of pin ZN */
} /* end of cell */
} /* end of library */
```

Environmental Derating Factors Attributes

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

- `k_volt_cell_leakage_power`
- `k_temp_cell_leakage_power`
- `k_process_cell_leakage_power`

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

Example

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

See the *Library Compiler Technology and Symbol Libraries Reference Manual* for the `cell_leakage_power` attribute syntax. See Chapter 11, “Building Environments,” for information about scaling factors, operating conditions, and default values.

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

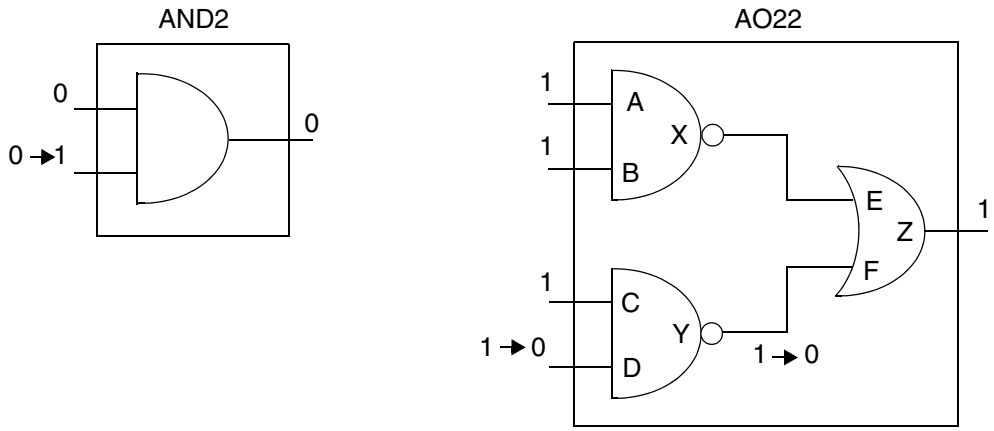
- To include the effect of the output capacitance in the `internal_power` attribute, which gives the output pins zero capacitance
- To give the output pins a real capacitance, which causes them to be included in the switching power, and model only short-circuit power as the cell's internal power

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.

This description is not entirely accurate, because some power is dissipated as a result of internal transitions that do not cause output transitions. However, those are relatively minor in comparison (they consume much less power) and should be ignored.

Figure 1-2 shows two examples of an input transition that does not cause a corresponding output transition.

Figure 1-2 Complex Gate Example



In Case 1, input B of the AND2 gate undergoes a 0-to-1 transition but the output remains stable at 0. This might consume a small amount of power as one of the N-transistors opens, but the current flow will be very small.

In Case 2, input D of the multilevel gate AO22 undergoes a 1-to-0 transition, causing a 1-to-0 transition at internal pin Y. However, output Z remains stable at 1. The significance of the power dissipation in this case depends on the load of the internal wire connected to Y. In Case 1, power dissipation is negligible, but in Case 2, power dissipation might result in some inaccuracy.

You can set the `internal_power` group attribute so that multiple input or output pins that share logic can transition together within the same time period.

Pins transitioning within the same time period can lower the level of power consumption.

Modeling Internal Power Lookup Tables

You should measure the energy dissipated by varying either input voltage transition or output load while holding the other constant. Because a table indexed by T input transition times and C output load capacitances has TxC entries, the cell's internal power must be characterized TxC times, once for each input transition time and output load capacitance combination. For example, if internal power will be modeled by use of a 3x3 table at the output of the cell, the design will have 9 input voltage transitions—output load combinations where energy dissipation must be measured.

The `library` group supports a one-, two-, or three-dimensional `internal power` lookup table indexed by the total output load capacitances (best model), the input transition time, or both. The internal power lookup table uses the same syntax as the nonlinear lookup table for delay.

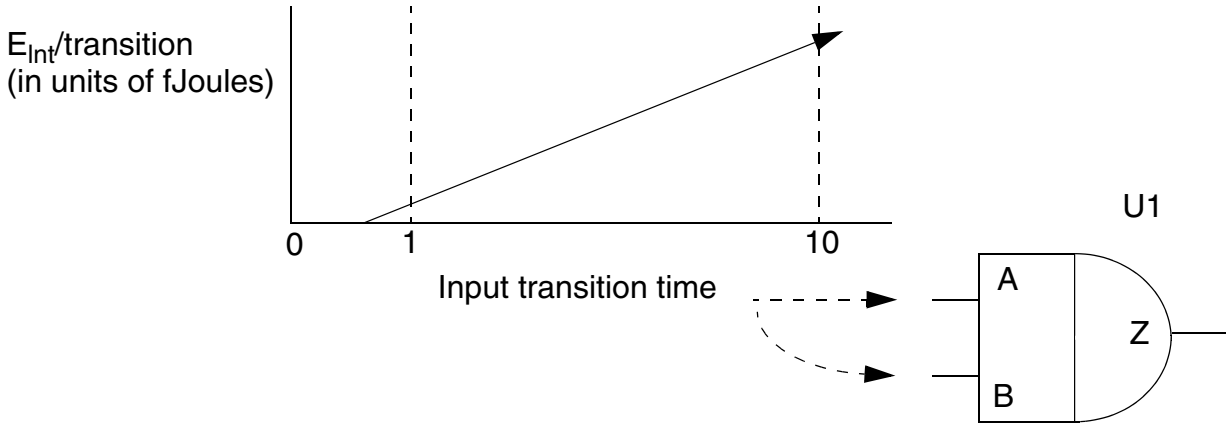
You can set `internal_power` attributes for input pins, which are indexed by input transition time. In this way, you can model gate AO22 in Figure 1-2 or, more important, the power consumed by the flip-flop clock or reset pins.

Note:

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

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

Figure 1-3 Internal Power for Cell U1



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

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

P_{Int}

Total internal power for the cell.

E

Internal energy for the pin.

AF

Activity factor.

Accurate sequential modeling requires a separate table for the clock and for the output pin the clock controls. The two tables are used to ensure that clock pin power and output power are accounted for separately, because a clock pin often toggles without causing any observable state change on the output pin. The separate power table scheme ensures that power dissipated within the cell is accounted for properly when the clock pin toggles but the output pin does not. The following discussion pertains to single-output, single-clock sequential cells, but the concept is also extendable to multioutput, multiclock cells.

Clock Pin Power

This energy is characterized by simulation of a single full cycle (one rise transition and one fall transition) of the clock, with no transition at the output and input pins. A one-dimensional internal power table indexed by input transition time should be attached to the clock pin. Total energy dissipated in the cell during this simulation is measured. If separate rise and fall

power modeling is not used, the energy measured must be divided by 2 to get the energy dissipated by the clock pin transition, because the measurement is done for two transitions of the clock.

```
Clk_Pin_Energy = Clk_Total / 2
```

Add `Clk_Pin_Energy` as an entry indexed by input transition time in the one-dimensional internal power table attached to the clock pin.

Output Pin Power

This power is characterized by simulation of two full cycles of the clock, with two rise and fall transitions at the output. A two-dimensional internal power table should be attached to the output pin. Total energy dissipated in the cell during the two-full-cycle simulation (Out_total) is measured. If separate rise and fall power modeling is not used, the energy measured must be divided by 2, because the measurement is done for two transitions.

```
Output_Pin_Energy = (Out_total)/2 - 2*(Clk_Pin_Energy)
```

or

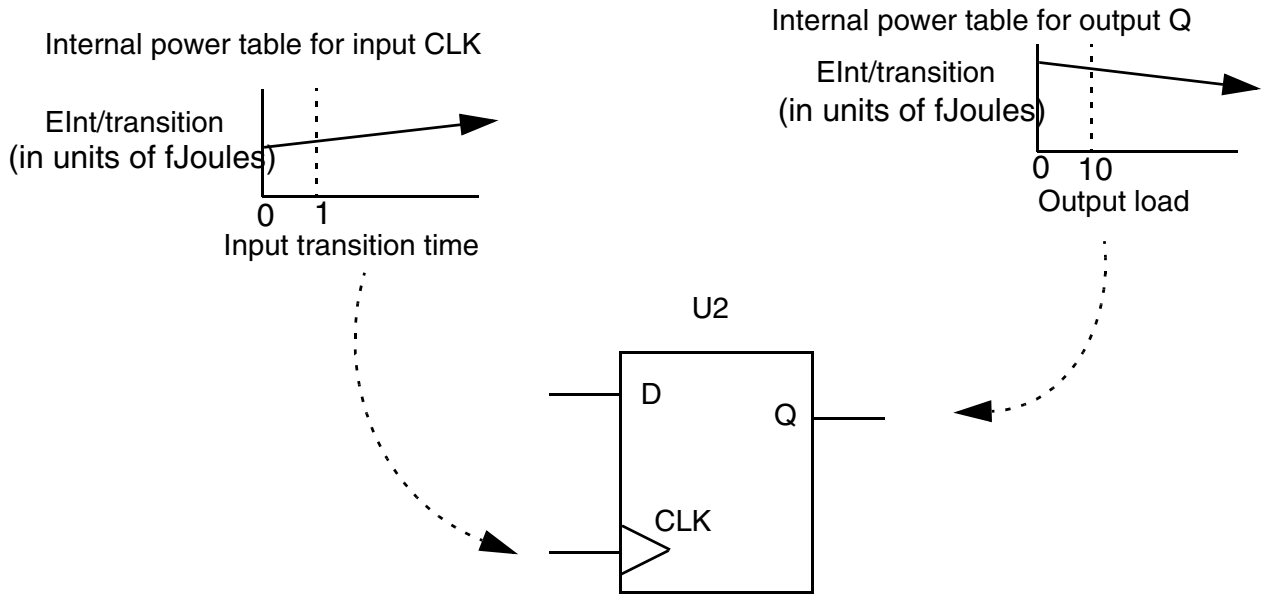
```
Output_Pin_Energy = (Out_total)/2 - Clk_Total
```

Note:

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

Figure 1-4 shows how to calculate the input pin power information for internal cell U2.

Figure 1-4 Internal Power for Cell U2



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

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

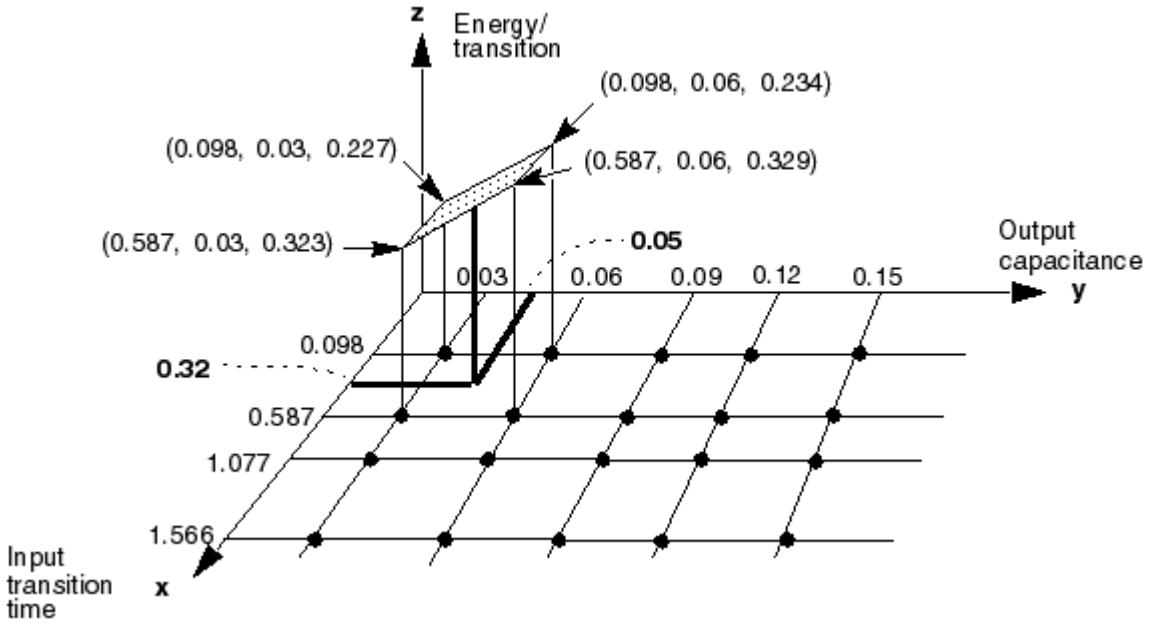
P_{Int}
: Total internal power for the cell.

E
: Internal energy for the pin.

AF
: Activity factor.

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

Figure 1-5 Internal Power Table for Cell Output



Calculating Switching Power

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

Switching power information is a function of a net's toggle rate, capacitive loading, associated clock frequency, and the supply voltage level of the design. The Synopsys library model supports all of these parameters except for the toggle rate.

An explicit units attribute is not required for switching power, because the units are implicitly determined by the units of the voltage, time, and capacitance attributes.

Because all toggle rates are internally adjusted to a period defined by the library-level `time_unit` attribute, the resulting value of switching power is defined in terms of joules per second (or watts) multiplied by the appropriate power of 10, as determined by the units for

time, capacitance, and voltage. For example, in a library with time units of 1 ns, capacitive units of 0.1 femtofarad, and voltage units of 1 volt, the calculation for the derived units for the library's switching power is:

$$\text{Power\_Units} = \frac{(1\ \text{V}^2) \times 0.1\ \text{ff}}{1\ \text{ns}} = 0.1\ \mu\text{W}$$

For a single net with a total load of 100 femtofarad, a toggle rate of two transitions every 100 ns, and a supply voltage of 5 volts, the calculation of the net's power dissipation is:

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

TR

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

C_{Load}

Capacitive load of each net.

Creating a Scalable Polynomial Power Model

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

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

- Polynomial representation
- Model description
- Scalable polynomial power calculation

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

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

Note:

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

Polynomial Representation

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

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

$$P_1Q_2 = (a_0 + a_1x_1)\,(b_0 + b_1x_2 + b_2x_2^2) =$$
$$A_{00} + A_{10}x_1 + A_{01}x_2 + A_{11}x_1x_2 + A_{02}x_2^2 + A_{12}x_1x_2^2$$

Model Description

The scalable polynomial power model syntax allows you to specify up to variable *n* polynomials. The dimensions are temperature, voltage, input_net_transition, output_net_transition, total_output_net_capacitance, equal_or_opposite_output_net_transition, and voltage*i*.

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

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

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

You specify polynomials in the expanded decomposed form. Assume m, n, p, q, r, s, and u as the orders of each of seven dimensions (x1—x7). The number of coefficients is

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

The coefficients are expressed as

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

Examples

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

Example 1-2 Sequence of Coefficients in the Case of a Three-Dimensional Polynomial

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

Example 1-3 shows how the sequence of the coefficients is represented in the `power` group:

Example 1-3 Sequence of Coefficients in the power Group

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

The scalable polynomial power model significantly improves memory usage, because

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

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

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

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

Representing Internal Power Information

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

Specifying the Power Model

Use the library level `power_model` attribute to specify the power model for your library. The two valid values are table_lookup and polynomial. If you do not specify a power model, Library Compiler assumes table_lookup.

Using Lookup Table Templates

To represent internal power, you can create templates of common information that multiple lookup tables can use. Use the following groups and attributes to define your lookup tables:

- The library-level `power_lut_template` group
- The `internal_power` group (see "Defining Internal Power Groups" on page 1-36)
- The associated library-level attributes that specify scaling factors and a default value

power_lut_template Group

Use this library-level group to create templates of common information that multiple lookup tables can use. A table template specifies the table parameters and the breakpoints for each axis. Assign each template a name. Make the template name the group name of a `fall_power` group, `power` group, or `rise_power` group in the `internal_power` group; doing so enables the power lookup tables to refer to the template.

Syntax

```
power_lut_template(name) {variable_1 : string ;variable_2 : string ;
          variable_3 : string ;index_1("float, ... , float") ;
          index_2("float, ... , float") ;index_3("float, ... , float") ;}
```

Template Variables

The lookup table template for specifying power uses three associated variables to characterize cells in the library for internal power:

- `variable_1`, which specifies the first dimensional variable
- `variable_2`, which specifies the second dimensional variable
- `variable_3`, which specifies the third dimensional variable

These variables indicate the parameters used to index into the lookup table along the first, second, and third table axes.

Following are valid values for `variable_1`, `variable_2`, and `variable_3`:

total_output_net_capacitance

The loading of the output net capacitance of the pin specified in the `pin` group that contains the `internal_power` group.

equal_or_opposite_output_net_capacitance

The loading of the output net capacitance of the pin specified in the `equal_or_opposite_output` attribute in the `internal_power` group.

input_transition_time

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

For information about the `related_pin` attribute, see "Defining Internal Power Groups" on page 1-36.

Template Breakpoints

The index statements define the breakpoints for an axis. The breakpoints defined by `index_1` correspond to the parameter values indicated by `variable_1`. The breakpoints defined by `index_2` correspond to the parameter values indicated by `variable_2`. The breakpoints defined by `index_3` correspond to the parameter values indicated by `variable_3`.

You can overwrite the `index_1`, `index_2`, and `index_3` attribute values by providing the same `index_x` attributes in the `fall_power` group, `power` group, or `rise_power` group in the `internal_power` group.

The index values are lists of floating-point numbers greater than or equal to 0.0. The values in the list must be in increasing order.

The number of floating-point numbers in the indexes determines the size of each dimension, as Example 1-4 illustrates.

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

Example 1-4 shows four `power_lut_template` groups that have one-, two-, or three-dimensional templates.

Example 1-4 Four power_lut_template Groups

```
power_lut_template (output_by_cap) {
   variable_1 : total_output_net_capacitance ;
   index_1 ("0.0, 5.0, 20.0") ;
}

power_lut_template (output_by_cap_and_trans) {
   variable_1 : total_output_net_capacitance ;
   variable_2 : input_transition_time ;
   index_1 ("0.0, 5.0, 20.0") ;
   index_2 ("0.1, 1.0, 5.0") ;
}

power_lut_template (input_by_trans) {
   variable_1 : input_transition_time ;
   index_1 ("0.0, 1.0, 5.0") ;
}

power_lut_template (output_by_cap2_and_trans) {
   variable_1 : total_output_net_capacitance ;
   variable_2 : input_transition_time ;
   variable_3 : equal_or_output_net_capacitance ;
   index_1 ("0.0, 5.0, 20.0") ;
   index_2 ("0.1, 1.0, 5.0") ;
   index_3 ("0.1, 0.5, 1.0") ;
}
```

Scalar power_lut_template Group

Use this group to model cells with no power consumption.

Library Compiler has a predefined template named scalar; its value size is 1. You can refer to it by specifying scalar as the group name of a `fall_power` group, `power` group, or `rise_power` group in the `internal_power` group.

Example

```
internal_power() {
   power(scalar) {
      values ("0.0");
   }
}
```

Note:

You can use the scalar template with an assigned value of 0 (indicating that no power is consumed) for an `internal_power` group with a `rise_power` table and a `fall_power` table. When one group contains the scalar template, the other must contain one-, two-, or three-dimensional power values.

Using Scalable Polynomial Power Modeling

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

- The library-level `power_poly_template` group
- The `internal_power` group (see "Defining Internal Power Groups" on page 1-36)
- The associated library-level attributes that specify scaling factors and a default value

power_poly_template Group

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

Syntax

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

power_poly_template Variables

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

temperature

The temperature.

voltage

The primary power supply voltage.

voltage*i*

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

input_net_transition

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

total_output_net_capacitance

The loading of the output net capacitance of the pin specified in the `pin` group that contains the `internal_power` group

equal_or_opposite_output_net_capacitance

The loading of the output net capacitance of the pin specified in the `equal_or_opposite_output` attribute in the `internal_power` group.

parameter*i*

Used to reference user-defined process variables.

Mapping Power Relationships

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

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

Case 1

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

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

Case 2

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

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

Case 3

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

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

Case 4

The `mapping` attribute is required in order to specify a value defined in a `power_rail` group for `voltage1`. Because there is no mapping statement for the `voltage` variable, Library Compiler assumes the value defined in the `voltage` attribute in the `operating_conditions` group.

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

Case 5

The `mapping` attribute is required in order to specify a value defined in a `power_rail` group for `voltage1`.

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

Case 6

The `mapping` attribute is required in order to specify a value defined in a `power_rail` groups for `voltage1` and optionally to specify a value other than the default for `voltage`.

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

Defining Internal Power Groups

To specify the cell's internal power consumption, use the `internal_power` group within the `pin` group in the cell.

If the `internal_power` group is not present in a cell, it is assumed that the cell does not consume any internal power. You can define the optional complex attribute `index_1`, `index_2`, or `index_3` in this group to overwrite the `index_1`, `index_2`, or `index_3` attribute defined in the library-level `power_lut_template` to which it refers (see Example 1-7 on page 1-59).

Naming Power Relationships, Using the internal_power Group

Within the `internal_power` group you can identify the name or names of different power relationships. A single power relationship can occur between an identified pin and a single related pin identified with the `related_pin` attribute. Multiple power relationships can occur in many ways.

This list shows seven possible multiple power relationships. These relationships are described in more detail in the following sections:

- Between a single pin and a single related pin
- Between a single pin and multiple related pins
- Between a bundle and a single related pin
- Between a bundle and multiple related pins
- Between a bus and a single related pin
- Between a bus and multiple related pins
- Between a bus and related bus pins

Power Relationship Between a Single Pin and a Single Related Pin

Identify the power relationship that occurs between a single pin and a single related pin by entering a name in the `internal_power` group attribute as shown in the following example:

Example

```
cell (my_inverter) {
   ...
   pin (A) {

      direction : input;
      capacitance : 1;
```

```
   }
   pin (B) {
      direction : output
      function : "A'";
      internal_power (A_B) }
         related_pin : "A";
      ...
      }/* end internal_power() */
   }/* end pin B */
}/* end cell */
```

The power relationship is as follows:

From pin	To pin	Label
A	B	A_B

Power Relationships Between a Single Pin and Multiple Related Pins

This section describes how to identify the power relationships when an `internal_power` group is within a `pin` group and the power relationship has more than a single related pin. You identify the multiple power relationships on a name list entered with the `internal_power` group attribute as shown in the following example:

Example

```
cell (my_and) {
      ...
   pin (A) {
      direction : input;
      capacitance : 1;

   }
   pin (B) {
      direction : input;
      capacitance : 2;
   }
   pin (C) {
      direction : output
      function : "A B";
      internal_power (A_C, B_C) {
         related_pin : "A B";
         ...
      }/* end internal_power() */
   }/* end pin B */
}/* end cell */
```

The power relationships are as follows:

From pin	To pin	Label
A	C	A_C
B	C	B_C

Power Relationships Between a Bundle and a Single Related Pin

When the `internal_power` group is within a `bundle` group that has several members that have a single related pin, enter the names of the resulting multiple power relationships in a name list in the `internal_power` group.

Library Compiler assumes that the first name in the name list is the relationship between the related pin and the first pin in the bundle member list, the second name in the name list is the relationship between the related pin and the second pin in the bundle member list, and so on.

Example

```
...
bundle (Q){
   members (Q0, Q1, Q2, Q3);
   direction : output;
   function : "IQ";
   internal_power (G_Q0, G_Q1, G_Q2, G_Q3) {
      related_pin : "G";
   }
}
```

If G is a pin, as opposed to another `bundle` group, the power relationships are as follows:

From pin	To pin	Label
G	Q0	G_Q0
G	Q1	G_Q1
G	Q2	G_Q2
G	Q3	G_Q3

If G is another bundle of member size 4 and G0, G1, G2, and G3 are members of bundle G, the power relationships are as follows:

From pin	To pin	Label
G0	Q0	G_Q0
G1	Q1	G_Q1
G2	Q2	G_Q2
G3	Q3	G_Q3

Note:

If G is a bundle of a member size other than 4, it's an error due to incompatible width.

Power Relationships Between a Bundle and Multiple Related Pins

When the `internal_power` group is within a `bundle` group that has several members, each having a corresponding related pin, enter the names of the resulting multiple power relationships as a name list in the `internal_power` group.

Library Compiler assumes that the first name in the name list is the relationship between the related pin and the first pin in the bundle member list, the second name in the name list is the relationship between the second related pin and the second pin in the bundle member list, and so on.

Example

```
bundle (Q){
   members (Q0, Q1, Q2, Q3);
   direction : output;
   function : "IQ";
   internal_power (G_Q0, H_Q0, G_Q1, H_Q1, G_Q2, H_Q2, G_Q3, H_Q3) {
      related_pin : "G H";
   }
}
```

If G is a pin, as opposed to another `bundle` group, the power relationships are as follows:

From pin	To pin	Label
G	Q0	G_Q0
H	Q0	H_Q0
G	Q1	G_Q1
H	Q1	H_Q1

G	Q2	G_Q2
H	Q2	H_Q2
G	Q3	G_Q3
H	Q3	H_Q3

If G is another bundle of member size 4 and G0, G1, G2, and G3 are members of bundle G, the power relationships are as follows:

From pin	To pin	Label
G0	Q0	G_Q0
H	Q0	H_Q0
G1	Q1	G_Q1
H	Q1	H_Q1
G2	Q2	G_Q2
H	Q2	H_Q2
From pin	To pin	Label
G3	Q3	G_Q3
H	Q3	H_Q3

The same rule applies if H is a size 4 bundle.

Note:
If G is a bundle of a member size other than 4, it's an error due to incompatible width.

Power Relationships Between a Bus and a Single Related Pin

This section describes how to identify the power relationships created when an `internal_power` group is within a `bus` group that has several bits that have the same single related pin. You identify the resulting multiple power relationships by entering a name list, using the `internal_power` group attribute.

Library Compiler assumes that the first name in the name list identifies the relationship between the related pin and the most significant bit (MSB) in the `bus` group, the second name in the name list identifies the relationship between the related pin and the second MSB in the `bus` group, and so on.

Example

```
 ...
bus (X){
/*assuming MSB is X[0] */
   bus_type : bus4;
   direction : output;
   capacitance : 1;
   pin (X[0:3]){
      function : "B'";
      internal_power (B_X0, B_X1, B_X2, B_X3){
         related_pin : "B";
      }
   }
}
```

If B is a pin, as opposed to another 4-bit bus, the power relationships are as follows:

From pin	To pin	Label
B	X[0]	B_X0
B	X[1]	B_X1
B	X[2]	B_X2
B	X[3]	B_X3

If B is another 4-bit bus and B[0] is the MSB for bus B, the power relationships are as follows:

From pin	To pin	Label
B[0]	X[0]	B_X0
B[1]	X[1]	B_X1
B[2]	X[2]	B_X2
B[3]	X[3]	B_X3

Power Relationships Between a Bus and Multiple Related Pins

This section describes the power relationships created when an `internal_power` group is within a `bus` group that has several bits, where each bit has its own related pin. You identify the resulting multiple power relationships by entering a name list, using the `internal_power` group attribute.

Library Compiler assumes that the first name in the name list is the relationship between the first related pin and the most significant bit (MSB) in the `bus` group, the second name in the name list is the relationship between the second related pin and the second MSB in the `bus` group, and so on.

Example

```
bus (X){ /*assuming MSB is X[0] */
    bus_type : bus4;
    direction : output;
    capacitance : 1;
    pin (X[0:3]){
        function : "B'";
        internal_power (B_X0, C_X0, B_X1, C_X1, B_X2, C_X2, B_X3,C_X3 ){
            related_pin : "B C";
        }
    }
}
```

If B and C are pins, as opposed to another 4-bit bus, the power relationships are as follows:

From pin	To pin	Label
B	X[0]	B_X0
C	X[0]	C_X0
B	X[1]	B_X1
C	X[1]	C_X1

From pin	To pin	Label
B	X[2]	B_X2
C	X[2]	C_X2
B	X[3]	B_X3
C	X[3]	C_X3

If B is another 4-bit bus and B[0] is the MSB for bus B, the power relationships are as follows:

From pin	To pin	Label
B[0]	X[0]	B_X0
C	X[0]	C_X0
B[1]	X[1]	B_X1
C	X[1]	C_X1
B[2]	X[2]	B_X2
C	X[2]	C_X2
B[3]	X[3]	B_X3
C	X[3]	C_X3

The same rule applies if C is a 4-bit bus.

Power Relationships Between a Bus and Related Bus Pins

This section describes the power relationships created when an `internal_power` group is within a `bus` group that has several bits that have to be matched with several related bus pins of a required width. You identify the resulting multiple power relationships by entering a name list, using the `internal_power` group attribute.

Library Compiler assumes that the first name in the name list is the relationship between the first pin of the related bus pins of the necessary width and the MSB in the `bus` group, the second name in the name list is the relationship between the second pin of the related bus pins and the second MSB in the `bus` group, and so on.

Example

```
  ...
/* assuming related_bus_pins is width of 2 bits */
bus (X){
/*assuming MSB is X[0] */
    bus_type : bus4;
    direction : output;
    capacitance : 1;
    pin (X[0:3]){
        function : "B'";
        internal_power (B0_X0, B0_X1, B0_X2, B0_X3, B1_X0, B1_X1, B1_X2, B1_X3 ){
            related_bus_pins : "B";
        }
    }
```

```
}
```

If B is another 2-bit bus and B[0] is its MSB, the power relationships are as follows:

From pin	To pin	Label
B[0]	X[0]	B0_X0
B[0]	X[1]	B0_X1
B[0]	X[2]	B0_X2
B[0]	X[3]	B0_X3
B[1]	X[0]	B1_X0
B[1]	X[1]	B1_X1
B[1]	X[2]	B1_X2
B[1]	X[3]	B1_X3

internal_power Group

To define an `internal_power` group in a `pin` group, use these simple attributes, complex attributes, and groups:

Simple attributes:

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

Complex attributes:

- `mode`

Groups:

- fall_power
- power
- rise_power

equal_or_opposite_output Simple Attribute

This attribute designates an optional output pin or pins whose capacitance Library Compiler uses to access a three-dimensional table in the internal_power group.

Syntax

```
equal_or_opposite_output : "name |name_list" ;
```

name | name_list

Name of output pin or pins.

Note:

This pin (or these pins) have to be functionally equal to or the opposite of the pin named in the same pin group.

Example

```
equal_or_opposite_output : "Q" ;
```

Note:

The output capacitance of this pin (or pins) is used as the equal_or_opposite_output_net_capacitance variable in the internal power lookup table.

falling_together_group Simple Attribute

Use the falling_together_group attribute to identify the list of two or more input or output pins that share logic and are falling together during the same time period. Set this time period with the switching_interval attribute. See "switching_interval Simple Attribute" on page 1-49 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.

power_level Simple Attribute

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

For more information, see “Modeling Libraries With Multiple Power Supplies” on page 1-65 or “internal_power Group" in *Library Compiler Technology and Symbol Libraries Reference Manual*.

Syntax

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

name

Name of the power rail defined in the `power_supply` group.

Example

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

For an output pin within a multiple power supply cell, regardless of the `output_signal_level` value, you must define `internal_power` tables for all the `rail_connection` of the cell.

Example

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

```
        }
        internal_power() {
            related_pin : "A";
            power_level : VDD2;    /* must be there */
            power (power_load) {
                values (" 0.111111, 0.111111, 0.111111, 0.111111,\ 0.111111");
            }
        }
    }
  ...
  }
}
```

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

Example

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

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

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

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

Example

```
library (example) {
   ...
   power_supply() {
      default_power_rail : vdd;
      power_rail(VDD1, 1.95);
      power_rail(VDD2, 1.85);
      power_rail(VDD3, 1.75);
   } ;
   ...
   cell ( cell1 ) {
```

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

related_pin Simple Attribute

This attribute associates the `internal_power` group with a specific input or output pin. If `related_pin` is an output pin, it must be functionally equal to or the opposite of the pin in that `pin` group.

If `related_pin` is an input pin or output pin, the pin's transition time is used as a variable in the internal power lookup table.

Syntax

`related_pin : "`*`name | name_list`*`" ;`

name | name_list

Name of the input or output pin or pins

Example

```
related_pin : "A B" ;
```

The pin or pins in the `related_pin` attribute denote the path dependency for the `internal_power` group. Library Compiler accesses a particular `internal_power` group if the input pin or pins cause the corresponding output pin named in the `pin` group to toggle.

If you want to define a two-dimensional or three-dimensional table, specify all functionally related pins in a `related_pin` attribute.

rising_together_group Simple Attribute

The `rising_together_group` attribute identifies the list of two or more input or output pins that share logic and are rising during the same time period. This time period is defined with the `switching_interval` attribute. See "switching_interval Simple Attribute," which follows, for details.

Together, the `rising_together_group` attribute and `switching_interval` attribute settings determine the level of power consumption.

Define the `rising_together_group` attribute in the `internal_power` group, as shown here.

```
cell (name_string) {
   pin (name_string) {
      internal_power () {
         falling_together_group : "list of pins" ;
         rising_together_group : "list of pins" ;
         switching_interval : float;
         rise_power () {
            ...
         }
         fall_power () {
            ...
         }
      }
   }
}
```

list of pins

The names of the input or output pins that share logic and are rising during the same time period.

switching_interval Simple Attribute

The `switching_interval` attribute defines the time interval during which two or more pins that share logic are falling, rising, or switching (either falling or rising) during the same time period.

Set the `switching_interval` attribute together with the `falling_together_group`, `rising_together_group`, or `switching_together_group` attribute. Together with one of these attributes, the `switching_interval` attribute defines a level of power consumption.

For details about the attributes that are set together with the `switching_interval` attribute, see “falling_together_group Simple Attribute” on page 1-45; “rising_together_group Simple Attribute” on page 1-49; and “switching_together_group Simple Attribute,” which follows.

Syntax

```
switching_interval : float ;
```

float

A floating-point number that represents the time interval during which two or more pins that share logic are switching together.

Example

```
pin (Z) {
   direction : output;
   internal_power () {
      switching_together_group : "A B"; /*if pins A, B, and Z switch*/ ;
      switching_interval : 5.0; /*switching within 5 time units */;
      power () {
         ...
      }
   }

}
```

switching_together_group Simple Attribute

The `switching_together_group` attribute identifies the list of two or more input or output pins that share logic, are either falling or rising during the same time period, and are not affecting power consumption.

Define the time period with the `switching_interval` attribute. See “switching_interval Simple Attribute” on page 1-49 for details.

Define the `switching_together_group` attribute in the `internal_power` group, as shown here.

```
cell (name_string) {
   pin (name_string) {
      internal_power () {
         switching_together_group : "list of pins" ;
         switching_interval : float;
         power () {
            ...
         }
      }
   }
}
```

list of pins

The names of the input or output pins that share logic, are either falling or rising during the same time period, and are not affecting power consumption.

when Simple Attribute

This attribute specifies a state-dependent condition that determines whether the internal power table is accessed.

You can use the `when` attribute to define one-, two-, or three-dimensional tables in the `internal_power` group.

Syntax

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

Boolean expression

Name or names of input and output pins, buses, and bundles with corresponding Boolean operators.

Table 1-1 on page 1-10 lists the Boolean operators valid in a `when` statement.

Example

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

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

Example 1-5 The Same Pin Specified in Both the when and related_pin Attributes

```
pin (Z) {
   function : "A & B & C";
   internal_power () {
      related_pin : "B";
      when : "!A & B";
      ...
   }
}
```

If the `when` attribute and the `related_pin` attribute contain the same pin, Library Compiler generates an error message similar to the following:

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

mode Complex Attribute

You define the `mode` attribute within a `internal_power` group. A `mode` attribute pertains to an individual pin. The pin is active when `mode` is instantiated with a name and a value. You can specify multiple instances of the `mode` attribute but only one instance for each pin.

Syntax

```
mode (mode_name, mode_value);
```

Library Compiler issues an error message if the *mode_name* and *mode_value* strings are not already defined by a `mode_definition` group in the cell.

Example

```
cell (my_cell) {
   mode_definition (<mode_name>) {
    mode_value(namestring) {
      when : <boolean expression>;
      sdf_cond : <boolean expression>;
    } …
...
pin (B) {
direction : output
function : <boolean expression>;
internal_power (namestring) {
related_pin : <pin_name>;
/*when : <boolean expression>;*/
mode (mode_name, mode_value);
...
}/* end internal_power() */
}/* end pin B */
}/* end cell */
```

Example 1-6 shows a `mode` instance description.

Example 1-6 A mode Instance Description

```
library (my_library) {
  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 : "A";
        sdf_cond : "A == 1";
      }
      mode_value(write) {
        when : "!A";
        sdf_cond : "A == 0";
        }
    }
    pin(A) {
      direction : input;
      max_transition : 2100.0;
      capacitance : 0.002000;
      fanout_load : 1;
      ...
```

```
   }
   pin(I) {
     direction : input;
     max_transition : 2100.0;
     capacitance : 0.002000;
     fanout_load : 1;
     ...
   }
   pin(Z) {
     direction : output;
     max_capacitance : 0.175000;
     max_fanout : 58;
     max_transition : 1400.0;
     function : "(I)'";
     internal_power () {
       related_pin : "I";
       mode(rw, read);
       /* when : "A";*/
       ...
     }
     internal_power () {
       related_pin : "I";
       mode(rw, write);
       /* when : "!A";*/
      ...
     }
     timing() {
       related_pin : "I";
       timing_sense : negative_unate;
       ...
       } /* timing I -> Z */
    } /* I */
    ...
  } /* cell(inv0d0) */
} /* library */
```

fall_power Group

Use a `fall_power` group to define a fall transition for a pin. If you specify a `fall_power` group, you must also specify a `rise_power` group.

You define a `fall_power` group in an `internal_power` group in a cell-level `pin` group, as shown here:

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

Complex Attributes

```
index_1 ("float, ..., float") ; /* lookup table */
index_2 ("float, ..., float") ; /* lookup table */
index_3 ("float, ..., float") ; /* lookup table */
values ("float, ..., float") ; /* lookup table */
orders ("integer, ..., integer")/* polynomial */
coefs ("float, ..., float")/* polynomial */
```

Group

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

index_1, index_2, index_3 Attributes

These attributes identify internal cell consumption per fall transition. Define these attributes in the `internal_power` group or in the library-level `power_lut_template` group.

values Attribute

This attribute defines internal cell consumption per fall transition.

Example

```
values ("2.2, 3.7, 4.3", "1.7, 2.1, 3.5", "1.0, 1.5, 2.8");
```

orders Attribute

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

coefs Attribute

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

domain Group

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

Complex Attributes

```
variable_n_range
orders
coefs
```

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

Design Compiler converts the `values` attribute information to power consumption by multiplying the unit by the factor transition or `per_unit_time`, as shown here:

- `nindex_1` floating-point numbers if the lookup table is one-dimensional
- `nindex_1` x `nindex_2` floating-point numbers if the lookup table is two-dimensional
- `nindex_1` x `nindex_2` x `nindex_3` floating-point numbers if the lookup table is three-dimensional

The `nindex_1`, `nindex_2`, and `nindex_3` numbers are the size of `index_1`, `index_2`, and `index_3` in this group or in the `power_lut_template` group it inherits. Quotation marks enclose a group. Each group represents a row in the table.

power Group

The `power` group is defined within an `internal_power` group in a `pin` group at the cell level, as shown here:

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

Complex Attributes

```
index_1 ("float, ..., float") ; /* lookup table */
index_2 ("float, ..., float") ; /* lookup table */
index_3 ("float, ..., float") ; /* lookup table */
values ("float, ..., float") ; /* lookup table */
orders ("integer, ..., integer")/* polynomial */
coefs ("float, ..., float")/* polynomial */
```

Group

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

index_1, index_2, index_3 Attributes

These attributes identify internal cell consumption per transition. Define these attributes in the `internal_power` group or in the library-level `power_lut_template` group.

values Attribute

This attribute defines internal cell power consumption per rise or fall transition.

This power information is accessed when the pin has a rise transition or a fall transition. The values in the table specify the average power per transition.

orders Attribute

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

coefs Attribute

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

domain Group

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

Complex Attributes

```
variable_n_range
orders
coefs
```

Note:

The internal power value is derived from the capacitance unit exponent (with mantissa discarded) and the voltage unit.

rise_power Group

A `rise_power` group is defined in an `internal_power` group at the cell level, as shown here:

```
cell (name) {
```

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

Rise power is accessed when the pin has a rise transition. If you have a `rise_power` group, you must have a `fall_power` group.

Complex Attributes

```
index_1 ("float, ..., float") ; /* lookup table */
index_2 ("float, ..., float") ; /* lookup table */
index_3 ("float, ..., float") ; /* lookup table */
values ("float, ..., float") ; /* lookup table */
orders ("integer, ..., integer")/* polynomial */
coefs ("float, ..., float")/* polynomial */
```

Group

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

index_1, index_2, index_3 Attributes

These attributes identify internal cell consumption per rise transition. Define these attributes in the `internal_power` group or in the library-level `power_lut_template` group.

values Attribute

This attribute defines internal cell power consumption per rise transition.

orders Attribute

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

coefs Attribute

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

domain Group

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

```
      }
   }
}
```

Complex Attributes

```
variable_n_range
orders
coefs
```

Note:

Design Compiler converts the `values` attribute information to power consumption, by multiplying the unit by the factor transition or per_unit_time. For more information, see "values Attribute" on page 1-54.

Syntax for One-Dimensional, Two-Dimensional, and Three-Dimensional Tables

You can define a one-, two-, or three-dimensional table in the `internal_power` group in either of the following two ways:

- Using the `power` group. For two- and three-dimensional tables, define the `power` group and the `related_pin` attribute.
- Using a combination of the `related_pin` attribute, the `fall_power` group, and the `rise_power` group.

The syntax for a one-dimensional table using the `power` group is shown here:

```
internal_power() {
   power (template name) {
      values("float, ..., float") ;
   }
}
```

The syntax for a one-dimensional table using the `fall_power` and `rise_power` groups is shown here:

```
internal_power() {
   fall_power (template name) {
      values("float, ..., float");
   }
   rise_power (template name) {
      values("float, ..., float");
   }
}
```

The syntax for a two-dimensional table using the `power` and `related_pin` groups is shown here:

```
internal_power() {
   related_pin : "name | name_list" ;
   power (template name) {
```

```
      values("float, ..., float") ;
   }
}
```

The syntax for a two-dimensional table using the `related_pin`, `fall_power`, and `rise_power` groups is shown here:

```
internal_power() {
   related_pin : "name | name_list" ;
   fall_power (template name) {
      values("float, ..., float");
   }
   rise_power (template name) {
      values("float, ..., float");
   }
}
```

The syntax for a three-dimensional table using the `power` and `related_pin` groups is shown here:

```
internal_power() {
   related_pin : "name | name_list" ;
   power (template name) {
      values("float, ..., float") ;
   }
   equal_or_opposite_output : "name | name_list" ;
}
```

The syntax for a three-dimensional table using the `related_pin`, `fall_power`, and `rise_power` groups is shown here:

```
internal_power() {
   related_pin : "name | name_list" ;
   fall_power (template name) {
      values ("float, ..., float") ;
   }
   rise_power (template name) {
      values ("float, ..., float") ;
   }
   equal_or_opposite_output : "name | name_list" ;
}
```

Example 1-7 shows cells that contain internal power information in the `pin` group.

Example 1-7 A Library With Internal Power

```
library(internal_power_example) {
   ...
   power_lut_template(output_by_cap_and_trans) {
      variable_1 : total_output_net_capacitance ;
      variable_2 : input_transition_time ;
      index_1 ("0.0, 5.0, 20.0");
      index_2 ("0.0, 1.0, 20.0");
```

```
    }
    ...
    cell(AN2) {
       pin(Z) {
          direction : output ;
          internal_power {
             power(output_by_cap_and_trans) {
                values ("2.2, 3.7, 4.3", "1.7, 2.1, 3.5", "1.0, 1.5, 2.8");
             }
             related_pin : "A B" ;
          }
       }
       pin(A) {
          direction : input ;
          ...
       }
       pin(B) {
          direction : input ;
          ...
       }
    }
}
```

Power-Scaling Factors

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

Syntax

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

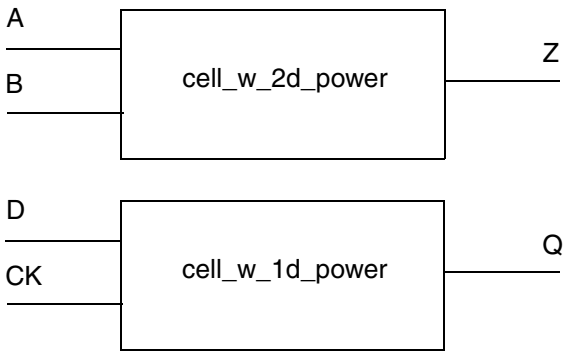
Example

```
library(power_sample) {
   ...
   k_volt_internal_power : 0.000000;
   k_temp_internal_power : 0.000000;
   k_process_internal_power : 0.000000;
   ...
}
```

Internal Power Examples

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

Figure 1-6 Library Cells With Internal Power Information



One-Dimensional Power Lookup Table

Example 1-8 is the library description of the cell shown in Figure 1-6, a cell with a one-dimensional internal power table defined in the `pin` groups.

Example 1-8 One-Dimensional Internal Power Table

```
library(internal_power_example) {
...
   power_lut_template(output_by_cap) {
      variable_1 : total_output_net_capacitance ;
      index_1("0.0, 5.0, 20.0") ;
   }
   ...
   power_lut_template(input_by_trans) {
      variable_1 : input_transition_time ;
      index_1 ("0.0, 1.0, 2.0") ;
   }
   ...
   cell(FLOP1) {
      pin(CP) {
         direction : input ;
         internal_power()
            power (input_by_trans) {
               values("1.5, 2.6, 4.7") ;
            }
         }
      }
      ...
      pin(Q) {
         direction : input ;
         internal_power()
            power(output_by_cap) {
               values("9.0, 5.0, 2.0") ;
            }
         }
      }
```

```
    }
}
```

In Example 1-8, 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:

E_{CP} = 2.6
E_{Q} = 5.0

If the `time_unit` attribute value in the technology library is 1 ns, the total internal power consumed by this gate is

$$Pint = (E_{CP} \times AF_{CP}) + (E_{Q} \times AF_{Q})$$
$$= (2.6 \times 2 \times 10\text{-}2) + (5.0 \times 1 \times 10\text{-}2)$$
$$= 0.102$$

The activity factors AF_{CP} and AF_{Q} are adjusted to the `time_unit` specified in the technology library. The unit of P_{int} depends on the unit of the `voltage_unit` and `capacitive_load_unit` attributes in the technology library.

If the following attributes and values are specified in the technology library, the unit of the `internal_power` group is 10^{-15} joule/transition.

```
voltage_unit : "1V";
capacitive_load_unit(1.0, "ff");
```

Because the unit of the activity factor is transition/ns, the total internal power dissipation of this gate is

```
0.102 x 10-15 joule/transition x transition/10-9second = 0.102 uW.
```

Note:

For a detailed explanation of how to calculate internal power, see the *Power Compiler Reference Manual*.

Two-Dimensional Power Lookup Table

Example 1-9 is the library description of the cell in Figure 1-6, a cell with a two-dimensional internal power table defined in the `pin` groups.

Example 1-9 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 x 2 x 10-2 + 2.5 x 3 x 10-2) / (2 x 10-2  + 3  x 10-2 ) =

2.0
```

Because the output load at pin Z is 5.0, the following power table values are used:

```
2.1 = 1.0A + B
3.5 = 20.0A + B
```

Consequently, the values for A and B become

```
A = 0.0739
B = 2.0263
```

and the resulting values for Ez and Pint are

```
Ez    = 2.0 x 0.0739 + 2.0263
      = 2.1741
      = 2.2

Pint  = Ez  x  AFz
      = 2.2 x 1 x 10-E2
      = 0.022
```

The activity factor AF_Z is adjusted to the `time_unit` specified in the technology library. The unit of `internal_power` can be applied here.

Three-Dimensional Power Lookup Table

Example 1-10 is the library description for the cell in Figure 1-6, a cell with a three-dimensional internal power table.

Example 1-10 Three-Dimensional Internal Power Table

```
library(internal_power_example) {
    ...
    power_lut_template(output_by_cap1_cap2_and_trans) {
        variable_1 : total_output1_net_capacitance ;
        variable_2 : equal_or_opposite_output_net_capacitance ;
        variable_3 : input_transition_time ;
        index_1 ("0.0, 5.0, 20.0") ;
        index_2 ("0.0, 5.0, 20.0") ;
        index_3 ("0.0, 1.0, 2.0") ;
    }
    ...
    cell(FLOP1) {
        pin(CP) {
            direction : input ;
            ...
        }
        pin(D) {
            direction : input ;
            ...
        }
        pin(S) {
            direction : input ;
            ...
        }
        pin(R) {
            direction : input ;
            ...
        }
        pin(Q) {
            direction : output ;
            internal_power() {
                power(output_by_cap1_cap2_and_trans) {
                    values("2.2, 3.7, 4.3", "1.7, 2.1, 3.5", "1.0, 1.5,\
                    .8", "2.1, 3.6, 4.2", "1.6, 2.0, 3.4", "0.9, 1.5, .7" \
                    "2.0, 3.5, 4.1", "1.5, 1.9, 3.3", "0.8, 1.4,2.6");
                }
                equal_or_opposite_output : "QN" ;
                related_pin : "CP" ;
            }
            ...
        }
        ...
    }
}
```

In this example, the input transition time at pin CP is 1.0. The output load at pin QN is 5.0. The output load at pin Q is 5.0. The toggle rate at pin Q is one transition per 100 ns.

Using the transition time and the loading at pin Q and QN to index into the table, you get the following results:

```
EQ = 2.0
```

If the `time_unit` in the technology library is 1 ns, the total internal power consumed by this gate is

```
Pint =     EQ  x  AF_Q
           =     2.0  x  1  x  10-2
           =     0.020
```

The activity factor AF_Q is adjusted to the `time_unit` specified in the technology library. The unit of `internal_power` can be applied here.

Modeling Libraries With Multiple Power Supplies

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

This syntax supports

- Multiple power supplies
- Multiple input/output voltages

Power Supply Information

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

In most technologies, factors such as operating temperature, supply voltage, and the manufacturing process can strongly affect circuit performance. However, the power supply for many new processes has switched from 5 volts to 3.3 volts to lower the power consumption. Therefore, to support more than one power supply (for example, 3.3 volts and 5 volts), you need to make several changes to your technology library.

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

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

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

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

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

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

For more examples, see "Example Libraries With Multiple Power Supplies" on page 1-70.

Defining Power Supply Groups

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

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

Note:

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

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

power_rail Complex Attribute

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

Syntax

power_rail *(power_supply_name$_{string}$,voltage_value$_{float}$)*;

power_supply_name

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

voltage_value

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

Example

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

default_power_rail Simple Attribute

Specifies a default voltage value for nominal operating conditions.

Syntax

default_power_rail : *power_supply_name$_{string}$* ;

Example

```
default_power_rail : VDD1 ;
```

Defining Cells With Multiple Power Supplies

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

rail_connection Complex Attribute

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

Syntax

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

connection_name

Specifies a power connection for the pin name.

power_supply_name

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

Example 1-11 shows a cell with rail_connection attributes.

Example 1-11 cell with rail_connection Attributes

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

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

input_signal_level and output_signal_level Simple Attributes

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

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

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

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

Syntax

```
input_signal_level: power_supply_name_string ;

output_signal_level: power_supply_name_string ;
```

power_supply_name

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

Example

```
input_signal_level: VDD1 ;

output_signal_level: VDD2 ;
```

Example 1-12 shows a library with a `power_supply` group, nominal operating conditions, and an `operating_conditions` group.

Example 1-12 Library With a power_supply Group With Nominal Operating Conditions and an operating_conditions Group

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

Multiple Power Supply Library Requirements

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

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

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

Example Libraries With Multiple Power Supplies

Example 1-13 shows a library with multiple power supplies defined at the library, cell, and pin levels.

Example 1-13 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 1-14 Library Defining All Power Supplies With Nominal Operating Conditions

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

Example 1-15 Cell With Default Power Supply

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

Example 1-16 Cell With One Power Supply

```
cell(with_one_power_supply) {
   area : 10;
   rail_connection  (PV1, VDD1) ;
   pin (A) {
      direction: input;
      capacitance: 1.0;
```

```
      }
      pin (Z) {
         direction: output;
         function : "!A";
      }
   } /* end cell */
```

Example 1-17 Cell With Two Power Supplies

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

Example 1-18 Cell With Two Power Supplies and a Bidirectional Pin

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

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

Input and Output Voltage Information

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

Syntax

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

vil

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

vih

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

vimin

The minimum acceptable input voltage.

vimax

The maximum acceptable input voltage.

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

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

```
input_voltage : my_input_voltage ;
```

input_voltage Group

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

Example

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

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

All voltage values are in the units you define with the `library` group `voltage_unit` attribute.

output_voltage Group

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

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

All voltage values are in the unit you define with the `library` group `voltage_unit` attribute.

Example

```
library (my_library) {
...
  output_voltage(GENERAL) {
     vol : 0.4;
     voh : 2.4;
     vomin : -0.e;
     vomax : VDD + 0.3;
   }
}
```

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

- Driver VOmax > Load VImax
- Driver VOmin< Load VImin

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

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

Modeling Libraries With Integrated Clock-Gating Cells

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

You can perform automatic clock gating at the register transfer level, using the HDL Compiler and Power Compiler tools from Synopsys.

What Clock Gating Does

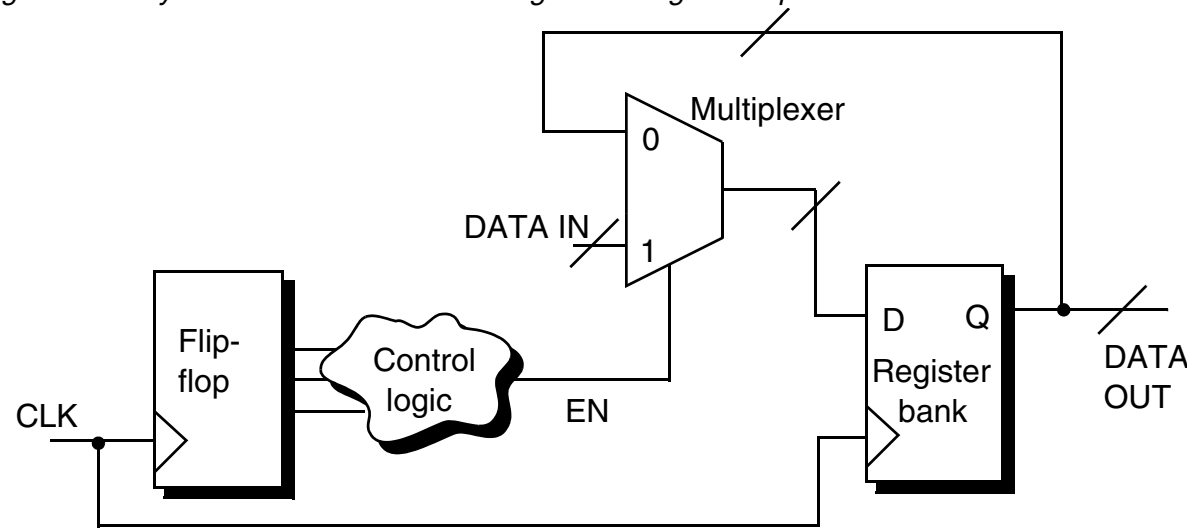
Clock gating provides a power-efficient implementation of register banks that are disabled during some clock cycles.

A register bank is a group of flip-flops that share the same clock and synchronous control signals and that are inferred from the same HDL variable. Synchronous control signals include synchronous load-enable, synchronous set, synchronous reset, and synchronous toggle.

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

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

Figure 1-7 Synchronous Load-Enable Register Using a Multiplexer



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

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

By controlling the clock signal for the register, you can eliminate the need for reloading the same value in the register through multiple clock cycles. Clock gating inserts a 2-input gate into the register bank's clock network, creating the control to eliminate unnecessary register activity.

Clock gating reduces the clock network's power dissipation and often relaxes the datapath timing. If your design has large multibit registers, clock gating can save power and reduce the number of gates in the design. However, for smaller register banks, the overhead of adding logic to the clock tree might not compare favorably to the power saved by eliminating a few feedback nets and multiplexers.

Using integrated clock-gating cell functionality, you have the option of doing the following:

- Use latch-free or latch-based clock gating.
- Insert logic to increase testability.

For details, see "Using an Integrated Clock-Gating Cell" and "Setting Pin Attributes for an Integrated Cell" on page 1-78.

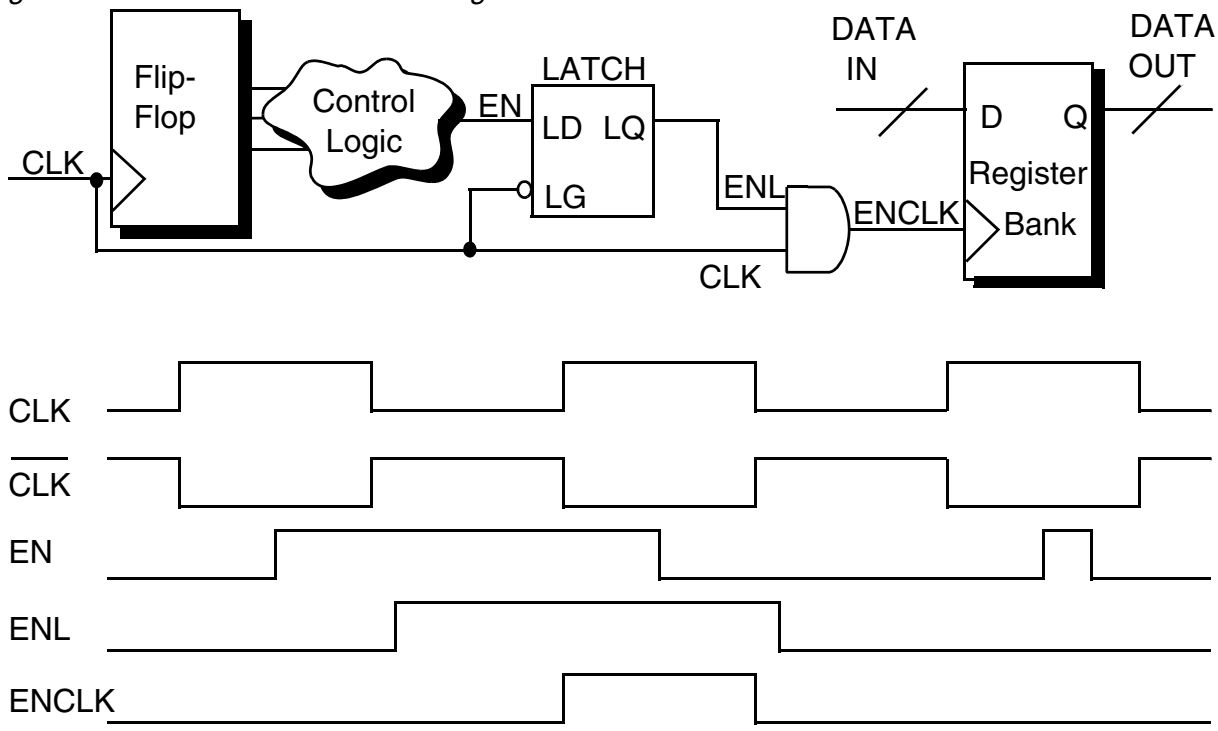
Looking at a Gated Clock

Clock gating saves power by eliminating the unnecessary activity associated with reloading register banks. Clock gating eliminates the feedback net and multiplexer shown in Figure 1-7, by inserting a 2-input gate in the clock net of the register. The 2-input clock gate selectively prevents clock edges, thus preventing the gated clock signal from clocking the gated register.

Clock gating can also insert inverters or buffers to satisfy timing or clock waveform and duty requirements.

Figure 1-8 shows the 2-input clock gate as an AND gate; however, depending on the type of register and the gating style, gating can use NAND, OR, and NOR gates instead. At the bottom of Figure 1-8, the waveforms of the signals are shown with respect to the clock signal, CLK.

Figure 1-8 Latch-Based Clock Gating



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

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

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

Using an Integrated Clock-Gating Cell

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

Create an integrated clock-gating cell that integrates the various combinational and sequential elements of the clock-gating circuitry into a single cell, which is compiled to gates and located in the technology library.

Library Compiler recognizes a clock-gating cell by accessing the state tables and state functions of the library cell pins.

Setting Pin Attributes for an Integrated Cell

The clock-gating software requires the pins of your integrated cells to be set with the attributes listed in Table 1-3. Setting some of the pin attributes, such as those for test and observability, is optional.

Table 1-3 Pin Attributes for Integrated Clock-Gating Cells

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

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

clock_gate_clock_pin Attribute

The `clock_gate_clock_pin` attribute identifies an input pin connected to a clock signal. Design Compiler uses the `clock_gate_clock_pin` attribute when it compiles a design containing gated clocks that were introduced by Power Compiler.

Syntax

```
clock_gate_clock_pin : true | false ;
```

true | false

A true value labels the pin as a clock pin. A false value labels the pin as *not* a clock pin.

Example

```
clock_gate_clock_pin : true;
```

clock_gate_enable_pin Simple Attribute

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

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

Syntax

```
clock_gate_enable_pin : true | false ;
```

true | false

A true value labels the input pin of a clock-gating cell connected to an enable signal as the enable pin. A false value does not label the input pin as an enable pin.

Example

```
clock_gate_enable_pin : true;
```

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

For more information about identifying clock-gating cells, see the *Library Compiler Methodology and Modeling Functionality in Technology Libraries User Guide*.

For more information about identifying pins on integrated clock-gating cells, see “Using an Integrated Clock-Gating Cell” on page 1-78.

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

clock_gate_test_pin Attribute

The `clock_gate_test_pin` attribute identifies an input pin connected to a `test_mode` or `scan_enable` signal. This attribute is used by Design Compiler when it compiles a design containing gated clocks that were introduced by Power Compiler.

Syntax

```
clock_gate_test_pin : true | false ;
```

true | false

A true value labels the pin as a test (`test_mode` or `scan_enable`) pin. A false value labels the pin as *not* a test pin.

Example

```
clock_gate_test_pin : true;
```

clock_gate_obs_pin Attribute

The `clock_gate_obs_pin` attribute identifies an output pin connected to an observability signal. Design Compiler uses the `clock_gate_obs_pin` attribute when it compiles a design containing gated clocks that were introduced by Power Compiler.

Syntax

```
clock_gate_obs_pin : true | false ;
```

true | false

A true value labels the pin as an observability pin. A false value labels the pin as *not* an observability pin.

Example

```
clock_gate_obs_pin : true;
```

clock_gate_out_pin Attribute

The `clock_gate_out_pin` attribute identifies an output port connected to an `enable_clock` signal. Design Compiler uses the `clock_gate_out_pin` attribute when it compiles a design containing gated clocks that were introduced by Power Compiler.

Syntax

```
clock_gate_out_pin : true | false ;
```

true | false

A true value labels the pin as a clock-gated output (`enable_clock`) pin. A false value labels the pin as *not* a clock-gated output pin.

Example

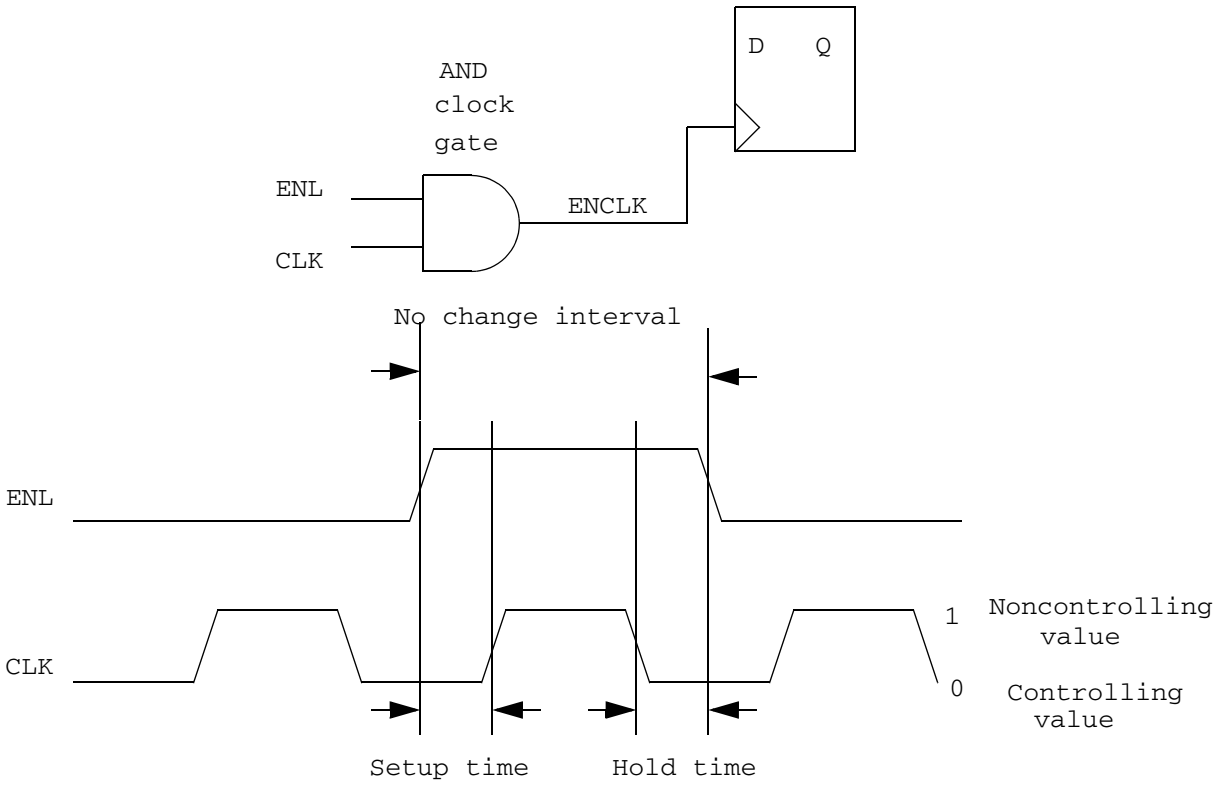
```
clock_gate_out_pin : true;
```

Clock-Gating Timing Considerations

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

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

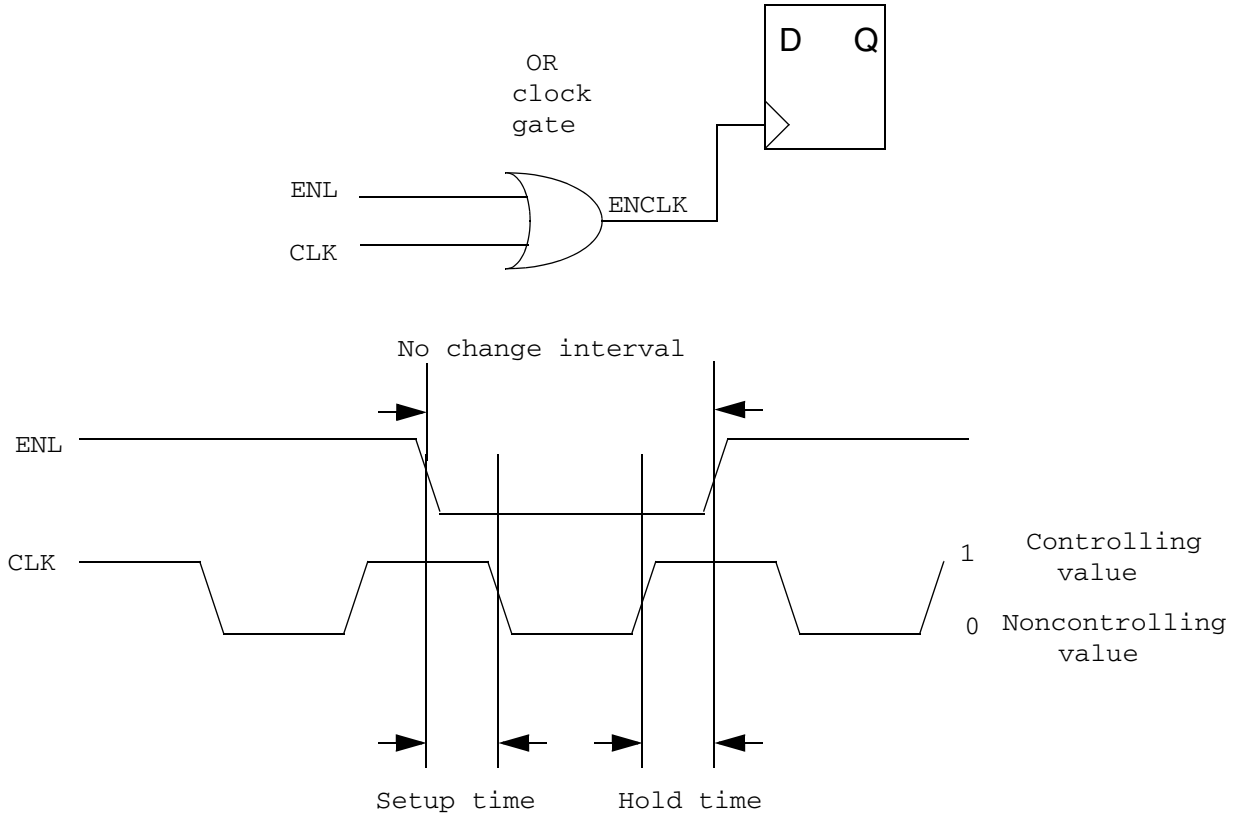
Figure 1-9 Setup and Hold Times for an AND Clock Gate



The setup time specifies how long the clock-gate input (ENL) must be stable before the clock input (CLK) makes a transition to a noncontrolling value. The hold time ensures that the clock-gate input (ENL) is stable for the time you specify after the clock input (CLK) returns to

a controlling value. The setup and hold times ensure that the ENL signal is stable for the entire time the CLK signal has a noncontrolling value, which prevents clipping or glitching of the ENCLK clock signal.

Figure 1-10 Setup and Hold Times for an OR Clock Gate



Timing Considerations for Integrated Cells

Clock gating requires certain timing arcs on your integrated cell.

For latch-based clock gating,

- Define setup and hold arcs on the enable pins with respect to the same controlling edge of the clock.
- Define combinational arcs from the clock and enable inputs to the output.

For latch-free clock gating,

- Define no-change arcs on the enable pins. These arcs must be no-change arcs, because they are defined with respect to different clock edges.

- Define combinational arcs from the clock and enable inputs to the output.

Table 1-4 specifies the setup and hold arcs required on the integrated cells. Set the setup and the hold arcs on the enable pin as specified by the `clock_gate_enable_pin` attribute with respect to the value entered for the `clock_gating_integrated_cell` attribute.

For the latch- and flip-flop-based styles, the setup and hold arcs are the conventional type and are set with respect to the same clock edge. However, for the latch-free style, the setup and hold arcs are set with respect to different clock edges and therefore must be specified as no-change arcs. Note that all arcs for integrated cells must be combinational arcs.

Table 1-4 Values of the clock_gating_integrated_cell Attribute

Value of `clock_gating_integrated_cell` **attributes**	**Setup arc**	**Hold arc**
`latch_posedge`	rising	rising
`latch_negedge`	falling	falling
`none_posedge`	falling	rising
`none_negedge`	rising	falling
`ff_posedge`	falling	falling
`ff_negedge`	rising	rising

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

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

Note:

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

Integrated Clock-Gating Cell Example

Example 1-19 shows what you might enter in setting up a cell for integrated clock gating. This example uses the `latch_posedge_precontrol_obs` value option for the `clock_gating_integrated_cell` attribute.

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

Example 1-19 Integrated Clock-Gating Cell

```
cell(CGLPCO) {
   area : 1;
   clock_gating_integrated_cell : "latch_posedge_precontrol_obs";
   dont_use : true;
   statetable(" CLK EN SE", "IQ") {
      table : " L  L  L : - : L ,\
              L  L  H : - : H ,\
              L  H  L : - : H ,\
              L  H  H : - : H ,\
              H  -  - : - : N ";
   }
   pin(IQ) {
      direction : internal;
      internal_node : "IQ";
   }
   pin(EN) {
      direction : input;
      capacitance : 0.017997;
      clock_gate_enable_pin : true;
      timing() {
         timing_type : setup_rising;
         intrinsic_rise : 0.4;
         intrinsic_fall : 0.4;
         related_pin : "CLK";
      }
      timing() {
         timing_type : hold_rising;
         intrinsic_rise : 0.4;
         intrinsic_fall : 0.4;
         related_pin : "CLK";
      }
   }
   pin(SE) {
      direction : input;
      capacitance : 0.017997;
      clock_gate_test_pin : true;
   }
   pin(CLK) {
      direction : input;
```

```
        capacitance : 0.031419;
        clock_gate_clock_pin : true;
        min_pulse_width_low  : 0.319;
    }
    pin(GCLK) {
        direction : output;
        state_function : "CLK * IQ";
        max_capacitance : 0.500;
        clock_gate_out_pin : true;
        timing() {
            timing_sense : positive_unate;
            intrinsic_rise : 0.48;
            intrinsic_fall : 0.77;
            rise_resistance : 0.1443;
            fall_resistance : 0.0523;
            slope_rise : 0.0;
            slope_fall : 0.0;
            related_pin : "EN CLK";
        }
     internal_power (){
            rise_power(li4X3){
                index_1("0.0150, 0.0400, 0.1050, 0.3550");
                index_2("0.050, 0.451, 1.501");
                values("0.141, 0.148, 0.256",\
                "0.162, 0.145, 0.234",\
                "0.192, 0.200, 0.284",\
                "0.199, 0.219, 0.297");
            }
            fall_power(li4X3){
                index_1("0.0150, 0.0400, 0.1050, 0.3550");
                index_2("0.050, 0.451, 1.500");
                values("0.141, 0.148, 0.256",\
                "0.162, 0.145, 0.234",\
                "0.192, 0.200, 0.284",\
                "0.199, 0.219, 0.297");
            }
            related_pin : "CLK EN" ;
        }
    }
    pin(OBS) {
        direction : output;
        state_function : "EN";
        max_capacitance : 0.500;
        clock_gate_obs_pin : true;
    }
}
```

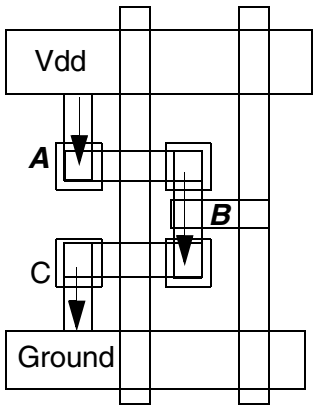
Modeling Electromigration

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

Overview

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

Figure 1-11 Electromigration in an Inverter



The electromigration failure strongly depends on the toggle rate.

The reliability of a metal line is most commonly described by the mean time to failure (MTTF), based on a set of experiments for a set of wires. The general expression for MTTF is

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

A

Material constant based on microstructure and geometric properties.

J

Average current density.

n

Constant whose value ranges between 1 and 3, representing the current density dependence.

Ea

Activation energy.

k

Boltzmann's constant.

T

Temperature in Kelvin.

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

Good library characterization is an essential first step toward solving the electromigration problem. For the purposes of verification, each cell output should be characterized with a multidimensional table, in which the dimensions are output load (C_l), average input transition times, rising and falling (T_{rf}), temperature in Kelvin (T), and the cell lifetime (L). Some of these parameters can be fixed at specific values, decreasing the number of dimensions in the characterization table. The most common table is the two-dimension table, C_l and T_{rf}. The stored values are the maximal toggle rates.

Scaling of the electromigration constraints is performed for different operating temperatures. For enabling temperature scaling, a new factor, `em_temp_exp_degradation`, must be defined in the library. Consider the exponential factor in the previous section, exp(nEa/kT); `em_temp_exp_degradation` represents its value for124 the nominal temperature. When the circuit operates in a nonnominal temperature, the electromigration constraints in the library will be scaled with the following coefficient:

$$\text{emp\_scale} = \exp(nE_a kT_{operating}) / \exp(nE_a kT_{nominal}) =$$

$$= \exp(nE_a kT_{operating}) / \text{em\_temp\_exp\_degradation} =$$

$$= \text{em\_temp\_exp\_degradation}^{T_{nominal} / T_{operating} - 1}$$

Controlling Electromigration

You can control electromigration by establishing an upper boundary for the output toggle rate. You achieve this by annotating cells with electromigration characterization tables representing the net's maximal toggle rates.

Specifically, do the following to control electromigration:

Use the em_lut_template group to name an index template to be used by the em_max_toggle_rate group, which you use to set the net's maximum toggle rates. The em_lut_template group has the variable_1 and variable_2 string attributes and the index_1 and index_2 complex attributes.

- Use the variable_1 string attribute to specify the first dimensional variable and the variable_2 string attribute to specify the second dimensional variable used by the library developer to characterize cells within the library for electromigration. You can assign input_transition_time or total_output_net_capacitance to variable_1 and variable_2.
- Use the index_1 complex attribute to specify the breakpoints along the variable_1 table axis, and use the index_2 complex attribute to specify the breakpoints along the variable_2 table axis.

The electromigration pin-level group attribute contains the em_max_toggle_rate group, which you use to specify the maximum toggle rate, and the related_pin and related_bus_pins attributes.

- Use the related_pin attribute to identify the input pin with which the electromigration group is associated.
- Use the related_bus_pins attribute to identify the bus group of the pin or pins with which the electromigration group is associated.
- Assign the same name for the em_max_toggle_rate pin-level group you specified for the em_lut_template library-level group whose template you want em_max_toggle_rate to use.

The em_max_toggle_rate pin-level group contains the values complex attribute; the related_pin and related_bus_pins simple attributes; and, optionally, the index_1 and index_2 complex attributes.

- Use the values complex attribute to specify the nets' maximum toggle rates for every index_1 breakpoint along the variable_1 axis if the table is one-dimensional.
- If the table is two-dimensional, use values to specify the nets' maximum toggle rates for all points where the breakpoints of index_1 intersect with the breakpoints of index_2. The value for these points is equal to nindex_1 x nindex_2, where index_1 and index_2 are the size to which the em_lut_template group's index_1 and index_2 attributes are set.
- You can also use the em_max_toggle_rate group's optional index_1 and index_2 attributes to overwrite the em_lut_template group's index_1 and index_2 values.
- State dependency in both lookup tables and polynomials.

Use the optional em_temp_degradation_factor at the library level or the cell level to specify the electromigration temperature exponential degradation factor. If this factor is not defined, the nominal temperature electromigration tables are used for all operating temperatures.

Use the report_lib -em command to get a report on electromigration for a specified library.

em_lut_template Group

Use the em_lut_template group at the library level to specify the name of the index template to be used by the em_max_toggle_rate group, which you use to set the net's maximum toggle rates to control electromigration.

Syntax

```
library (name_string) {
    em_lut_template(name_string) {
       variable_1: input_transition_time | total_output_net_capacitance ;
       variable_2: input_transition_time | total_output_net_capacitance ;
       index_1("float, ..., float");
       index_2("float, ..., float");
    }
}
```

variable_1 and variable_2 Simple Attributes

Use variable_1 to assign values to the first dimension and variable_2 to assign values to the second dimension for the templates on electromigration tables. The values can be either input_transition_time or total_output_net_capacitance.

Syntax

```
variable_1: input_transition_time |
total_output_net_capacitance ;
variable_2: input_transition_time |
total_output_net_capacitance ;
```

The value you assign to variable_1 is determined by how the index_1 complex attribute is measured, and the value you assign to variable_2 is determined by how the index_2 complex attribute is measured.

Assign input_transition_time for variable_1 if the complex attribute index_1 is measured with the input net transition time of the pin specified in the related_pin attribute or the pin associated with the electromigration group. Assign total_output_net_capacitance to variable_1 if the complex attribute index_1 is measured with the loading of the output net capacitance of the pin associated with the em_max_toggle_rate group.

Assign input_transition_time for variable_2 if the complex attribute index_2 is measured with the input net transition time of the pin specified in the related_pin attribute, in the related_bus_pins attribute, or in the pin associated with the electromigration group.

Assign total_output_net_capacitance to variable_2 if the complex attribute index_2 is measured with the loading of the output net capacitance of the pin associated with the electromigration group.

Example

```
variable_1 : total_output_net_capacitance ;
variable_2 : input_transition_time ;
```

index_1 and index_2 Complex Attributes

You can use the index_1 optional attribute to specify the breakpoints of the first dimension of an electromigration table used to characterize cells for electromigration within the library. You can use the index_2 optional attribute to specify the breakpoints of the second dimension of an electromigration table used to characterize cells for electromigration within the library.

Syntax

```
index_1 : ("float, ..., float") ;
index_2 : ("float, ..., float") ;
```

float

Floating-point numbers that identify the maximum toggle rate frequency from 1 to 0 and from 0 to 1.

You can overwrite the values entered for the em_lut_template group's index_1, by entering a value for the em_max_toggle_rate group's index_1. You can overwrite the values entered for the em_lut_template group's index_2, by entering a value for the em_max_toggle_rate group's index_2.

The following are the rules for the relationship between variables and indexes:

- If you have variable_1, you can have only index_1.
- If you have variable_1 and variable_2, you can have index_1 and index_2.
- The value you enter for variable_1 (used for one-dimensional tables) is determined by how index_1 is measured. The value you enter for variable_2 (used for two-dimensional tables) is determined by how index_2 is measured.

Example

```
index_1 ("0.0, 5.0, 20.0") ;
index_2 ("0.0, 1.0, 2.0") ;
```

electromigration Group

An `electromigration` group is defined in a `pin` group, as shown here:

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

Simple Attributes

```
related_pin : "name | name_list" ;/* path dependency */
related_bus_pins : "list of pins" ;/* list of pin names */
```

related_pin Simple Attribute

This attribute associates the `electromigration` group with a specific input pin. The input pin's input transition time is used as a variable in the electromigration lookup table.

If more than one input pin is specified in this attribute, the weighted input transition time of all input pins specified is used to index the electromigration table.

Syntax

```
related_pin : "name | name_list" ;
```

name | *name_list*

Name of input pin or pins.

Example

```
related_pin : "A B" ;
```

The pin or pins in the `related_pin` attribute denote the path dependency for the electromigration group. A particular `electromigration` group is accessed if the input pin or pins named in the `related_pin` attribute cause the corresponding output pin named in the `pin` group to toggle. All functionally related pins must be specified in a `related_pin` attribute if two-dimensional tables are being used.

Group Statements

```
em_max_toggle_rate (em_template_name) {}
```

These attributes and groups are described in the following sections.

related_bus_pins Simple Attribute

This attribute associates the `electromigration` group with the input pin or pins of a specific `bus` group. The input pin's input transition time is used as a variable in the electromigration lookup table.

If more than one input pin is specified in this attribute, the weighted input transition time of all input pins specified is used to index the electromigration table.

Syntax

```
related_bus_pins : "name1 [name2 name3 ... ]" ;
```

Example

```
related_bus_pins : "A" ;
```

The pin or pins in the `related_bus_pins` attribute denote the path dependency for the `electromigration` group. A particular `electromigration` group is accessed if the input pin or pins named in the `related_bus_pins` attribute cause the corresponding output pin named in the `pin` group to toggle. All functionally related pins must be specified in a `related_bus_pins` attribute if two-dimensional tables are being used.

em_max_toggle_rate Group

The `em_max_toggle_rate` group is a pin-level group that is defined within the electromigration `pin` group.

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

Complex Attributes

```
index_1 ("float, ..., float") ; /*this attribute is
optional*/
index_2 ("float, ..., float") ; /*this attribute is
optional*/
values ("float, ...,float ") ;
```

These attributes are defined in the following sections.

Index_1 and Index_2 Complex Attributes

You can use the index_1 optional attribute to specify the breakpoints of the first dimension of an electromigration table to characterize cells for electromigration within the library. You can use the index_2 optional attribute to specify breakpoints of the second dimension of an electromigration table used to characterize cells for electromigration within the library.

You can overwrite the values entered for the em_lut_template group's index_1, by entering values for the em_max_toggle_rate group's index_1. You can overwrite the values entered for the em_lut_template group's index_2, by entering values for the em_max_toggle_rate group's index_2.

Syntax

```
index_1 ("float, ..., float") ; /*this attribute is
optional*/
index_2 ("float, ..., float") ; /*this attribute is
optional*/
```

float

Floating-point numbers that identify the maximum toggle rate frequency.

Example

```
index_1 ("0.0, 5.0, 20.0") ;
index_2 ("0.0, 1.0, 2.0") ;
```

values Complex Attribute

This complex attribute is used to specify the net's maximum toggle rates.

This attribute can be a list of nindex_1 positive floating-point numbers if the table is one-dimensional.

This attribute can also be nindex_1 x nindex_2 positive floating-point numbers if the table is two-dimensional, where nindex_1 is the size of index_1 and nindex_2 is the size of index_2 that is specified for these two indexes in the em_max_toggle_rate group or in the em_lut_template group.

Syntax

```
values("float, ..., float") ;
```

float

Floating-point numbers that identify the maximum toggle rate frequency.

Example (One-Dimensional Table)

```
values : ("1.5, 1.0, 0.5") ;
```

Example (Two-Dimensional Table)

```
values : ("2.0, 1.0, 0.5", "1.5, 0.75, 0.33","1.0, 0.5,
```

```
0.15",) ;
```

em_temp_degradation_factor Simple Attribute

The `em_temp_degradation_factor` attribute specifies the electromigration exponential degradation factor.

Syntax

```
em_temp_degradation_factor : value_float ;
```

value

A floating-point number in centigrade units consistent with other temperature specifications throughout the library.

Example

```
em_temp_degradation_factor : 40.0 ;
```

Scalable Polynomial Electromigration Model

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

Syntax

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

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

Syntax

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

2

Advanced Low-Power Modeling

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

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

This chapter includes the following sections:

Power and Ground (PG) Pins

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

Important:

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

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

In order to overcome the limitations in the solution based on the `rail_connection` attribute and single ground pin, and to satisfy new Synopsys tool requirements, the syntax needs to be applied on all cells, including standard cells. The following section provides the general syntax for power and ground pin libraries in standard cells. The same syntax is also applicable to all other category of special cells.

Partial PG Pin Cell Modeling

Library Compiler supports partial PG pin cells. Partial PG pin cells are cells that have power PG pins only, ground PG pins only, or cells that do not have power or ground PG pins. Partial PG pin cells are defined in the following categories:

Table 2-1 Partial PG Pin Cell Categories

Partial PG Pin Cell	Description
Cells with one power pin and one ground pin	These cells are defined as having • 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

Table 2-1 Partial PG Pin Cell Categories (Continued)

Partial PG Pin Cell	Description
Cells with one power pin and no ground pins	These cells are defined as having • 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
Cells with no power pins and one ground pin	These cells are defined as having • 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 • 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 cells are acceptable with certain conditions. However, the tool issues a warning message and stores them in the .db file:

- ETM cells

  ETM cells do not need to have a pair of PG pins. Library Compiler generates a warning message for these cells, reporting the missing PG pin information, but it does not generate an error. A cell is identified as an ETM cell if it has either the `interface_timing` attribute or the `timing_model_type` attribute specified.

- Black box cells

  A black box cell with no timing, noise, or power information does not need to have at least one power pin and one ground PG pin.

- Metal fills and antenna cells

  Black box cells without functions do not need to have at least power pin and one ground pin.

- Cells without signal pins

  Cells without signal pins do not need to have at least one power pin and one ground PG pin.

- Load cells

  Cells without inout or output signal pins are required to have only one PG pin, either a power pin 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` o `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 can support the following attributes:

- To prevent a cell from being inserted automatically into the netlist, specify the `dont_touch` attribute 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 should not be added to a design during optimization.

- Library Compiler reports cells with partial PG pins by 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 2-1 shows a cell with only one primary ground pin.

Example 2-1 Partial PG Pin Cell With Only One Primary Ground Pin

```
cell (PULLDOWN) {
      pg_pin ( VSS ) {
          voltage_name : VSS;
          pg_type : primary_ground;
     }

    area : 1.0;
    dont_touch : true;
    dont_use : true;

    pin (X) {
      related_ground_pin : VSS;
      direction : output;
      function : "0";
      three_state : "!A";
      max_capacitance : 0.19;
    timing() {
      related_pin : "A";
      timing_type : three_state_enable;
      cell_rise(scalar) { values ( "0.1");}
      rise_transition(scalar) { values ( "0.1");}
      cell_fall(scalar) { values ( "0.1");}
      fall_transition(scalar) { values ( "0.1");}
    }
    timing() {
      related_pin : "A";
      timing_type : three_state_disable;
      cell_rise(scalar) { values ( "0.1");}
      rise_transition(scalar) { values ( "0.1");}
      cell_fall(scalar) { values ( "0.1");}
      fall_transition(scalar) { values ( "0.1");}
```

```
    }
    }
    pin (A) {
       related_ground_pin : VSS;
      direction : input;
      capacitance : 0.1;
      rise_capacitance : 0.1;
      rise_capacitance_range (0.1, 0.2);
      fall_capacitance : 0.1;
      fall_capacitance_range (0.1, 0.2);
    }
  }
```

Partial PG Pin Cell Checks

Library Compiler performs checks for partial PG pin cells and issues an error or warning message if certain conditions occur. For a detailed list of library checks for partial PG pin cells, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

PG Pin Syntax

The power and ground pin syntax for general cells is as follows:

```
library(library_name) {
  ...
  voltage_map(voltage_name, voltage_value);
  voltage_map(voltage_name, voltage_value);
  ...
  operating_conditions(oc_name) {
    ...
    voltage : value;
    ...
  }
  ...
  default_operating_conditions : oc_name;
  cell(cell_name) {
    pg_pin (pg_pin_name_p1) {
      voltage_name : voltage_name_p1;
      pg_type : type_value;
    }
    pg_pin (pg_pin_name_g1) {
      voltage_name : voltage_name_g1;
      pg_type : type_value;
    }
    pg_pin (pg_pin_name_p2) {
      voltage_name : voltage_name_p2;
      pg_type : type_value;
    }
    pg_pin (pg_pin_name_g2) {
```

```
      voltage_name : voltage_name_g2;
      pg_type : type_value;
    }
    ...
    leakage_power() {
      related_pg_pin : pg_pin_name_p1;
      ...
    }
    ...
    pin (pin_name1) {
      direction : input/inout;
      related_power_pin : pg_pin_name_p1;
      related_ground_pin : pg_pin_name_g1;
      ...
    }
    ...
    pin (pin_name2) {
      direction : inout/output;
      power_down_function : (!pg_pin_name_p1 + !pg_pin_name_p2 + \
                       !pg_pin_name_g1 + !pg_pin_name_g2) ;
      related_power_pin : pg_pin_namen_p2;
      related_ground_pin : pg_pin_namen_g2;
      internal_power() {
        related_pg_pin : pg_pin_name_p2;
       ...
       } /* end internal_power group */
       ...
     }/* end pin group*/
     ...
  }/* end cell group*/
  ...
}/* end library group*/
```

Library-Level Attributes

This section describes library-level attributes.

voltage_map Complex Attribute

The library-level `voltage_map` attribute associates the voltage name with the relative voltage values. These specified voltage names are referenced by the `pg_pin` groups defined at the cell level. If specified in a library, this syntax identifies the library to be a power and ground pin library.

default_operating_conditions Simple Attribute

The `default_operating_conditions` attribute is required to specify explicitly the default `operating_conditions` group in the library, which helps to identify the operating condition process, voltage, and temperature (PVT) points that are used during library characterization. At least one voltage map in the library should have a value of 0, which will become the reference value to which other voltage map values relate.

Cell-Level Attributes

This section describes cell-level attributes for `pg_pin` groups.

pg_pin Group

The `pg_pin` groups are used to represent the power and ground pins of a cell. Library cells can have multiple power and ground pins. The `pg_pin` groups are mandatory for each cell using the power and ground pin syntax, and a cell must have at least one `primary_power` power pin and at least one `primary_ground` ground pin.

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 a I/O cell, you must set the `is_pad` attribute to `true` in either a `pg_pin` group or on a signal pin for that cell. Otherwise, Library Compiler issues an error message. If the cell-level `pad_cell` attribute is specified on a I/O cell and there is no signal pin or PG pin in the pad cell, Library Complier issues a warning message.

voltage_name Simple Attribute

The `voltage_name` string attribute is mandatory in all `pg_pin` groups. The `voltage_name` attribute specifies the associated voltage name of the power and ground pin defined using the `voltage_map` complex attribute at the library level.

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 2-2 describes the pg_type values.

Table 2-2 pg_type Values

Value	Description
primary_power	Specifies that pg_pin is a primary power source (the default). If the pg_type attribute is not specified, primary_power is the pg_type value.
primary_ground	Specifies that pg_pin is a primary ground source.
backup_power	Specifies that pg_pin is a backup (secondary) power source (for retention registers, always-on logic, and so on).
backup_ground	Specifies that pg_pin is a backup (secondary) ground source (for retention registers, always-on logic, and so on).
internal_power	Specifies that pg_pin is an internal power source for switch cells.
internal_ground	Specifies that pg_pin is an internal ground source for switch cells.
pwell	Specifies regular p-wells for substrate-bias modeling.
nwell	Specifies regular n-wells for substrate-bias modeling.
deepnwell	Specifies isolation n-wells for substrate-bias modeling.
deeppwell	Specifies isolation p-wells for substrate-bias modeling.

physical_connection Simple Attribute

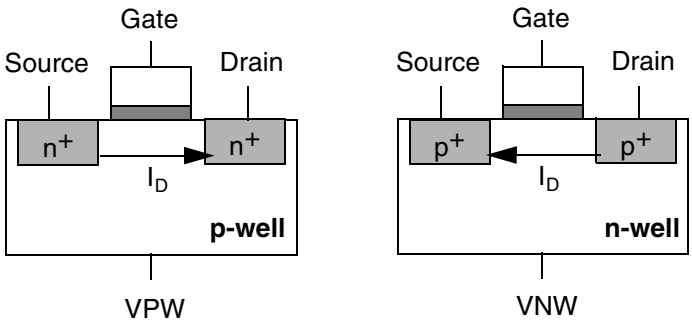
The physical_connection attribute can have the following values: device_layer and routing_pin. The device_layer value specifies that the bias connection is physically external to the cell. In this case, the library provides biasing tap cells that connect through the device layers. The routing_pin value specifies that the bias connection is inside a cell and is exported as a physical geometry and a routing pin. Macros with pin access generally use the routing_pin value if the cell has bias pins with geometry that is visible in the FRAM view.

related_bias_pin Attribute

The `related_bias_pin` attribute defines all bias pins associated with a power or ground pin within a cell. The `related_bias_pin` attribute is required only when the attribute is declared in a pin group but it does not specify a complete relationship between the bias pin and power and ground pin for a library cell.

The `related_bias_pin` attribute also defines all bias pins associated with a signal pin. To associate substrate-bias pins to signal pins, use the `related_bias_pin` attribute to specify one of the following `pg_type` values: `pwell`, `nwell`, `deeppwell`, `deepnwell`. Figure 2-1 shows transistors with p-well and n-well substrate-bias pins.

Figure 2-1 Transistors With p-well and n-well Substrate-Bias Pins



The `related_bias_pin` attribute can be specified in the `pg_pin` group. If you define the `related_bias_pin` attribute for signal pins only and do not specify the attribute in the `pg_pin` group, Library Compiler issues an error message. You do not need to specify bias PG pins again at the pin level because the `related_power_pin` and `related_ground_pin` attribute pairs that link a signal pin to primary, internal, or backup power and ground rails are already linked by the `related_bias_pin` attribute setting specified inside the `pg_pin` group.

user_pg_type Simple Attribute

The `user_pg_type` optional attribute allows you to customize the type of power and ground pin. It accepts any string value. The following example shows a `pg_pin` library with the `user_pg_type` attribute specified. The `user_pg_type` attribute must be specified with the `pg_type` attribute. If you do not specify `pg_type`, the power and ground pin value automatically defaults to `primary_power`.

```
pg_pin (pg_pin_name) {
   voltage_name : voltage_name;
   pg_type : < primary_power | primary_ground |
   backup_power | backup_ground |
   internal_power | internal_ground >
   user_pg_type : user_pg_type_name;
}
```

Pin-Level Attributes

This section describes pin-level attributes.

power_down_function Attribute

The `power_down_function` string attribute specifies the Boolean condition under which the cell's output pin is switched off by the state of the power and ground pins (when the cell is in off mode due to the external power pin states). If `power_down_function` is evaluated as 1, X is assumed on the pin, meaning that the pin is assumed to be in an unknown state.

You specify the `power_down_function` attribute for combinational and sequential cells. For simple and complex sequential cells, `power_down_function` also determines the condition of the cell's internal state. Knowing the sequential cell's internal state is necessary for formal verification tools when they perform equivalence checking because the internal state is what is verified.

For more information about using the `power_down_function` attribute for sequential cells, see the "Defining Sequential Cells" chapter in the *Library Compiler Methodology and Modeling Functionality in Technology Libraries User Guide*.

related_power_pin and related_ground_pin Attributes

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

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

- The primary power and primary ground pins are related to the signal pins. This behavior only applies to standard cells. For special cells, you must specify this relationship explicitly.
- The first `pg_pin` that has the `pg_type` attribute set to `primary_power` becomes the default value for `related_power_pin`.
- The first `pg_pin` that has the `pg_type` attribute set to `primary_ground` becomes the default value for `related_ground_pin`.

In a library based on power and ground pin syntax, the `pg_pin` groups are mandatory for each cell, and a cell must have at least one `primary_power` power pin and at least one `primary_ground` ground pin. Therefore, a default `related_power_pin` and `related_ground_pin` will always exist in any cell.

output_signal_level_low and output_signal_level_high Attributes

The `output_signal_level_low` and `output_signal_level_high` attributes can be defined at the pin level for the output pins and inout pins. The regular signal swings are derived for regular cells using the `related_power_pin` and `related_ground_pin` specifications.

input_signal_level_low and input_signal_level_high Attributes

The `input_signal_level_low` and `input_signal_level_high` attributes can be defined at the pin level for the input pins. The regular signal swings are derived for regular cells using the `related_power_pin` and `related_ground_pin` specifications.

related_pg_pin Attribute

The `related_pg_pin` attribute is used to associate a power and ground pin with leakage power and internal power tables. (The leakage power and internal power tables must be associated with the cell's power and ground pins.)

In the absence of a `related_pg_pin` attribute, the `internal_power` and `leakage_power` specifications apply to the whole cell (cell-specific power specification).

Table 2-3 lists the power and ground pin attributes and groups in the old syntax and maps them to the attributes and groups in the new syntax, introduced in the Y-2006.06 release.

Table 2-3 Power and Ground Pin Syntax Changes

Old syntax	New syntax
`nom_voltage` simple attribute	no change
`power_supply` group	`voltage_map` complex attribute
`rail_connection` complex attribute	`pg_pin` group
`power_level` simple attribute	`related_pg_pin` simple attribute
`input_voltage` / `output_voltage` groups	no change
`input_voltage` / `output_voltage` simple attributes	no change

Table 2-3 Power and Ground Pin Syntax Changes (Continued)

Old syntax	New syntax
`input_signal_level` simple attribute	`related_power_pin` and `related_ground_pin` attributes (plus `input_signal_level` to model overdrive cells)
`output_signal_level` simple attribute	`related_power_pin` and `related_ground_pin` simple attributes
`input_signal_level_low` / `input_signal_level_high` simple attributes	no change
`output_signal_level_low` / `output_signal_level_high` simple attributes	no change
`operating_condition` with `power_rail` attributes	The `power_rail` attribute no longer needs to be defined in the `operating_conditions` group

Naming Conventions for Power and Ground Pins in Logic Libraries

The `pg_pin` names must match the names of the power and ground pins in the physical library exactly in order to correctly link the physical and logical views. For example, if the name of your power and ground pin in the physical view is VDD for power and VSS for ground, the same names should be specified in your logical library, as shown in the following example:

```
pg_pin(VDD) {
...
}
and
pg_pin(VSS) {
...
}
```

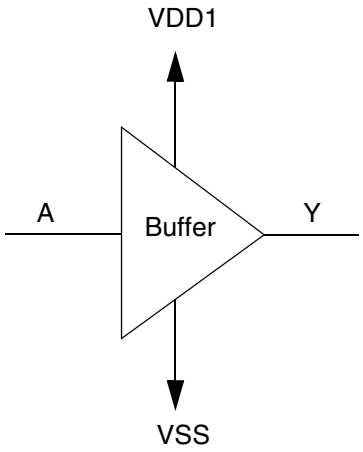
PG Pin Library Checks

Library Compiler performs checks for standard cells that are based on PG pin syntax and issues an error or warning message if certain conditions occur. For a detailed list of library checks for PG pin libraries, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Standard Cell With One Power and Ground Pin Example

Figure 2-2 shows a standard cell with a power and ground pin. The figure is followed by an example.

Figure 2-2 Standard Cell Buffer Schematic



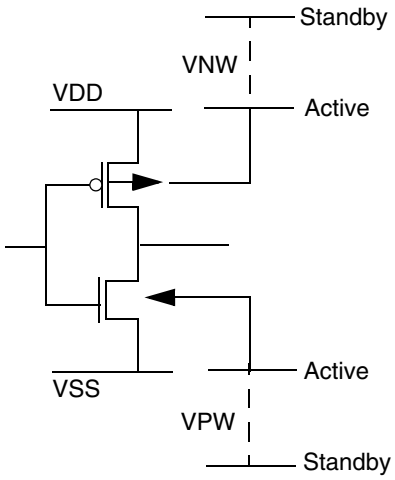

```
library(standard_cell_library_example) {

  voltage_map(VDD, 1.0);
  voltage_map(VSS, 0.0);

  operating_conditions(XYZ) {
    voltage : 1.0;
    ...
  }
  default_operating_conditions : XYZ;

  cell(BUF) {

    pg_pin(VDD) {
      voltage_name : VDD;
      pg_type : primary_power;
    }
    pg_pin(VSS) {
      voltage_name : VSS;
      pg_type : primary_ground;
    }

    leakage_power() {
      related_pg_pin : VDD;
      when : "!A";
     value : 1.5;
    }
```

```
    pin(A) {
      related_power_pin : VDD;
      related_ground_pin : VSS;
    }

    pin(Y) {
      direction : output;
      power_down_function : "!VDD + VSS";
      related_power_pin : VDD;
      related_ground_pin : VSS;

      timing() {
        related_pin : A;
        cell_rise(template) {
        ...
        }
        cell_fall(template) {
        ...
        }
        rise_transition(template) {
        ...
        }
        fall_transition(template) {
        ...
        }
       }
     internal_power() {
       related_pin : A;
       related_pg_pin : VDD;
       ...
     }
    }/* end pin group*/
    ...
  }/*end cell group*/
...
}/*end library group*/
```

Inverter With Substrate-Bias Pins Example

Figure 2-3 shows an inverter with substrate-bias pins. The figure is followed by an example.

Figure 2-3 Inverter With Substrate-Bias Pins




```
library(low_power_cells) {

  delay_model : table_lookup;

  /* unit attributes */
  time_unit : "1ns";
  voltage_unit : "1V";
  current_unit : "1mA";
  pulling_resistance_unit : "1kohm";
  leakage_power_unit : "1pW";
  capacitive_load_unit (1.0,pf);


voltage_map(VDD, 0.8);  /* primary power */
voltage_map(VSS, 0.0);   /* primary ground */
voltage_map(VNW, 0.8);  /* bias power */
voltage_map(VPW, 0.0);  /* bias ground */


  /* operation conditions */
  operating_conditions(XYZ) {
    process: 1;
    temperature: 125;
    voltage: 0.8;
    tree_type: balanced_tree
  }
  default_operating_conditions : XYZ;
```

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

  /* default attributes */
  default_leakage_power_density: 0.0;
  default_cell_leakage_power: 0.0;
  default_fanout_load: 1.0;
  default_output_pin_cap: 0.0;
  default_inout_pin_cap: 0.1;
  default_input_pin_cap: 0.1;
  default_max_transition: 1.0;

cell(std_cell_inv) {
  cell_footprint : inv;
  area : 1.0;
pg_pin(VDD) {
voltage_name : VDD;
pg_type : primary_power;
    related_bias_pin : "VNW";
}
pg_pin(VSS) {
voltage_name : VSS;
pg_type : primary_ground;
    related_bias_pin : "VPW";
}
pg_pin(VNW) {
voltage_name : VNW;
pg_type : nwell;
physical_connection : device_layer;
}
pg_pin(VPW) {
voltage_name : VPW;
pg_type : pwell;
physical_connection : device_layer;
}

  pin(A) {
    direction : input;
    capacitance : 1.0;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    related_bias_pin : "VPW VNW";
  }
  pin(Y) {
    direction : output;
    function : "A'";
```

```
        related_power_pin : VDD;
        related_ground_pin : VSS;
        related_bias_pin : "VPW VNW";
        power_down_function : "!VDD + VSS + VNW' + VPW";
        internal_power() {
        related_pg_pin : VDD;
          related_pin : "A";
          rise_power(scalar) { values ( "1.0");}
          fall_power(scalar) { values ( "1.0");}
        }
        timing() {
          related_pin : "A";
          timing_sense : positive_unate;
          cell_rise(scalar) { values ( "0.1");}
          rise_transition(scalar) { values ( "0.1");}
          cell_fall(scalar) { values ( "0.1");}
          fall_transition(scalar) { values ( "0.1");}
        }
        max_capacitance : 0.1;
      }
      cell_leakage_power : 1.0;
      leakage_power() {
        when :"!A";
        value : 1.5;
      }
      leakage_power() {
        when :"A";
        value : 0.5;
      }
    }
    }
```

Defining Power Data for Multiple-Rail Cells

The following example shows how to specify Liberty syntax to define power information for multiple-rail cells:

```
library(multi_rail_library) {
  delay_model : table_lookup;

  /* unit attributes */
  time_unit : "1ns";
  voltage_unit : "1V";
  current_unit : "1mA";
  pulling_resistance_unit : "1kohm";
  leakage_power_unit : "1pW";
  capacitive_load_unit (1.0,pf);


  /* power supply definition */
    voltage_map (VDD1, 1.0);
    voltage_map (VDD2, 2.0);
```

```
    voltage_map (VSS, 0.0);
    operating_conditions(XYZ) {
      process  : 1.0;
      voltage : 1.0;
    temperature : 25.0;
    }
    default_operating_conditions : XYZ;

  cell (multi_rail_cell) {
    pg_pin (VDD1) {
    voltage_name : VDD1 ;
    pg_type : primary_power ;
    }

    pg_pin (VDD2) {
    voltage_name : VDD2 ;
    pg_type : primary_power ;
    }

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

  /* Specify the same number of when states for each rail. Otherwise,
  Library Compiler will generate a parsing error. */

  leakage_power() {
    related_pg_pin : "VDD1";
    when : "!A1 !A2" ;
    value : 1.5 ;
  }
  leakage_power() {
    related_pg_pin : "VDD1";
    when : "!A1 A2" ;
    value : 2.0 ;
  }
  leakage_power() {
    related_pg_pin : "VDD1";
    when : "A1 !A2" ;
    value : 3.0 ;
  }
  leakage_power() {
    related_pg_pin : "VDD2";
    when : "!A1 !A2" ;
    value : 3.5 ;
  }
  leakage_power() {
    related_pg_pin : "VDD2";
    when : "!A1 A2" ;
    value : 3.0 ;
  }
  leakage_power() {
    related_pg_pin : "VDD2";
    when : "A1 !A2" ;
    value : 4.0 ;
  }
```

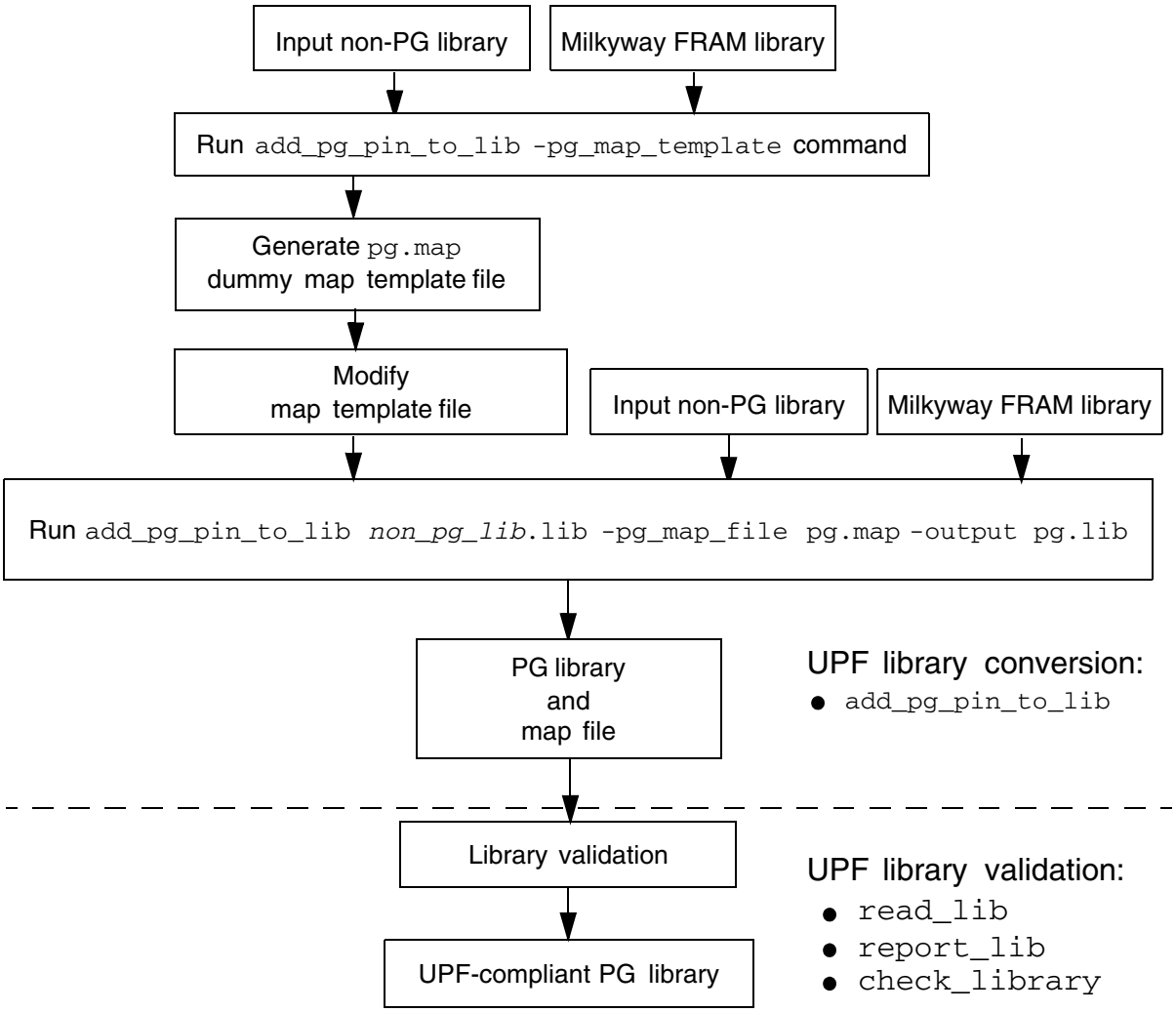
```
/* Default leakage power specification for each of the rails missing the
when states above */

leakage_power() {
  related_pg_pin : "VDD1";
  value : 3.0 ;
}
leakage_power() {
  related_pg_pin : "VDD2";
  value : 4.0 ;
}
  area : 1.0 ;
  pin(A1) {
    direction : input;
    capacitance : 0.1 ;
    related_power_pin : VDD1;
    related_ground_pin : VSS;
  }
  pin(A2) {
    direction : input;
    capacitance : 0.1 ;
    related_power_pin : VDD1;
    related_ground_pin : VSS;
  }
  pin(ZN) {
    direction : output;
    max_capacitance : 0.1;
    function : "(A1*A2)'";
    related_power_pin : VDD2;
    related_ground_pin : VSS;
  timing() {
    related_pin : "A1"
    cell_rise( scalar ) {
      values("0.0"); }
    rise_transition( scalar ) {
      values("0.0"); }
    cell_fall( scalar ) {
      values("0.0"); }
    fall_transition( scalar ) {
      values("0.0"); }
    }
    internal_power() {
      related_pin : " A1 "
      related_pg_pin : "VDD1";
      rise_power( scalar ) {
       values("0.0"); }
     fall_power( scalar ) {
       values("0.0"); }
    }
    /* end of internal power */

    internal_power() {
      related_pin : " A1 "
      related_pg_pin : "VDD2";
      rise_power( scalar ) {
        values("0.0"); }
      fall_power( scalar ) {
        values("0.0"); }
```

```
      }
      /* end of internal power */

      timing() {
        related_pin : "A2"
        cell_rise( scalar ) {
          values("0.0"); }
        rise_transition( scalar ) {
          values("0.0"); }
        cell_fall( scalar ) {
          values("0.0"); }
        fall_transition( scalar ) {
          values("0.0"); }
        }
        internal_power() {
        related_pin : " A2 "
        related_pg_pin : "VDD1";
        rise_power( scalar ) {
          values("0.0"); }
        fall_power( scalar ) {
          values("0.0"); }
        }
        /* end of internal power */

        internal_power() {
          related_pin : " A2 "
          related_pg_pin : "VDD2";
          rise_power( scalar ) {
            values("0.0"); }
          fall_power( scalar ) {
            values("0.0"); }
        } /* end of internal power */
    } /* end of pin group */
} /* end of cell */
```

Generating Libraries With PG Pin Syntax

You can convert a library that is not based on PG pin syntax to a library with PG pin syntax by using the `add_pg_pin_to_lib` command. The converted library is UPF ready. You can then validate the library using library validation commands. Figure 2-4 highlights the steps that are necessary to convert a library using `add_pg_pin_to_lib` and validate the library using library validation commands.

Figure 2-4 Flow for Converting and Validating PG Pin Libraries



The `add_pg_pin_to_lib` command is available at the Library Compiler, Design Compiler, and IC Compiler command prompt. The command converts and updates a logic library automatically from the old `rail_connection` syntax or from a format that is not based on PG pin syntax to a library with PG pin syntax. You must input the logic library that is not based on PG pin syntax and the related physical libraries with a Milkyway FRAM view, if available, and also a PG map file (side file).

Note:

The `add_pg_pin_to_lib` command is the same as the `add_pg_pin_to_db` command except that the library file that you input when you use `add_pg_pin_to_lib` is a .lib ASCII file whereas the input file for the `add_pg_pin_to_db` command is a .db binary source file. In all other ways, the `add_pg_pin_to_db` and `add_pg_pin_to_lib` commands have the same options and values and use the same flows.

Supporting Partial PG Pin Cells

The `add_pg_pin_to_lib` command supports partial PG pin cells. Partial PG pin cells are cells that have power PG pins only, ground PG pins only, or cells that do not have power or ground PG pins. For more information about partial PG pin cells, see "Partial PG Pin Cell Modeling" on page 2-2.

Using the Fast Flow

You can use the `-fast` option to generate a PG library from a non-PG library, using the information available in a FRAM view, the input non-PG library, or a combination of both a logic and physical library. You do not need the Milkyway FRAM views; however, the FRAM views help identify the names of the cell's PG pins. If you do not supply a FRAM view, the option can only infer the pin names if they are specified in the logic library through certain attributes, such as the `rail_connection` attribute.

add_pg_pin_to_lib Syntax

The following section describes the `add_pg_pin_to_lib` command syntax:

```
add_pg_pin_to_lib input_lib_filename
 [-mw_library_name mw_lib_name]
 [-pg_map_file pg_map_filename.map]
 [-pg_map_template pg.map_template_filename]
 [-expanded]
 [-verbose]
 [-fast]
 [-common_shell_path common_shell_path]
-output pg_lib_filename
```

Arguments

input_lib_filename

Specifies the name of the input logic library (.lib) file that is not based on PG pin syntax. You must specify the input logic library file to run the `add_pg_pin_to_lib` command. If the `search_path` environment variable is set, only the file name is required; otherwise, you must specify a file name with the full path.

`-mw_library_name` *mw_lib_name*

Specifies one or more Milkyway library names or FRAM views that correspond to the input logical .db file. You must specify either the `-mw_library_name` option or the `-pg_map_file` option for single rail cells. If the cells in the library are multiple-rail cells, you must specify the `-pg_map_file` option. You do not need to specify the `-mw_library_name` option if the PG map file is complete; however, if the map file is not

complete, you must specify `-mw_library_name`. If the `search_path` environment variable is set, the file name is required only; otherwise, you must specify a file name with the full path.

`-pg_map_file` *`pg_map_filename.map`*

Specifies the file name for mapping between data that is not based on PG pin syntax and data that is based on PG pin syntax. You must specify either the `-pg_map_file` option or the `-mw_library_name` option for single rail cells. If the cells in the library are multiple-rail cells, you must specify both the `-mw_library_name` and `-pg_map_file` options. If you specify the map file name only, the map file is generated in the current working directory. To save it to another location, you must specify the full path to the map file location.

For information about the PG map file format, see the *Adding PG Pin Syntax to Logical Libraries Application Note* in the *UPF Library Preparation Application Notes* documentation on SolvNet.

`-pg_map_template` *`pg.map_template_filename`*

Specifies the file name for the PG map file to be automatically generated. You can use this option if you want to quickly generate a PG map file template to use as the first step for building the final PG map file. Later, you can use this PG map file template to complete the final conversion step with the `-pg_map_file` option. When you specify the `-pg_map_template` option, you cannot use the `-pg_map_file` option during the same run.

If you specify the map file name only, the map file is generated in the current working directory. To save it to another location, you must specify the full path to the map file location.

You can use the `-pg_map_template` option with the input non-PG library file only or with the `-mw_library_name` option with the Milkyway FRAM view. You should include Milkyway FRAM views during map template generation so that the `add_pg_pin_to_lib` command can generate a nearly complete PG map file. This is because PG information that is missing from the logical library can be derived from the Milkyway views during PG map file generation, creating a more complete PG map file.

`-expanded`

Generates a PG map template file with the wildcards expanded. By default, the `add_pg_pin_to_lib` command tries to generate a PG map file that has as many wildcards as possible in compressed format in order to reduce the file size. However, if you want to debug the PG map file for any issues, you can use the `-expanded` option to generate a PG map file with all the wildcards expanded.

`-verbose`

Outputs all sections of the PG map data in a log file, whether the PG map data is generated automatically or expanded from the map file.

`-fast`

Allows you to quickly generate a PG library from a non-PG library, using the information available in a FRAM view, the input non-PG library, or a combination of both a logical and physical library.

`-common_shell_path` *`shell_path`*

Specifies the path where lc_shell, dc_shell, or icc_shell is located. Use the `-common_shell_path` option if you want to use a specific release version for compiling the library.

`-output` *`pg_lib_filename`*

Specifies the name of the new PG pin-based file that is generated after the `add_pg_pin_to_lib` conversion. You must specify the file name. No default name is used for the file. If you specify the file name only and not the full path, the file is generated in the current working directory. To save it to another location, you must specify the full path.

For more information about the `add_pg_pin_to_lib` command, including the PG map file formats, usage model recommendations, recommended flows, information about validating PG pin libraries, and `check_library` checks, see the *Adding PG Pin Syntax to Logical Libraries Application Note* in the *UPF Library Preparation Application Notes* documentation on SolvNet.

Level-Shifter Cells in a Multivoltage Design

Level-shifter cells (also known as buffer type level shifters) and isolation cells are cells used to connect the netlist in the different voltage domains in order to meet the design constraints. Level-shifter insertion is one of the most important aspects of multivoltage optimization flow. In multivoltage and shut-down designs, both level shifting and isolation are required. An enable level shifter fulfills both these requirements because it can function as an isolation cell or as a level shifter.

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

- Which power and ground pin of the level shifter is used for voltage boundary hookup during level shifter insertion. This information allows the optimization tools to determine on which side of the voltage boundary a particular level shifter is allowed.
- Which voltage conversions the particular level shifter can handle. Specifically, does the level shifter work for conversion from high voltage to low voltage (HL), from low voltage to high voltage (LH), or both (HL_LH)?

- What the input and output voltage ranges are for a level shifter under all operating conditions.

Operating Voltages

In a multivoltage design, each design instance can operate at its specified operating voltage. Therefore, it is important that each voltage correspond to one or more logical hierarchies. All cells in a hierarchy operate at the same voltage except for level shifters.

The operating voltages are annotated on the top design, design instances, or hierarchical ports through PVT operating conditions.

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.

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

- Two power supplies
- Specified input and output voltages

The functionality of level shifters includes

- Identifying nets in the design that need voltage adjustments
- Analyzing the target library for the availability of level shifters
- Ripping the net and instantiating level shifters where appropriate

Level Shifter Syntax

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

  Note:

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

output_voltage_range Attribute

The `output_voltage_range` attribute is similar to the `input_voltage_range` attribute except that it specifies the allowed voltage range of the level shifter for the output pin instead of the input pin.

The `output_voltage_range` syntax differs from the pin-level `output_signal_level_low` and `output_signal_level_high` syntax in the following ways:

- The `output_signal_level_low` and `output_signal_level_high` attributes are defined on the output pins under one operating condition (the default operating condition of the library).

- The `output_signal_level_low` and `output_signal_level_high` attributes are used to specify the partial voltage swing of an output pin (that is, to specify only partial swings rather than the full rail-to-rail swing). Note that `output_voltage_range` is not related to the voltage swing.

Note:

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

Pin-Level Attributes

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

std_cell_main_rail Attribute

The `std_cell_main_rail` Boolean attribute is defined in a `primary_power` power pin. When the attribute is set to true, the `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` simple attribute identifies a data pin of a level-shifter cell. The valid values are true or false. The attribute is set to false by default, meaning that the pin is a regular signal pin of the level shifter cell.

level_shifter_enable_pin Attribute

The `level_shifter_enable_pin` attribute identifies an enable pin of a level-shifter cell. The valid values are true or false. The attribute is set to false by default, meaning that the pin is a regular signal pin of the level-shifter cell.

input_voltage_range and output_voltage_range Attributes

The `input_voltage_range` and `output_voltage_range` attributes are used to specify the allowed voltage ranges of the input or an output pin of the level-shifter cell. The attributes define two floating values where the first value is the lower bound and the second value is the upper bound.

Note:

The pin-level attribute specifications always override the cell-level specifications.

input_signal_level Attribute

The `input_signal_level` attribute is defined at the pin level and is used to specify which signal is driving the input pin. The attribute defines special overdrive cells that do not have a physical relationship with the power and ground on input pins.

If the `input_signal_level` and the `related_power_pin` and `related_ground_pin` attributes are defined on any input pin, the full voltage swing derived from the `input_signal_level` attribute takes precedence over the voltage swing derived from the `related_power_pin` and `related_ground_pin` attributes during timing calculations.

power_down_function Attribute

The `power_down_function` string attribute identifies the Boolean condition under which the cell's signal output pin is switched off (when the cell is in off mode due to the external power pin states). If the `power_down_function` is set to 1, then X is assumed on the pin.

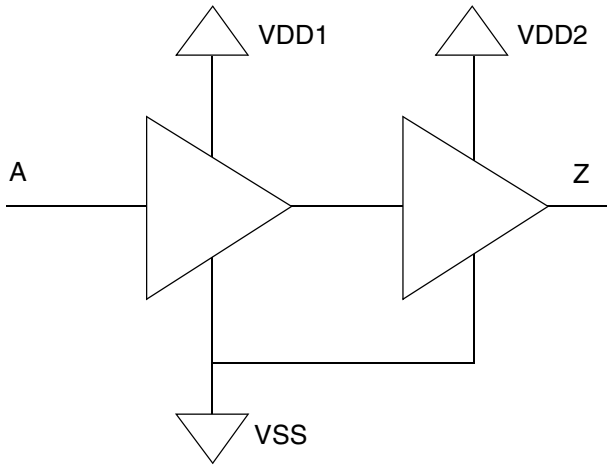
Level Shifter Modeling Examples

The following sections provide examples for modeling a simple buffer type low-to-high level-shifter cell, a simple buffer type high-to-low level-shifter cell, an overdrive high-to-low level-shifter cell, an enable level-shifter cell, and an enable isolation cell.

Simple Buffer Type Low-to-High Level Shifter

Figure 2-5 shows a simple 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 2-5 Buffer Type Low-to-High Level-Shifter Cell




```
library(level_shifter_cell_library_example) {
```

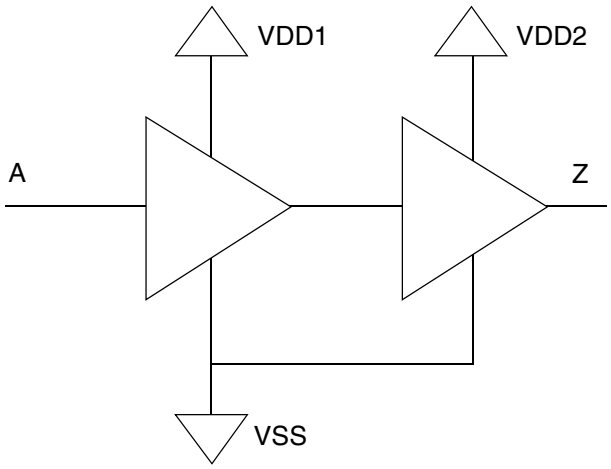
```
voltage_map(VDD1, 0.8);
voltage_map(VDD2, 1.2);
voltage_map(VSS, 0.0);
operating_conditions(XYZ) {
  process  : 1.0;
  voltage : 3.0;
temperature : 25.0;
}
default_operating_conditions : XYZ;

cell(Buffer_Type_LH_Level_shifter) {
  is_level_shifter : true;
  level_shifter_type :  LH ;

  pg_pin(VDD1) {
    voltage_name : VDD1;
    pg_type : primary_power;
    std_cell_main_rail : true;
  }
  pg_pin(VDD2) {
    voltage_name : VDD2;
    pg_type : primary_power;
  }
  pg_pin(VSS) {
    voltage_name : VSS;
    pg_type : primary_ground;
  }

  leakage_power() {
    when : "!A";
    value : 1.5;
    related_pg_pin : VDD1;
  }
  leakage_power() {
    when : "!A";
    value : 2.7;
    related_pg_pin : VDD2;
  }

  pin(A) {
    direction : input;
    related_power_pin : VDD1;
    related_ground_pin : VSS;
    input_voltage_range ( 0.7 , 0.9);
  }

  pin(Z) {
    direction : output;
    related_power_pin : VDD2;
    related_ground_pin : VSS;
    function : "A";
    power_down_function : "!VDD1 + !VDD2 + VSS";
    output_voltage_range (1.1 , 1.3);
```

```
      timing() {
        related_pin : A;
        cell_rise(template) {
        ...
        }
        cell_fall(template) {
        ...
        }
        rise_transition(template) {
        ...
        }
        fall_transition(template) {
        ...
        }
      }

      internal_power() {
        related_pin : A;
        related_pg_pin : VDD1;
        ...
      }
      internal_power() {
        related_pin : A;
        related_pg_pin : VDD2;
       ...
      }

    } /* end pin group*/
  ...
  } /*end cell group*/
  ...
} /*end library group*/
```

Simple Buffer Type High-to-Low Level Shifter

Figure 2-6 shows a simple 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 2-6 Buffer Type High-to-Low Level-Shifter Cell




```
library(level_shifter_cell_library_example) {
  voltage_map(VDD1, 1.2);
  voltage_map(VDD2, 0.8);
  voltage_map(VSS, 0.0);
  operating_conditions(XYZ) {
    process  : 1.0;
    voltage : 3.0;
  temperature : 25.0;
  }
  default_operating_conditions : XYZ;

  cell(Buffer_Type_HL_Level_shifter) {
    is_level_shifter : true;
    level_shifter_type :  HL ;

    pg_pin(VDD1) {
      voltage_name : VDD1;
      pg_type : primary_power;
      std_cell_main_rail : true;
    }
    pg_pin(VDD2) {
      voltage_name : VDD2;
      pg_type : primary_power;
    }
    pg_pin(VSS) {
      voltage_name : VSS;
      pg_type : primary_ground;
    }

    leakage_power() {
      when : "!A";
      value : 1.5;
      related_pg_pin : VDD1;
    }
```

```
    leakage_power() {
      when : "!A";
      value : 2.7;
      related_pg_pin : VDD2;
    }

    pin(A) {
      direction : input;
      related_power_pin : VDD1;
      related_ground_pin : VSS;
      input_voltage_range ( 0.7 , 0.9);
    }

    pin(Z) {
      direction : output;
      related_power_pin : VDD2;
      related_ground_pin : VSS;
      function : "A";
      power_down_function : "!VDD1 + !VDD2 + VSS";
      output_voltage_range (1.1 , 1.3);

      timing() {
        related_pin : A;
        cell_rise(template) {
        ...
        }
        cell_fall(template) {
        ...
        }
        rise_transition(template) {
        ...
        }
        fall_transition(template) {
        ...
        }
      }

      internal_power() {
        related_pin : A;
        related_pg_pin : VDD1;
        ...
      }
      internal_power() {
        related_pin : A;
        related_pg_pin : VDD2;
       ...
      }

    } /* end pin group*/
  ...
  } /*end cell group*/
  ...
} /*end library group*/
```

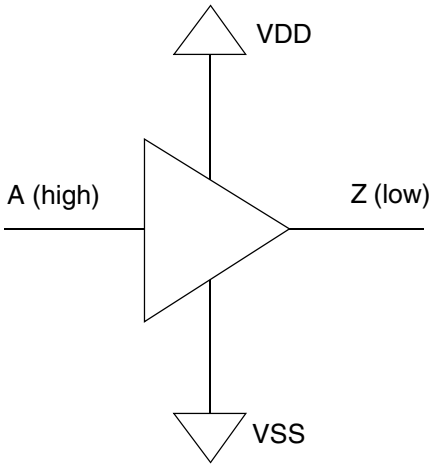
Overdrive Level-Shifter Cell

The cell in Figure 2-7 is known as an overdrive level-shifter cell. To model an overdrive level-shifter cell, specify the `related_ground_pin` attribute and the `input_signal_level` attribute, as shown in the example that follows the figure.

Note:

The area of a multiple-rail level-shifter cell (as shown in Figure 2-6) is larger than that of an overdrive level-shifter cell (as shown in Figure 2-7). Keep the area in mind when you design your cell.

Figure 2-7 Overdrive Level-Shifter Cell



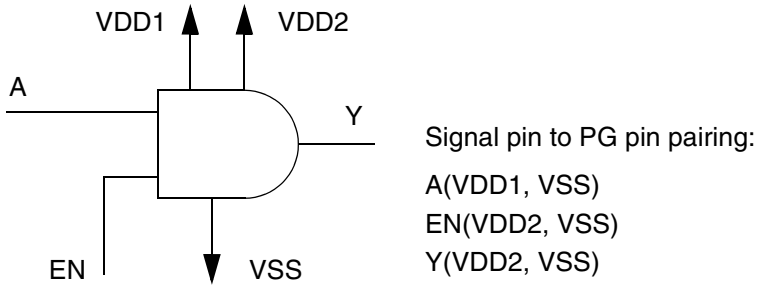

```
library(level_shifter_cell_library_example) {
  voltage_map(VDD1, 1.2);
  voltage_map(VDD, 0.8);
  voltage_map(VSS, 0.0);
  operating_conditions(XYZ) {
    process  : 1.0;
    voltage : 3.0;
    temperature : 25.0;
  }
  default_operating_conditions : XYZ;

  cell(Buffer_Type_HL_Level_shifter) {
    is_level_shifter : true;
    level_shifter_type :  HL ;

    pg_pin(VDD) {
      voltage_name : VDD;
      pg_type : primary_power;
      std_cell_main_rail : true;
    }
    pg_pin(VSS) {
      voltage_name : VSS;
```

```
      pg_type : primary_ground;
    }
    leakage_power() {
      when : "!A";
      value : 1.5;
      related_pg_pin : VDD;
    }

    pin(A) {
      direction : input;
/* Defining the input_signal_level attribute identifies an Overdrive
Level Shifter cell */
      input_signal_level : VDD1;
      related_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 */
```

Enable Level-Shifter Cell

Enable level shifters shift voltage levels between and locks the output to a logic value. Figure 2-8 shows an enable level-shifter cell schematic. In the figure, 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. Note that the enable pin is linked to VDD2. The example that follows the figure shows a library containing an enable level shifter.

Figure 2-8 Enable Level-Shifter Cell Schematic




```
library(enable_level_shifter_library_example) {

  voltage_map(VDD1, 0.8);
  voltage_map(VDD2, 1.2);
  voltage_map(VSS, 0.0);

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

  cell(Enable_Level_Shifter) {
    is_level_shifter : true;
    level_shifter_type :  LH ;

    pg_pin(VDD1) {
      voltage_name : VDD1;
      pg_type : primary_power;
      std_cell_main_rail : true;
    }
    pg_pin(VSS) {
      voltage_name : VSS;
      pg_type : primary_ground;
    }
    pg_pin(VDD2) {
      voltage_name : VDD2;
      pg_type : primary_power;
```

```
}


leakage_power() {
  when : "!A";
  value : 1.5;
  related_pg_pin : VDD1;
}
leakage_power() {
  when : "!A";
  value : 1.5;
  related_pg_pin : VDD2;
}

pin(A) {
  direction : input;
  related_power_pin : VDD1;
  related_ground_pin : VSS;
  input_voltage_range ( 0.7 , 0.9);
  level_shifter_data_pin : true;
}

pin(EN) {
  direction : input;
  related_power_pin : VDD2;
  related_ground_pin : VSS;
  input_voltage_range ( 1.1 , 1.3);
  level_shifter_enable_pin : true;
}

pin(Y) {
  direction : output;
  related_power_pin : VDD2;
  related_ground_pin : VSS;
  function : "A * EN";
  power_down_function : "!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*/
```

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 (sample_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 ( vnw1 ) {
  pg_type : nwell ;
  …
}
pg_pin ( vnw2 ) {
  pg_type : nwell ;
  …
}
pin ( I ) {
  direction : input;
  related_power_pin : vdd1
```

```
  related_ground_pin : vss
  related_bias_pin : "vnw1 vpw"
  …
}
pin ( Z ) {
  direction : output;
  function : "I";
  related_power_pin : "vdd2" ;
  related_ground_pin : "vss" ;
  related_ bias_pin : "vnw2 vpw";
  power_down_function : "!vdd1 + !vdd2 + vss + !vnw1 + !vnw2 + vpw";
  …
}
   } /* End of cell group */
}/* End of library group */
```

Level-Shifter Library Checks

Library Compiler performs checks for level-shifter cells and issues an error or warning message if certain conditions occur. For a detailed list of library checks for level-shifter cells, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Isolation Cell Modeling

In multiple-supply designs and multivoltage with shut-down designs, the outputs from the shut-down partition into the active partition must be maintained at predictable signal levels. This signal isolation is achieved by using an isolation cell. The isolation logic ensures that all inputs to the active partition are clamped to a fixed value.

Isolation Cell Syntax

The syntax for isolation cells is as follows:

```
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 ;
    use_iso_rail_for_buffering = true ;
    ...
  }
...
  pin (output) {
    direction : output;
    use_iso_rail_for_buffering = true ;
    function : "Boolean Expression";
    power_down_function : "Boolean Expression";
    ...
  }/* End pin group */

}/* End Cell group */
```

Cell-Level Attributes

This section describes cell-level attributes for 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 the same as any ordinary standard cell.

Pin-Level Attributes

This section describes pin-level attributes for isolation cells.

isolation_cell_data_pin Attribute

The `isolation_cell_data_pin` optional attribute identifies the data pin of any isolation cell. The valid values are true or false. If not specified, all the isolation cell's input pins default to a data pin.

isolation_cell_enable_pin Attribute

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

power_down_function Attribute

The `power_down_function` string attribute identifies the Boolean condition under which the cell's output pin is switched off (when the cell is in off mode due to the external power pin states). If the `power_down_function` is set to 1, then X is assumed on the pin.

permit_power_down Attribute

The `permit_power_down` string attribute indicates that the power pin may be powered down while in the isolation mode. The valid values are true or false. If not specified, the default is true, meaning that the power pin can be power down while in isolation mode.

use_iso_rail_for_buffering Attribute

The `use_iso_rail_for_buffering` string attribute indicates that, when connecting a driver or the receivers (depending on the “direction” of the pin), actual connected power and ground rails are not considered as is; UPF isolation supply set for power-state reference is considered. Actual power and ground rails may have been optimized for further power reduction. Indirectly, the `use_iso_rail_for_buffering` attribute indicates that partial power down of the cell is permitted with respect to the pin. The valid values are true or false. If not specified, the default is true, meaning that UPF isolation supply set for power-state reference is considered.

Attribute Dependencies

The isolation cell attributes have the following dependencies:

- An isolation control, when activated, causes the output to become a constant.
- An isolation control that permits partial power down must appear in the `power_down_function` attribute of the same output, and it blocks the terms involving the powered-down rail from evaluating to true. (The other power rail that is still “alive” provides the output value in the isolation mode).

Hence, an isolation cell cannot use partial power down if the `power_down_function` expression of its power pin does not involve any pins with the `isolation_cell_enable_pin` attribute set.

Note:

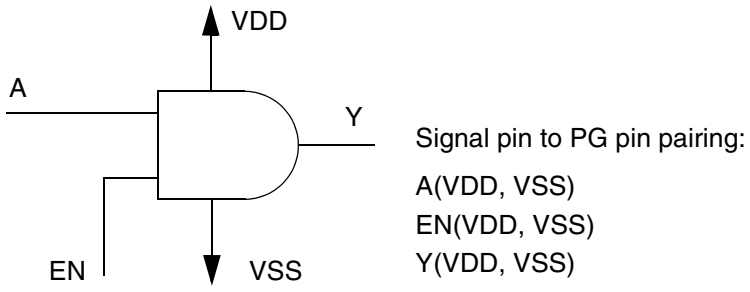
Currently, Library Compiler allows only one pin with the `isolation_cell_enable_pin` attribute in accordance to IEEE Standard 1801 but allows other non-isolation type enable pins to co-exist in the cell.

Isolation Cell Example

Isolation cells cannot shift the voltage levels like level-shifters cell can. All other characteristics are the same between a level-shifter cell and an isolation cell.

Figure 2-9 shows an isolation cell schematic. The library in this example describes only the portion related to the power and ground pin syntax. In the figure, the A, EN, and Y signal pins are all linked to the VDD and VSS power and ground pin pair. The example that follows the figure shows a library containing an isolation cell.

Figure 2-9 Isolation Cell Schematic




```
library(isolation_cell_library_example) {

  voltage_map(VDD, 1.0);
  voltage_map(VSS , 0.0);

  operating_conditions(XYZ) {
    voltage : 1.0;
    ...
  }
  default_operating_conditions : XYZ;

cell(Isolation_Cell) {
is_isolation_cell : true;
dont_touch : true;
dont_use : true;

pg_pin(VDD) {
  voltage_name : VDD;
  pg_type : primary_power;
}
```

```
pg_pin(VSS) {
  voltage_name : VSS;
  pg_type : primary_ground;
}

leakage_power() {
  when : "!A";
  value : 1.5;
  related_pg_pin : VDD;
}

pin(A) {
  direction : input;
  related_power_pin : VDD;
  related_ground_pin : VSS;
  isolation_cell_data_pin : true;
}

pin(EN) {
  direction : input;
  related_power_pin : VDD;
  related_ground_pin : VSS;
  isolation_cell_enable_pin : true;
}
pin(Y) {
  direction : output;
  related_power_pin : VDD;
  related_ground_pin : VSS;
  function : "A * EN";
  power_down_function : "!VDD + VSS";
  timing() {
    related_pin : "A EN";
    cell_rise(template) {
    ...
    }
    cell_fall(template) {
    ...
    }
    rise_transition(template) {
    ...
    }
    fall_transition(template) {
    ...
    }
  }
  internal_power() {
    related_pin : A;
    related_pg_pin : VDD;
    ...
  }

    }/* end pin group*/
```

```
    ...
  }/*end cell group*/
  ...
}/*end library group*/
```

Isolation Cell Library Checks

Library Compiler performs checks for isolation cells and issues an error or warning message if certain conditions occur. For a detailed list of library checks for isolation cells, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Switch Cell Modeling

Switch cells, also known as multithreshold complementary metal oxide semiconductor cells (or multithreshold CMOS 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, which control power nets based on a PMOS transistor, and footer switch cells, which control ground nets based on a NMOS transistor. This type of cell is a switch that drives the power to other logic cells. It is used as a big switch to the supply rails and to turn off design partitions when the relative logic is inactive. Therefore, coarse-grain switch cells can reduce the leakage of the inactive logic.

  In addition, coarse-grain switch cells can also have the properties of a fine-grain switch cell. For example, they can act as a switch and have output pins that might or might not be shut off by the internal switch pins.

- Fine grain

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

  Currently, Library Compiler supports only fine-grain switch cells for macro cells.

Coarse-Grain Switch Cells

Coarse-grain switch cells must have the following properties:

- They must be able to model the condition under which the cell turns off the controlled design partition or the cells connected in the output power pin's fanout logical cone. This is modeled with a switching function based on special switch signal pins as well as a separate but related power-down function based on power pin inputs.
- They must be able to model the "acknowledge" output pins (output pins whose signal is used to propagate the switch signal to the next-switch cell or to a power controller input's logic cloud). Timing is also propagated from the input switch pin to the acknowledge output pins.
- They must have at least one virtual output power and ground pin (virtual VDD or virtual VSS), one regular input power and ground pin (VDD or VSS), and one switch input pin. There is no limit on the number of pins a coarse-grain cell can have.

  The following describes a simple coarse-grain switch header cell and a simple coarse-grain switch footer cell:

  - Header cell: one switch input pin, one VDD (power) input power and ground pin, and one virtual VDD (internal power) output power and ground pin
  - Footer cell: one switch input pin, one virtual VSS (internal ground) output power and ground pin, and one VSS (ground) input power and ground pin
- They can have multiple switch pins and multiple acknowledge output pins.
- They must have the steady state current (I/V) information to determine the resistance value when the switch is on.
- The timing information can be specified for coarse-grain switch cells on the output pins, and it can be state dependent for switch pins.

The power output pins in a coarse-grain switch cell can have the following two states:

- Awake/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` models the templates for the steady state current information later modeled inside the `dc_current` group. The `input_voltage` variable specifies the switch pin's input voltage values, and the `output_voltage` variable specifies the output voltage values. 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 multithreshold-CMOS switch cell.

The `dc_current` group, which is used for steady state current modeling, must be defined at the cell level. It is defined by two indexes: `index_1` and `index_2`. The `index_1` attribute represents a string that includes a comma-separated set of N values, where N represents the table rows. The values define the voltage levels measured at either the input voltage or the output voltage. When referring to the input voltage, the voltage level is measured at the related switch pin, and the set of N values must have at least two values for N. When referring to the output voltage, the voltage level is measured at the related internal PG pin, and the set of N values must have at least three values for N. This voltage level is related to a common reference ground for the cell. The set of N `index_1` values must increase monotonically or Library Compiler issues an error.

The `index_2` attribute represents a string that includes a comma-separated set of M values, where M represents the table columns. The values define the voltage levels that are specified at either the input voltage or the output voltage. When referring to the input voltage, the voltage level is measured at the related switch pin, and the set of M values must have at least two values for M. When referring to the output voltage, the voltage level is measured at the related internal PG pin, and the set of M values must have at least three values for M. This voltage level is related to a common reference ground for the cell. The set of M `index_2` values must increase monotonically or Library Compiler issues an error.

Cell-Level Attribute

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

switch_cell_type Attribute

The `switch_cell_type` attribute provides a complete description of the switch cell so that Library Compiler does not need to infer the switch cell type from the cell modeling description. The valid enumerated values for this attribute 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 syntax is modified from the Boolean function attribute. In addition to its existing usage for signal output pins, it is used for the coarse-grain switch cells' virtual VDD output pins to represent the propagated power level through the switch as a function of the input power and ground pins. This is usually a logical buffer and is useful in cases where the VDD and VSS connectivity might be erroneously reversed.

In coarse grain switch cells, the pg_function attribute is specified inside the pg_pin group.

power_down_function Attribute

The power_down_function string attribute is used to identify the condition under which the cell's signal output pin is switched off (when the cell is in off mode due to the external power pin states). If power_down_function is set to 1, then X is assumed on the pin.

Note:

If certain signal pins are not associated to a power and ground pin (using the related_power_pin and the related_ground_pin attributes), Library Compiler associates those signal pins to the primary power rails defined on the cell and issues the following warning message:

```
Warning: Line 1641, Connect pin 'XYZ' to the default power
pg_pin 'VDD'. (LBDB-725)
```

pg_pin Group

The cell-level pg_pin group is used to model the VDD and VSS pins and virtual VDD and VSS pins of a coarse-grain switch cell. The syntax is based on the Y-2006.06 power and ground pin syntax.

Fine-Grained Switch Support for Macro Cells

A macro cell with a fine-grained switch is a cell that contains a special switch transistor with a control pin that can turn off the power supply of the cell when it is idle. This significantly lowers the power consumption of a design.

With the growing popularity of low-power designs, macro cells with fine-grained switches play an important role. The syntax identifies a cell as a macro cell and specifies the correct power pin that supplies power to each signal pin.

Macro Cell With Fine-Grained Switch Syntax

```
cell(<cell_name>) {
         is_macro_cell : true;
         switch_cell_type : coarse_grain | fine_grain;

  pg_pin ( <power/ground pin name> ) {
    pg_type : primary_power | primary_ground | backup_power |
backup_ground;
    direction: input | inout | output;
    …
  }

/* This is a special pg pin that uses "switch_function" to describe the
logic to shut
off the attached design partition */
  pg_pin ( <internal power/ground pin name>) {
    direction: internal | input | output | inout;
    pg_type : internal_power | internal_ground;
    switch_function : "<function_string>";
    pg_function : "<function_string>";
    …
  }

   pin (<input_pin_name>) {
     direction : input |  inout;
     switch_pin : true | false;
     …
   }
            …
   pin(<output_pin_name>) {
     direction : output | inout;
     power_down_function : <function_string>;
     …
   } /* end pin group */
} /* end cell group */
```

Cell-Level Attributes

The following attributes are cell-level attributes for macro cells with fine-grained switches.

is_macro_cell Attribute

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

switch_cell_type Attribute

The `switch_cell_type` attribute is enhanced to support macro cells with internal switches. The valid enumerated values for this attribute are `coarse_grain` and `fine_grain`.

pg_pin Group

The following attribute can be specified under the `pg_pin` group for macro cells with fine-grained switches.

direction Attribute

The `direction` attribute supports `internal` as a valid value for macros when the internal power and ground is not visible or accessible at the cell boundary.

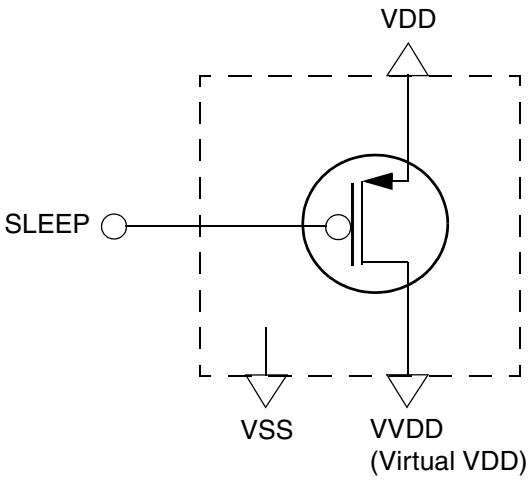
Switch-Cell Modeling Examples

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

Simple Coarse-Grain Header Switch Cell

Figure 2-10 and the example that follows it show a simple coarse-grain header switch cell.

Figure 2-10 Simple Coarse-Grain Header Switch Cell



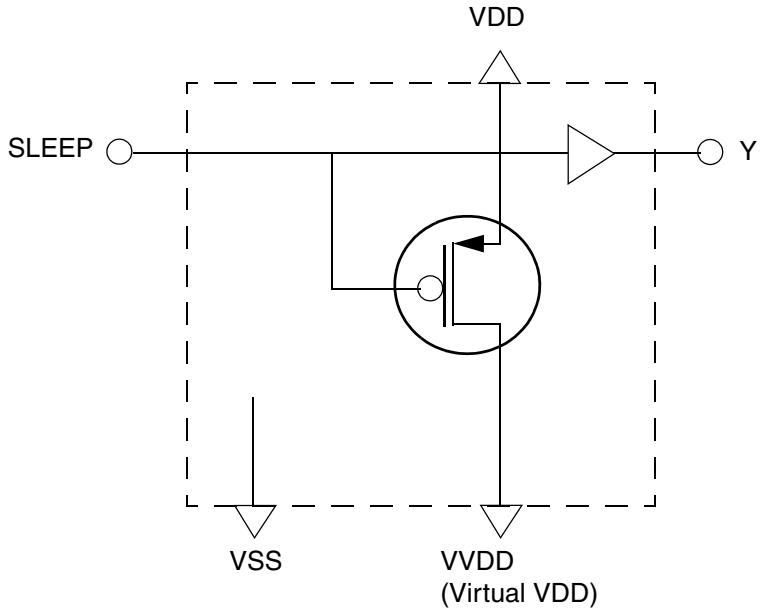

```
library (simple_coarse_grain_lib) {

  …
  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 2-11 and the example that follows it show a complex coarse-grain header switch cell.

Figure 2-11 Complex Coarse-Grain Header Switch Cell



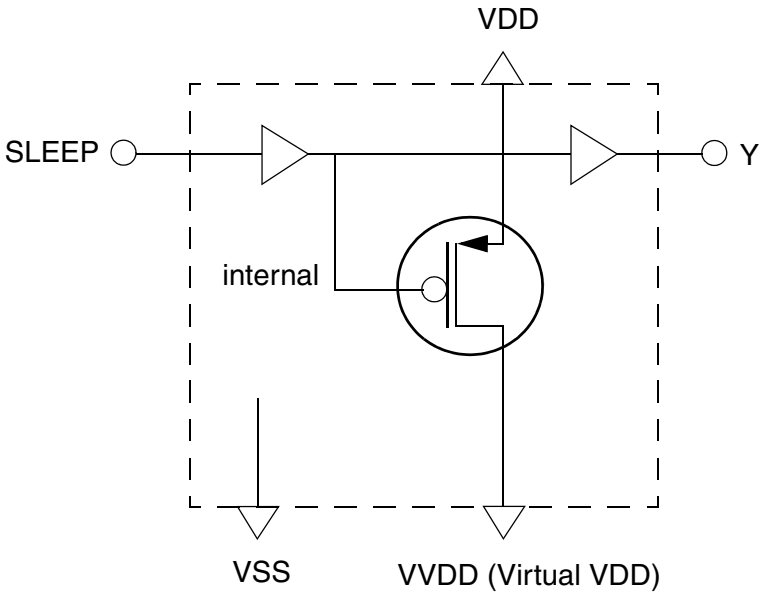

```
library (complex_coarse_grain_lib) {
  ...
  current_unit : 1mA;
  ...
  voltage_map(VDD,  1.0);
  voltage_map(VVDD, 0.8);
  voltage_map(VSS,  0.0);

  operating_conditions(XYZ) {
    process  : 1.0;
    voltage : 1.0;
    temperature : 25.0;
  }
  default_operating_conditions : XYZ;

  lu_table_template ( ivt1 ) {
  variable_1 : input_voltage;
  variable_2 : output_voltage;
  index_1 ( "0.1, 0.2, 0.4, 0.8, 1.0" );
  index_2 ( "0.1, 0.2, 0.4, 0.8, 1.0" );
  }
  ...
  cell ( Complex_CG_Switch ) {
    ...
    switch_cell_type : coarse_grain;
    pg_pin ( VDD ) {
      pg_type : primary_power;
      voltage_name : VDD;
```

```
      direction: input ;
  }
     pg_pin ( VVDD ) {
       pg_type : internal_power;
       direction : output ;
       voltage_name : VVDD;
       switch_function : "SLEEP";
       pg_function : "VDD" ;
     }
     pg_pin ( VSS ) {
       pg_type : primary_ground;
       voltage_name : VSS;
       direction : input ;
     }

     /* I/V curve information */
     dc_current ( ivt1 ) {
       related_switch_pin : SLEEP;     /* control pin */
       related_pg_pin : VDD;           /* source power pin */
       related_internal_pg_pin : VVDD; /* drain internal power pin*/
       values(     "0.010, 0.020, 0.030, 0.040, 0.050", \
                   "0.011, 0.021, 0.031, 0.041, 0.051", \
                   "0.012, 0.022, 0.032, 0.042, 0.052", \
                   "0.013, 0.023, 0.033, 0.043, 0.053", \
                   "0.014, 0.024, 0.034, 0.044, 0.054");
              }

     pin ( SLEEP ) {
       direction : input;
       switch_pin : true;
       capacitance: 1.0;
       related_power_pin : VDD;
       related_ground_pin : VSS;
       ...
     }
     ...
     pin (Y) {
       direction : output;
       function : "SLEEP";
       related_power_pin : VDD;
       related_ground_pin : VSS;
       power_down_function : "!VDD + VSS";
       timing() {
         related_pin : SLEEP;
         ...
       }
     } /* end pin group */
   } /* end cell group*/
  } /* end library group*/
```

Complex Coarse-Grain Switch Cell With an Internal Switch Pin

Figure 2-12 and the example that follows it show a complex coarse-grain switch cell with an internal switch pin.

Figure 2-12 Complex Coarse-Grain Switch Cell With an Internal Switch Pin




```
library (Complex_CG_lib) {
  …
  current_unit : 1mA;
  …

  voltage_map(VDD, 1.0);
  voltage_map(VVDD, 0.8);
  voltage_map(VSS, 0.0);

  operating_conditions(XYZ) {
    process  : 1.0;
    voltage : 1.0;
    temperature : 25.0;
  }
  default_operating_conditions : XYZ;

  lu_table_template ( ivt1 ) {
  variable_1 : input_voltage;
  variable_2 : output_voltage;
  index_1 ( "0.1, 0.2, 0.4, 0.8, 1.0" );
  index_2 ( "0.1, 0.2, 0.4, 0.8, 1.0" );
  }
  …
```

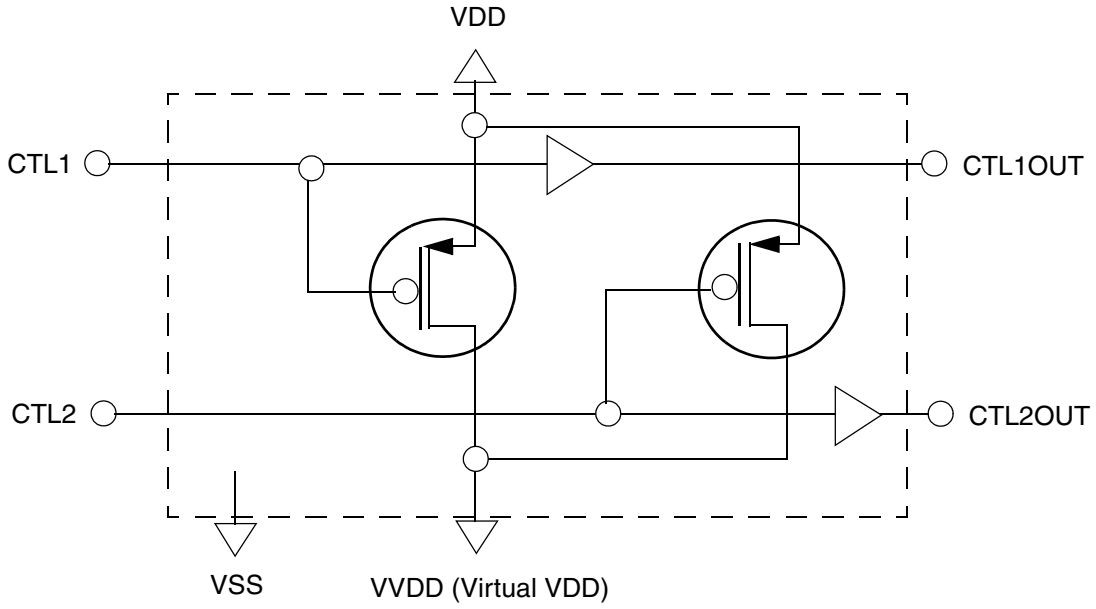
```
cell (COMPLEX_HEADER_WITH_INTERNAL_SWITCH_PIN) {
  cell_footprint : complex_mtpmos;
  area : 1.0;
  switch_cell_type : coarse_grain;
pg_pin(VDD) {
  voltage_name : VDD;
  pg_type : primary_power;
  direction : input;
}
pg_pin(VVDD) {
  voltage_name : VVDD;
  pg_type : internal_power;
  direction : output;
  switch_function : "SLEEP" ;
}
pg_pin(VSS) {
  voltage_name : VSS;
  pg_type : primary_ground;
  direction : input;
}

dc_current(ivt1) {
  related_switch_pin : internal;
  related_pg_pin : VDD;
  related_internal_pg_pin : VVDD;
  values(     "0.010, 0.020, 0.030, 0.040, 0.050", \
              "0.011, 0.021, 0.031, 0.041, 0.051", \
              "0.012, 0.022, 0.032, 0.042, 0.052", \
              "0.013, 0.023, 0.033, 0.043, 0.053", \
              "0.014, 0.024, 0.034, 0.044, 0.054");
        }

 pin(SLEEP) {
   switch_pin : true;
   direction : input;
   capacitance : 1.0;
   related_power_pin : VDD;
   related_ground_pin : VSS;
   …
 }

 pin(Y) {
   direction : output;
   function : "SLEEP";
   related_power_pin : VDD;
   related_ground_pin : VSS;
   power_down_function : "!VDD + VSS";

   timing() {
     related_pin : "SLEEP"
     …
   }
 } /* end pin group */
```

```
   pin(internal) {
     direction : internal;
     timing() {
     related_pin : "SLEEP"
     …
   }
  } /* end pin group */
 } /* end cell group */
} /* end library group */
```

Complex Coarse-Grain Switch Cell With Parallel Switches

Figure 2-13 and the example that follows it show a complex coarse-grain switch cell with two parallel switches.

Figure 2-13 Complex Coarse-Grain Switch Cell With Two Parallel Switches




```
library (Complex_CG_lib) {
  …
  current_unit : 1mA;
  …

  voltage_map(VDD, 1.0);
  voltage_map(VVDD, 0.8);
  voltage_map(VSS, 0.0);

  operating_conditions(XYZ) {
    process  : 1.0;
    voltage : 1.0;
    temperature : 25.0;
```

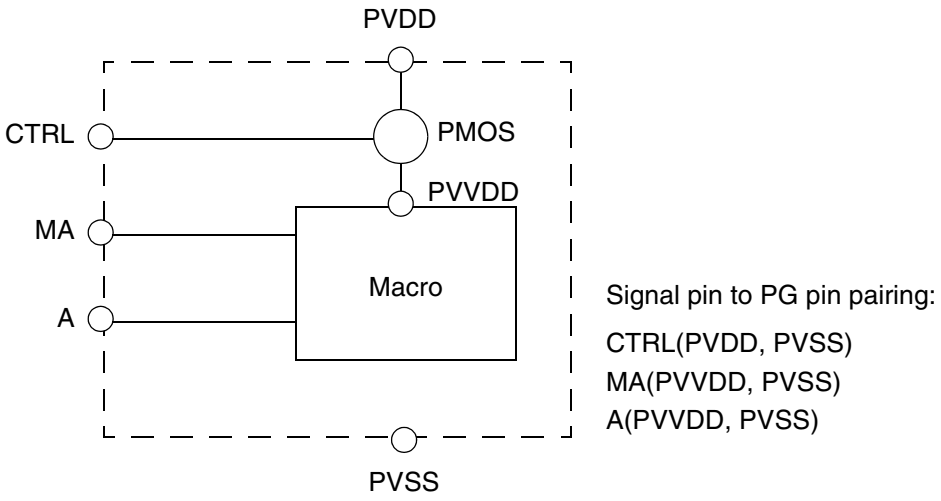
```
  }
  default_operating_conditions : XYZ;

  lu_table_template ( ivt1 ) {
  variable_1 : input_voltage;
  variable_2 : output_voltage;
  index_1 ( "0.1, 0.2, 0.4, 0.8, 1.0" );
  index_2 ( "0.1, 0.2, 0.4, 0.8, 1.0" );
  }
  …

  cell (COMPLEX_HEADER_WITH_TWO_PARALLEL_SWITCHES) {
    cell_footprint : complex_mtpmos;
    area : 1.0;
    switch_cell_type : coarse_grain;

  pg_pin(VDD) {
    voltage_name : VDD;
    pg_type : primary_power;
    direction : input;
  }
  pg_pin(VVDD) {
    voltage_name : VVDD;
    pg_type : internal_power;
    direction : output;
    switch_function : "CTL1 + CTL2" ;
  }
  pg_pin(VSS) {
    voltage_name : VSS;
    pg_type : primary_ground;
    direction : input;
}

  dc_current(ivt1) {
    related_switch_pin : CTL1;
    related_pg_pin : VDD;
    related_internal_pg_pin : VVDD;
    values(   "0.010, 0.020, 0.030, 0.040, 0.050", \
              "0.011, 0.021, 0.031, 0.041, 0.051", \
              "0.012, 0.022, 0.032, 0.042, 0.052", \
              "0.013, 0.023, 0.033, 0.043, 0.053", \
              "0.014, 0.024, 0.034, 0.044, 0.054");
     }

  dc_current(ivt1) {
    related_switch_pin : CTL2;
    related_pg_pin : VDD;
    related_internal_pg_pin : VVDD;
    values(   "0.010, 0.020, 0.030, 0.040, 0.050", \
              "0.011, 0.021, 0.031, 0.041, 0.051", \
              "0.012, 0.022, 0.032, 0.042, 0.052", \
              "0.013, 0.023, 0.033, 0.043, 0.053", \
              "0.014, 0.024, 0.034, 0.044, 0.054");
```

```
    }

  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 */
```

Macro Cell With Fine-Grained Internal Power Switches

Figure 2-14 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 2-14 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;
    …
    }
}
```

Switch Cell Library Checks

Library Compiler performs checks for switch cells and issues an error or warning message if certain conditions occur. For a detailed list of library checks for switch cells, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Retention Cell Modeling

Note:

The retention cell modeling information in this section is based on the D-2010.03 syntax. For information about retention cell modeling for libraries created prior to the D-2010.03 release, see the *Liberty Syntax for Retention Cell Modeling Application Note, Version A-2007.12* posted on SolvNet.

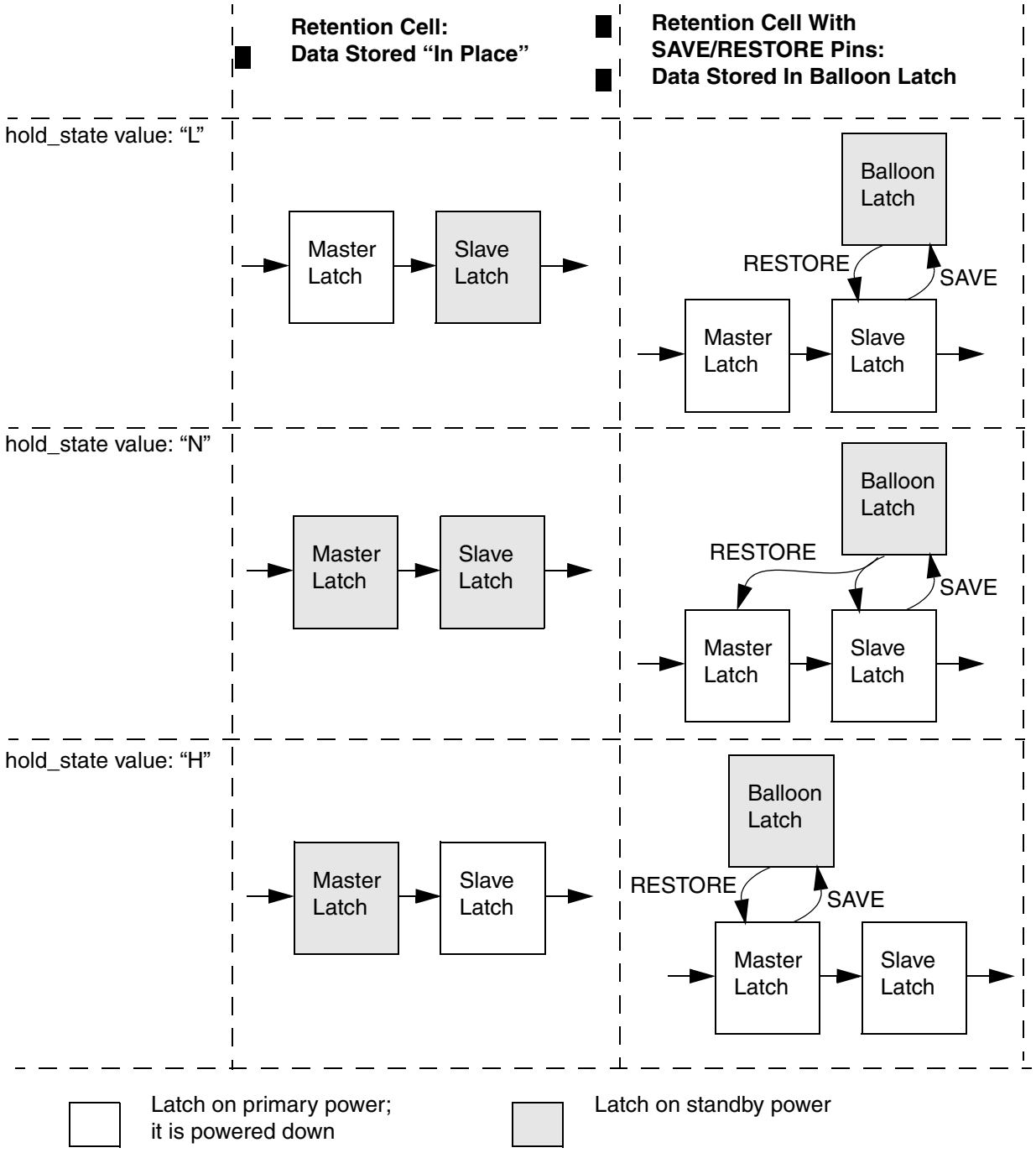
Since the goal of designs in the VDSM era hinge heavily on higher density and performance but lower power consumption, CMOS technology has been scaled down to reduce power consumption of designs. But scaling of voltage comes at a heavy price namely increase in sub threshold leakage power of CMOS technologies since to achieve performance the threshold voltages of CMOS circuits have also been scaled down considerably.

Leakage current is present in both active and stand-by modes. Leakage reduction of circuits can be achieved by shutting down the power supply for the low thresholds transistors in the designs but the state of designs will be lost during such power down cycles. Since designs need to wake up in the same state when they were shut down certain design elements need to have the capability to store the states of logic and the best and most easy way to achieve the same is to use the retention cells in designs.

Retention cells are sequential cells that can hold their internal state when the primary power supply is shut down and restore the state when the power is brought up. They consist of register logic and the retention circuitry that is supplied with a different power supply that must be active to maintain memory of the saved state. Retention cells reduce leakage current in standby mode (also sometimes called sleep mode or retention mode) without affecting the performance of the design during normal operation. Power is saved by instantiating retention registers, which rely on low-threshold transistors for performance during normal operation, and high-threshold, low-leakage transistors for saving the register state during standby mode.

Retention cells generally fall into two major categories namely store in-place where the slave or the master latch will store the state of the cell when the main power controlling the low threshold transistors are shut-down or have a feedback loop involving a balloon latch structure (which generally comprise of a latch with special control logic with high threshold transistors) and a regular flip-flop or latch (which comprise of low threshold transistors that will be powered down) to save and restore the cells state. A feedback loop in Balloon latch types of retention structures is a path from the output of the balloon latch back to its own input or internal node or to another flip-flop or latch. The two main categories of retention structure classifications are highlighted in Figure 2-15. The retention portion of the logic generally consists of high-threshold transistors and is powered by backup power.

Figure 2-15 Retention Cell Structures



Modeling Retention Cells

Retention registers generally include one of the following elements: a regular flip-flop with a slave or master latch that stores the data in place during the retention operation or a regular flip-flop with an extra balloon latch with special control logic that stores the state of the cell when it is powered down. The retention, or always-on, latch is generally implemented with high-threshold transistors. The always-on part of the logic is powered by a backup power supply. In addition to the separate power supplies, signals, such as save and restore signals, are required to control the storage of the data in the always-on logic or to enable the data transfer from the regular flip-flop to the balloon latch and back again, based on the mode of operation.

A retention register can have two operation modes: *normal active mode* and *retention mode*. The retention mode includes three stages: *save event*, *sleep mode*, and *restore event*. The regular flip-flop or latch operates in normal active mode and the balloon latch operates in retention mode. The normal active mode and retention mode stages are described as follows:

- normal active mode

  During this mode, the flip-flop or latch cell is in regular operation, meaning that the circuit is fully powered.

- retention mode
  - save event

    During this event sequence, the flip-flop or latch cell state is saved and the current state is stored before the power is shut down.

  - sleep mode

    During this mode, the power is turned off to the regular flip-flop or latch cell.

  - restore event

    During this event sequence, the output state is restored just after the power is restored and before the cell operation returns to normal active mode.

During normal active mode, the regular flip-flop operates at speed, and the always-on latch does not add to the load at the output. During sleep mode, the output data is transferred to the always-on latch, and the power supply to the flip-flop is shut off, thus reducing the leakage standby power. When the circuit is activated with the wake-up signal during the restore event sequence, the data in the always-on latch is transferred to the regular flip-flop for continuous operation.

Based on the application, different retention register types are available to address the clocking of the data from the register to the latch and back again. Different circuit designs allow you to determine how the data transfer occurs. For example, a clock-free retention

register is one type of retention register in which the state of the clock can be free-running in the always-on latch during power down. Other types of retention registers typically require the clock to be at high or low states during the save and restore events.

Retention Cell Modeling Syntax

The following syntax shows the modeling of retention cells. The reference cells are defined using flip-flop and latch syntax. The connectivity information for the input pins is defined by the `reference_input` attribute, based on the *`reference_pin_names`* variable, which specifies internal reference input nodes used within the `ff`, `latch`, `ff_bank`, and `latch_bank` groups.

```
cell(cell_name) {
  retention_cell : retention_cell_style;
  pin(pin_name) {
    retention_pin(pin_class, disable_value);
    ...
  }
  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 : internal_node_name;
   reference_input : pin_names;
   ...
  }
  ff/latch (["reference_pin_names",] variable1, variable2) {
    power_down_function : "Boolean_expression" ;
  }
  ff/latch_bank (["reference_pin_names",] variable1, variable2,
            bits) {
    power_down_function : "Boolean_expression" ;
  }
  ...

}
```

Note:

The syntax used to model scan versions of retention cells is the same syntax used to model scan sequential elements. All retention scan 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 along with the scan pins that have been specified with the `signal_type` attributes.

Cell-Level Attribute, Groups, and Variables

retention_cell Simple Attribute

The `retention_cell` attribute identifies a type of retention register with a string as its name. There can be multiple types of retention registers for a given register cell that provide the same functionality in normal mode but have different sleep signals, wake signals, or clocking schemes. For example, if a standard D flip-flop cell supports two retention strategies, such as type1 (where data is transferred when the clock is low) and type2 (where data is transferred when the clock is high), the `retention_cell` attribute value names must differ, 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 groups within a cell group.

reference_pin_names Variable

The optional, user-defined *`reference_pin_names`* variable specifies the internal reference input nodes that are used within the `ff`, `latch`, `ff_bank`, or `latch_bank` groups. If you do not specify the *`reference_pin_names`* variable, the node names used within the `ff`, `latch`, `ff_bank`, or `latch_bank` group are assumed to be actual pin or bus names within the cell.

variable1 and variable2 Variables

The `variable1` and `variable2` variables define internal reference output nodes. The `variable1` and `variable2` values in the `ff`, `latch`, `ff_bank`, or `latch_bank` groups must be unique within a cell.

bits Variable

The `bits` variable defines the width of the `ff_bank` and `latch_bank` component.

Pin-Level Attributes

retention_pin Complex Attribute

The `retention_pin` attribute identifies the retention pins of a retention cell.

Note:

Beginning with the Z-2007.03 release, the `power_gating_pin` attribute has been replaced by the `retention_pin` attribute. However, libraries with the old syntax are supported for backward compatibility.

The `retention_pin` attribute defines the following information:

- Pin class

  Valid values:

  - `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 save and restore operations, it must be specified using the `save_restore` attribute value. When this retention pin is saving the data, it is considered to be in *normal* mode. When it is not saving the data, it is restoring the data, and it is in *restore* mode.

- Disable value

  Defines the value of the retention pin when the cell works in normal mode. The valid values are 0 and 1.

  In the following example, the disable value of the `save_restore` pin `S_R` is `1`. The retention cell works in normal `save` mode when the shared `save_restore` pin is driven to 1 (high logic), and the retention cell works in `restore` mode when the `save_restore` pin is driven to 0 (low logic):

```
Pin(S_R) {
   ...
   retention_pin (save_restore, 1);
   ...
} /* end pin group */
```

Note:

If the retention signal pin is not associated with a power and ground pin by using the `related_power_pin` and `related_ground_pin` attributes, Library Compiler associates the signal pin to the first specified primary power and ground rails that are defined on the cell, and it marks the cell as `dont_touch/dont_use(d,u)` after generating the following warning messages:

```
Warning: Line 20, Cell 'retention_cell', pin 'RET', Connect pin 'RET' to
the default power pg_pin 'VDD'. (LBDB-725)
Warning: Line 20, Cell 'retention_cell', pin 'RET', Connect pin 'RET' to
the default ground pg_pin 'VSS'. (LBDB-725)
Warning: Line 20, Cell 'retention_cell', pin 'RET', is
missingrelated_power_pin to a 'power' pg_pin, so it will become a black
box cell for multivoltage functional optimization flow. It is being
marked as dont_touch, dont_use. (LBDB-914)
Warning: Line 100, Cell 'retention_cell', pin 'RET', is missing
related_ground_pin to a 'ground' pg_pin, so it will become a black box
cell for multivoltage functional optimization flow. It is being marked
dont_touch, dont_use. (LBDB-915)
```

function Attribute

You can define the `function` attribute in a pin group or a bus group. It maps an output, inout, or an internal pin to a corresponding internal node or to a `variable1` or `variable2` value in an `ff`, `latch`, `ff_bank`, or `latch_bank` group. The `function` attribute also accepts a Boolean equation containing `variable1` or `variable2`, as well as other input, inout, or internal pins.

reference_input Attribute

You can define the `reference_input` attribute in a pin group or a bus group. It specifies the input pins, which map directly to the reference pin names of the corresponding `ff`, `latch`, `ff_bank`, or `latch_bank` group. For each inout, output, or internal pin, the corresponding `ff`, `latch`, `ff_bank`, or `latch_bank` group is determined by the `variable1` or `variable2` value specified in its `function` statement.

Modeling Complex Retention Cells

This section describes modeling complex retention cell structures, such as edge-triggered retention signals. Use Figure 2-15 on page 2-65 as a reference for the retention cell structures.

Edge-Triggered Save and Restore Action

When a flip-flop uses a balloon or retention latch to store data when primary power is turned off, it requires a save and a restore control signal (which can be combined into one retention control). The following sections highlight the Liberty syntax that supports edge-triggered retention pins. The rationale draws from master-slave flip-flops treated as edge triggered.

Save Action

With the save operation, the retention storage element is generally a latch. It is generally safe to treat it as edge triggered rather than level sensitive because the output goes back to the flip-flop and transparency is not an issue. The trailing edge is the triggering edge. The behavior is similar to the master latch in an edge-triggered master-slave flip-flop.

Restore Action

The restore operation is more complex; however, the restore event can generally be treated as edge triggered. Transparency is not an issue provided that you manage the save, restore, and clock signals well. There are two ways to restore the state:

- You can treat the leading edge of the restore signal as the edge-triggered event. The flip-flop goes into transparency mode (it does not latch the data), but the data available in transparency mode is good and the same as the final latched data. This is similar to master-slave flip-flops, where the edge that causes the slave latch to go into transparency mode is the triggering edge for the flip-flop.
- You can treat the trailing edge as the triggering edge. At this point, the data is fully restored, and the flip-flop can again clock in data from the normal input. This method is more conservative, and it shortens the availability window of restored data.

Complex Retention Cell Modeling Syntax

The following syntax shows the modeling of complex retention cells. The syntax specific to complex retention cell modeling is highlighted.

```
cell(cell_name) {
  retention_cell : retention_cell_style;
  pin(pin_name) {
    retention_pin(pin_class, disable_value);
    save_action : L|H|R|F;
    restore_action : L|H|R|F;
    restore_edge_type : true_trigger | leading | trailing;
    ...
  }
  clock_condition() {
    clocked_on : "Boolean_expression";
    condition : "Boolean_expression";
    hold_state : L|H|N ;
    clocked_on_also : "Boolean_expression";
    condition_also : "Boolean_expression";
    hold_state_also : L|H|N;
  }
  preset_condition() {
        input : "Boolean_expression";
        condition : "Boolean_expression" ;
```

```
  }
  clear_condition() {
        input : "Boolean_expression" ;
        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";
     ...
  }
 ...
 }
}
```

Cell-Level Groups and Attributes

The following attributes are cell-level groups and attribute for retention cells.

clock_condition Group

The `clock_condition` group specifies special input conditions required to ensure signal legality during clock events. Different attributes under this group address the different scenarios of clock situations of retention cells that can happen when the cell is going into retention mode or going into normal active mode.

Allowing you to define edge-triggered behavior on save and restore signals sacrifices the ability to continuously monitor level-sensitive save and restore actions to check for all clock activities that are not allowed. An explicit clock-based check is necessary.

Within the `clock_condition` group, there are two classes of attributes: attributes without the `_also` suffix and attributes with the `_also` suffix. They correspond to the `clocked_on` and `clocked_on_also` attributes similar to those inside the `ff` group.

clocked_on Attribute

The `clocked_on` attribute is used to describe the situation when the clock is gated by another cell pin. Its Boolean expression should be the same as the one specified in the `clocked_on` attribute of the corresponding `ff` or `ff_bank` group.

condition Attribute

The `condition` attribute is evaluated at the “positive” edge of the `clocked_on` attribute. When the expression is evaluated to `false`, the cell is in an illegal state.

hold_state Attribute

The `hold_state` attribute describes to what value the `clocked_on` expression should evaluate for the retention mode. Permissible values are `L`, `H`, or `N` (which stands for stable low, stable high, or no-change respectively.)

If retention data is restored to both master and slave latches, the `hold_state` is `N` (which means an actual value of either `L` or `H` is fine). If retention data is only restored to the slave latch, the `hold_state` attribute should be `L` so that the slave latch can keep the data.

clocked_on_also Attribute

The `clocked_on_also` attribute is used to describe the situation when the clock is gated by another cell pin. Its Boolean expression is typically the same as the one specified in the `clocked_on_also` attribute of the corresponding `ff` or `ff_bank` group.

condition_also Attribute

The `condition_also` attribute is evaluated at the “positive” edge of the `clocked_on_also` attribute. When the expression is evaluated to `false`, the cell is in an illegal state.

This check is performed as the slave latch controlled by the `clocked_on_also` expression goes into hold mode.

hold_state_also Attribute

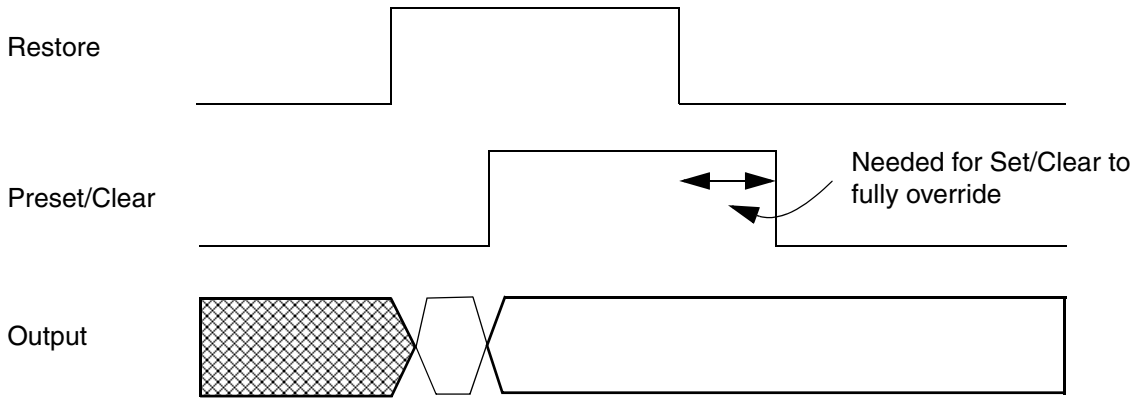
The `hold_state_also` attribute describes to what value the `clocked_on_also` expression should evaluate in retention mode. Permissible values are `L`, `H`, or `N` (which stands for stable low, stable high, or no-change respectively.)

preset_condition and clear_condition Groups

The `preset_condition` and `clear_condition` groups contain attributes for condition check on the normal-active-mode “preset” or “clear” expressions respectively.

As asynchronous controls, preset and clear have higher priority over the clock. However, their relation with save and restore is not straightforward because they normally do not overwrite the content of the balloon (retention) latch due to circuit complexity and performance considerations. If preset or clear is asserted during retention mode (specifically, the restore operation), it needs to extend beyond the restore window so that the flip-flop content can be successfully overwritten, as shown in Figure 2-16. Therefore, trailing-edge condition checks on preset and clear pins might be needed.

Figure 2-16 Preset and Clear Overlaps With Restore



input Attribute

The `input` attribute should be identical to the normal-active-mode expression, not including dedicated save and restore pins, in preset and clear inside the `ff` group. It defines how the (asynchronous) preset or clear control is asserted.

condition Attribute

The `condition` attribute is checked at the trailing edge of the input expression (that is, leaving asynchronous clear or preset state). When the expression is evaluated to `false`, the cell is in an illegal state.

Note that all (save, restore, preset, and clear) conditions are checked at the trailing edge of the control input as the respective latch goes into hold mode. However, due to the master and slave arrangement, both the leading and the trailing edges of the clock might need to be checked.

Pin-Level Groups and Attributes

The following attributes are defined inside the retention pin group.

save_action and restore_action Attributes

The `save_action` and `restore_action` attributes define when save event and restore event happen. Valid values are `L` (low), `H` (high), `R` (rise) and `F` (fall). For level-sensitive mode (that is, `L` and `H`), the actual event (storage) occurs at the trailing edge. For edge-trigger mode (that is, `R` and `F`), the edge defines when valid signal can be available at the output, subject to a nominal delay. See the `restore_edge_type` attribute for more information.

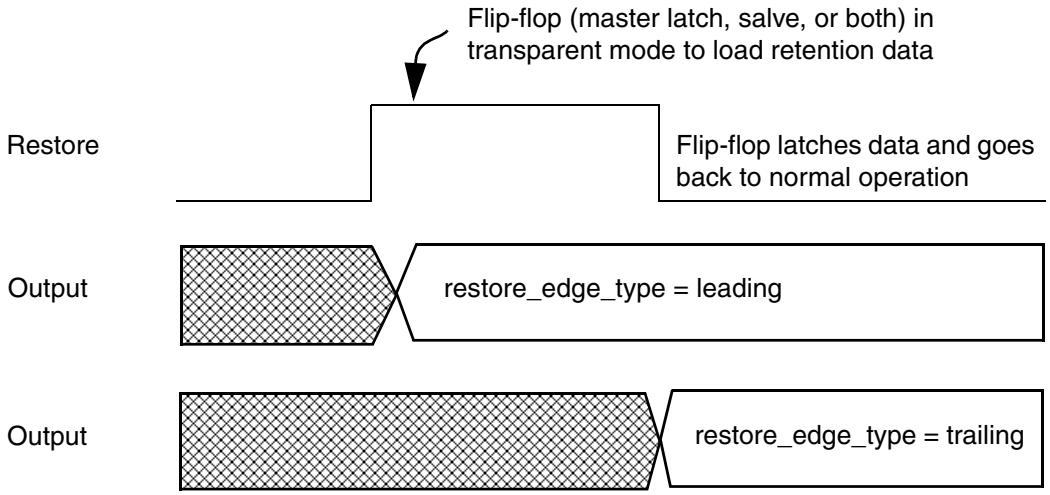
restore_edge_type Attribute

The `restore_edge_type` attribute supports the following edge-triggered mechanisms: `true_trigger`, `leading`, and `trailing`.

The `true_trigger` edge type refers to the fact that the restore event to the flip-flop is truly an edge-triggered clock. In other words, the flip-flop has two pairs of data and edge-triggered clock inputs. The flip-flop is always ready to take data gated by either clock at any time. There are no transparency windows imposed by the clocks on each other. As one clock triggers, the other clock can be at a stable 0 or 1.

The restore mechanism is latch-based for the `leading` and `trailing` edge types, meaning that the storage loops in the flip-flop are interrupted (they go into transparency mode) in order to load data from the retention element. The `leading` mechanism indicates that valid data is available at the output as the flip-flop goes into transparency. The `trailing` mechanism means that output is considered to be valid after the flip-flop is closed to restore data transfer. For both `leading` and `trailing` edges, the flip-flop reverts to normal operation after the trailing edge of the restore signal. This operation is shown in Figure 2-17:

Figure 2-17 Valid Data Window With leading and trailing restore_edge_type



Note that clock, preset, and clear all have leading-edge action, that is, they make the data available at the output while the (slave) latch is still in transparency mode. Therefore, the default value for the `restore_edge_type` attribute is also `leading`, to be consistent with the rest of the assumptions. Override this value with `trailing` or `true_trigger` depending on your modeling needs.

The save action on the balloon latch is generally `trailing` as its transparency has no outlet for use.

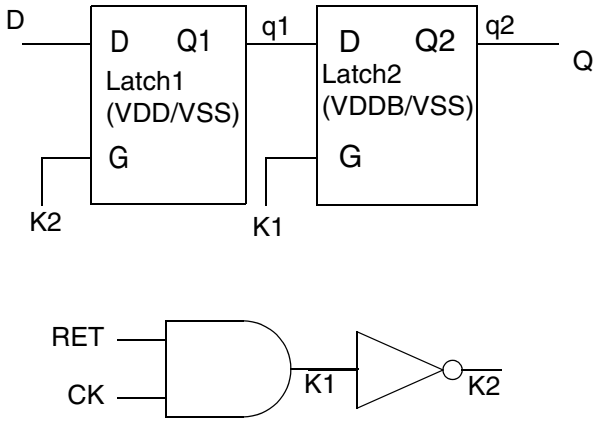
save_condition and restore_condition Attributes

The `save_condition` and `restore_condition` attributes specify the input condition when data is latched during save and restore respectively. These conditions are checked when respective `save_action` and `restore_action` are latched (at trailing edge). When the expression is evaluated to `false`, the cell is in an illegal state.

Retention Cell Model Examples

Figure 2-18 shows a schematic of a basic retention cell. The state of the cell is stored inside the slave latch powered by the backup power VDDB. In the figure, and in Example 2-2, the reference cells are defined using the `latch` syntax. The connectivity information for the input pins is specified in the `reference_input` attribute, which is mapped to the reference pin names of the corresponding `latch` group.

Figure 2-18 Simple Retention Cell Model Schematic



Example 2-2 Retention Cell Model Example Using Multiple Latch Group

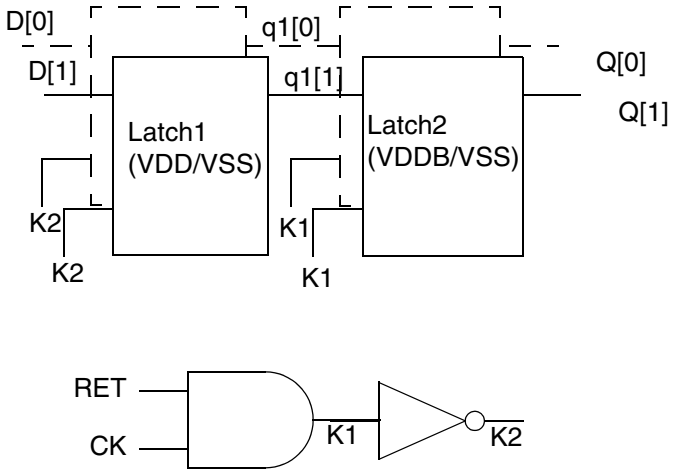
```
library (Retention_cell_Example_1) {
delay_model : table_lookup;

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 and groups */
cell (retention_cell) {
  retention_cell : my_ret_cell_1;
  ... /* Other cell level attributes and groups */
    pg_pin (VDD) {
      voltage_name : VDD;
      pg_type : primary_power;
    }
    pg_pin (VSS) {
      voltage_name : VSS;
      pg_type : primary_ground;
    }
    pg_pin (VDDB) {
      voltage_name : VDDB;
      pg_type : backup_power;
    }
    pin (RET) {
      direction : input;
      capacitance : 0.1;
      related_power_pin : VDDB;
      related_ground_pin : VSS;
      retention_pin(save_restore, "1");
      ... /* Other pin level attributes and groups */
    }
    pin(D) {
      direction : input;
      capacitance : 0.1;
      related_power_pin : VDD;
      related_ground_pin : VSS;
      ... /* Other pin level attributes and groups */
    }
    pin(CK) {
      direction : input;
      clock : true;
      capacitance : 0.1;
      related_power_pin : VDD;
      related_ground_pin : VSS;
      ... /* Other pin level attributes and groups */
    }
    pin (Q) {
      direction : output;
      function : "Q2";
      related_power_pin : VDD;
      related_ground_pin : VSS;
      reference_input : "RET CK q1";
      ... /* Other pin level attributes and groups */
    }
    pin (q1) {
```

```
      direction : internal;
      function : "Q1";
      related_power_pin : VDD;
      related_ground_pin : VSS;
      ... /* Other pin level attributes and groups */
    }
    latch ( Q1, QN1 ) {
      enable : " RET' + CK' ";
      data_in : " D ";
      power_down_function : "!VDD+VSS";
/* Latch1 is powered by primary power supply */
    }
    latch("p1 p2 p3", "Q2", "QN2") {
      enable : " p1 * p2 ";
      data_in : " p3 ";
      power_down_function : "!VDDB+VSS";
/* Latch2 is powered by backup power supply */
    } /* End Latch group */
  } /* End Cell group */
} /* End Library group */
```

Figure 2-19 and Example 2-3 show a retention cell structure that is defined using the `latch_bank` group that has multibit parallel inputs and output busses in the datapath and the clock path.

Figure 2-19 Multibit (2-bit) Retention Cell Model Schematic



Example 2-3 Retention Cell Model Example Using Multiple latch_bank Group

```
library (Retention_cell_Example_2) {
  delay_model : table_lookup;

  time_unit : "1ns";
  voltage_unit : "1V";
  current_unit : "1uA";
  capacitive_load_unit (0.1,ff);
```

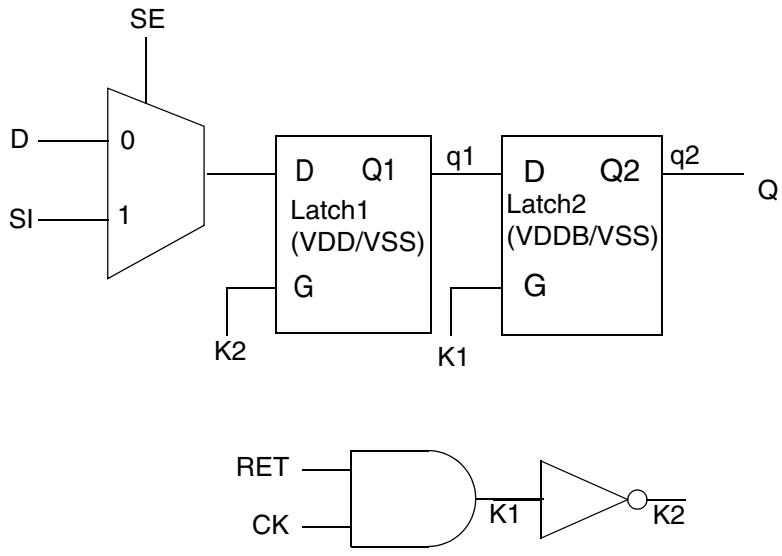
```
  default_fanout_load : 1.0;
  default_inout_pin_cap : 1.0;
  default_input_pin_cap : 1.0;
  default_output_pin_cap : 1.0;

  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;
  type (bus2) {
    base_type : array;
    data_type : bit;
    bit_width : 2;
    bit_from : 0;
    bit_to : 1;
    downto : false;
  }
...
/* Other library-level attributes */

cell (multibit_retention_cell) {
  retention_cell : MB_ret_cell;
  ... /* Other cell level attributes and groups */

  pg_pin(VDD) {
    voltage_name : VDD;
    pg_type : primary_power;
  }

  pg_pin(VSS) {
    voltage_name : VSS;
    pg_type : primary_ground;
```

```
}

pg_pin(VDDB) {
  voltage_name : VDDB;
  pg_type : backup_power;
}

pin(RET) {
  direction : input;
  capacitance : 0.1;
  related_power_pin : VDDB;
  related_ground_pin : VSS;
  retention_pin(save_restore, "1");
  ... /* Other pin level attributes and groups */
}

bus(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;
  min_pulse_width_high : 0.1;
  min_pulse_width_low : 0.2;
  ... /* Other pin-level attributes and groups */
}

bus(Q) {
  bus_type : bus2;
  direction : output;
  function : "Q2";
  related_power_pin : VDD;
  related_ground_pin : VSS;
  reference_input : "RET CK q1";
  ... /* Other pin-level attributes and groups */
}

bus(q1) {
  bus_type : bus2;
  direction : internal;
  function : "Q1";
  reference_input : "RET CK D";
  related_power_pin : VDD;
  related_ground_pin : VSS;
  ... /* Other pin-level attributes and groups */
```

```
  }

  latch_bank (“p1 p2,p3”, “Q1”, “QN1”, 2) {
    enable : " p1' + p2' ";
    data_in : " p3 ";
    power_down_function : "!VDD+VSS";
/* Latch1 is powered by primary power supply */
  }

  latch_bank ("p1 p2 p3", "Q2", "QN2", 2) {
    enable : " p1 * p2 ";
    data_in : " p3 ";
    power_down_function : "!VDDB+VSS";
 /* Latch2 is powered by backup power supply */
  }
  /* End Latch_bank group */
} /* End Cell group */
} /* End Library group */
```

Figure 2-20 and Example 2-4 shows a scan retention cell structure that is defined using the `latch` group. The cell is the scan version of the nonscan retention cell structure highlighted in Example 2-2. Note that the `test_cell` functional model of the cell has only the function of the nonscan version of the retention cell and Library Compiler can support multiple `latch`/`ff`/`latch_bank`/`ff_bank` groups inside the `test_cell` group. Similar to Example 2-2, this retention cell structure is also a store in-place (in the slave latch) retention cell.

Figure 2-20 MUX-Scan Retention Cell Model Schematic



Example 2-4 MUX-Scan Retention Cell Model

```
library (Retention_cell_Example_3) {
delay_model : table_lookup;
time_unit                    : "1ns";
```

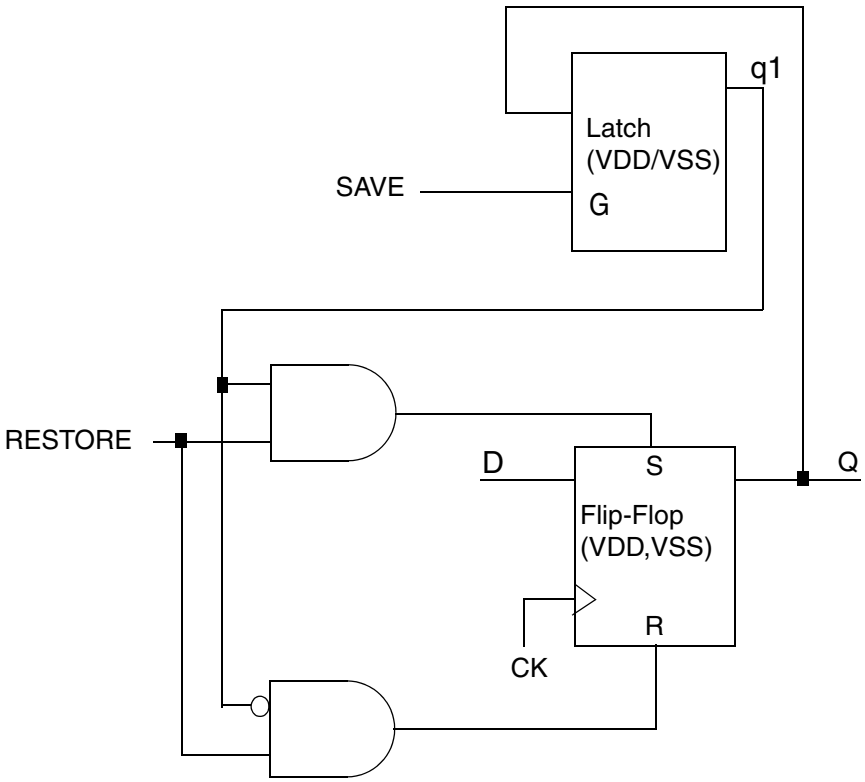
```
voltage_unit                 : "1V";
current_unit                 : "1uA";
capacitive_load_unit (0.1,ff);
default_fanout_load : 1.0;
default_inout_pin_cap : 1.0;
default_input_pin_cap : 1.0;
default_output_pin_cap : 1.0;
input_threshold_pct_rise     : 50 ;
input_threshold_pct_fall     : 50 ;
output_threshold_pct_rise    : 50 ;
output_threshold_pct_fall    : 50 ;
slew_lower_threshold_pct_fall : 30.0 ;
slew_lower_threshold_pct_rise : 30.0 ;
slew_upper_threshold_pct_fall : 70.0 ;
slew_upper_threshold_pct_rise : 70.0 ;
voltage_map (VDD, 1.0);
voltage_map (VDDB, 0.9);
voltage_map (VSS, 0.0);

nom_process                  :  1.0;
nom_temperature              : 25.0;
nom_voltage                  :  1.1;

operating_conditions(xyz) {
  process : 1.0 ;
  temperature : 25 ;
  voltage : 1.1 ;
  tree_type : "balanced_tree" ;
}
default_operating_conditions : xyz;
… /* Other library-level attributes */

cell (scan_retention_cell) {
  retention_cell : my_scan_ret_cell;
  …  /* Other cell-level attributes and groups */
  pg_pin(VDD) {
    voltage_name : VDD;
    pg_type : primary_power;
  }
  pg_pin(VSS) {
    voltage_name : VSS;
    pg_type : primary_ground;
          }
  pg_pin(VDDB) {
    voltage_name : VDDB;
    pg_type : backup_power;
  }
  pin(RET) {
    direction : input;
    capacitance : 0.1;
    related_power_pin : VDDB;
    related_ground_pin : VSS;
    retention_pin(save_restore, "1");
```

```
    … /* Other pin-level attributes and groups */
  }
  pin(D) {
    direction : input;
    capacitance : 0.1;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    … /* Other pin-level attributes and groups */
  }
  pin(CK) {
    direction : input;
    clock : true;
    capacitance : 0.1;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    … /* Other pin-level attributes and groups */
  }
  pin(Q) {
    direction : output;
    function : "Q2";
    related_power_pin : VDD;
    related_ground_pin : VSS;
    reference_input : "RET CK q1";
    … /* Other pin-level attributes and groups */
  }
  pin(q1) {
    direction : internal;
    function : "Q1";
    related_power_pin : VDD;
    related_ground_pin : VSS;
    reference_input : "RET CK SE D SI";
    … /* Other pin-level attributes and groups */
  }
  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 */
```

Figure 2-21 shows a retention cell structure that is defined using the `ff`/`latch` group. The cell has a balloon latch structure to store the data when the regular flip-flop logic is shutdown. The regular data is transferred to the output on the rising edge of the clock signal CK. The SAVE and RESTORE pins save and restore the data from the balloon latch to the cell output pin.

Figure 2-21 Retention Cell Model Schematic With Balloon Latch



Example 2-5 Retention Cell With Balloon Latch

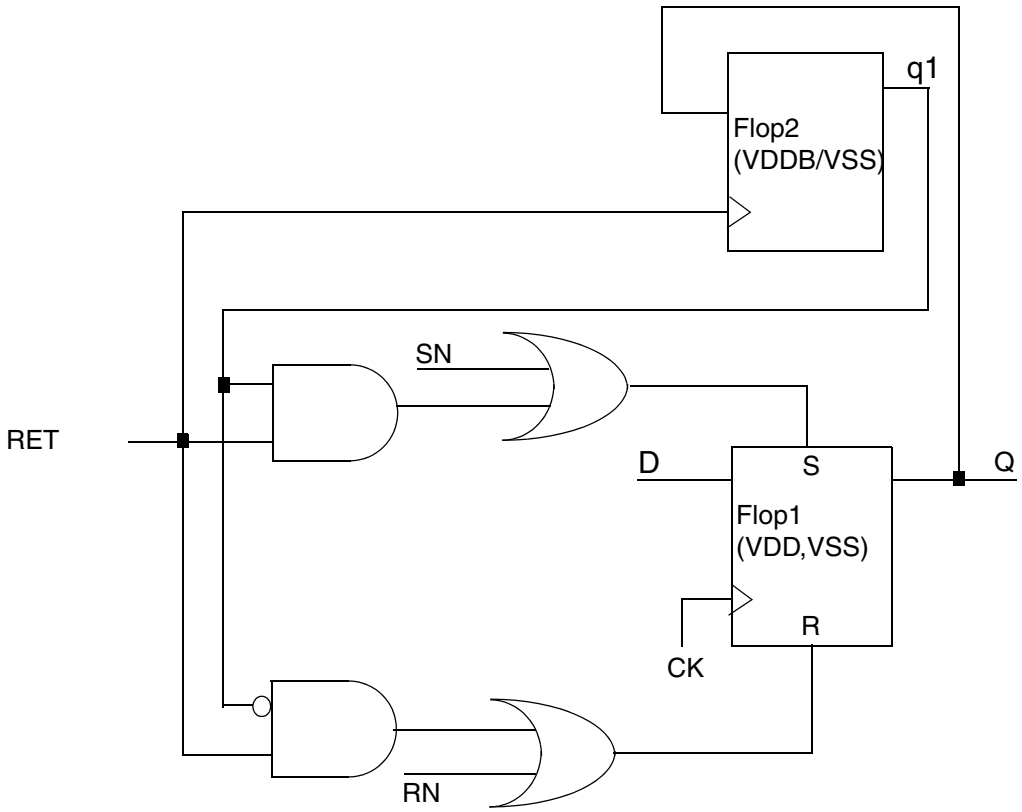
```
library (Retention_cell_Example_4) {
delay_model : table_lookup;
time_unit                    : "1ns";
voltage_unit                 : "1V";
current_unit                 : "1uA";
capacitive_load_unit (0.1,ff);
default_fanout_load : 1.0;
default_inout_pin_cap : 1.0;
default_input_pin_cap : 1.0;
default_output_pin_cap : 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 (balloon_retention_cell) {
  retention_cell : my_balloon_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(RESTORE) {
    direction : input;
    capacitance : 0.1;
    related_power_pin : VDDB;
    related_ground_pin : VSS;
    retention_pin(restore, "0");
    ... /* Other pin-level attributes and groups */
  }
  pin(SAVE) {
    direction : input;
    capacitance : 0.1;
    related_power_pin : VDDB;
    related_ground_pin : VSS;
    retention_pin(save, "0");
    ... /* 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;
    ... /* Other pin-level attributes and groups */
  }
  pin(q1) {
    direction : internal;
    function : "Q2";
    ... /* Other pin-level attributes and groups */
  }
  ff (“Q1”, “QN1”) {
    clocked_on : " CK ";
    data_in : "D";
    clear : "RESTORE * q1";
    preset : "RESTORE * !q1";
    clear_preset_var1 : "L";
    clear_preset_var2 : "H";

  power_down_function : "!VDD+VSS"; /* Flip-Flop is powered by Primary
power supply */
  }

  latch ("Q2", "QN2") {
    enable : "SAVE";
    data_in : "Q";
    power_down_function : "!VDDB+VSS"; /* Latch is powered by Backup
power supply */
  } /* End latch group */
 } /* End cell group */
} /* End Library group */
```

Figure 2-22 shows a retention cell structure that is defined using two `ff` groups. The cell has a balloon latch structure to store the data when the regular flip-flop logic is shutdown. The regular data is transferred to the output of Flop1 on the rising edge of the clock signal CK when the async preset (SN) and Clear (RN) are not active. In the retention mode, a single retention pin RET saves the data into the balloon flop and restores the data from the balloon flop to the cell output pin. On the rising edge of the RET signal, the data is saved inside the balloon flip-flop named Flop2 and the main Flop1 is being powered off. When the power is restored back to Flop1 and the retention pin becomes inactive, the data from Flop2 is restored to flop1 on the falling edge of the retention pin RET.

Figure 2-22 Edge-Triggered Retention Cell Model Schematic



Example 2-6 Retention Cell With Edge-Triggered Balloon Logic

```
library(edge_triggered_retention) {
delay_model : table_lookup;
time_unit                    : "1ns";
voltage_unit                 : "1V";
current_unit                 : "1uA";
capacitive_load_unit (0.1,ff);
default_fanout_load : 1.0;
default_inout_pin_cap : 1.0;
default_input_pin_cap : 1.0;
default_output_pin_cap : 1.0;
input_threshold_pct_rise     : 50 ;
input_threshold_pct_fall     : 50 ;
output_threshold_pct_rise    : 50 ;
output_threshold_pct_fall    : 50 ;
slew_lower_threshold_pct_fall : 30.0 ;
slew_lower_threshold_pct_rise : 30.0 ;
slew_upper_threshold_pct_fall : 70.0 ;
slew_upper_threshold_pct_rise : 70.0 ;
voltage_map(VDD, 1.0);
voltage_map(VDDB, 1.0);
```

```
voltage_map(VSS, 0.0);
  nom_process                  :  1.0;
  nom_temperature              :  25.0;
  nom_voltage                  :  1.1;
  operating_conditions(xyz) {
    process : 1.0 ;
    temperature : 25 ;
    voltage : 1.1 ;
    tree_type : "balanced_tree" ;

 }
default_operating_conditions : xyz;
cell (edge_trigger) {
  retention_cell : "edge_trigger";
  ... /* Other cell-level attributes and groups */
  pg_pin (VDD) {
    voltage_name : "VDD";
    pg_type : "primary_power";
  }
  pg_pin (VDDB) {
    voltage_name : "VDDB";
    pg_type : "backup_power";
  }
  pg_pin (VSS) {
  voltage_name : "VSS";
    pg_type : "primary_ground";
  }
  ff ("IQ1" , "IQN1") {
    next_state : "D";
    clocked_on : "CK";
    clear : "RN + (RET * q1')";
    preset : "SN + (RET * q1)";
    clear_preset_var1 : L;
    clear_preset_var1 : H;
    power_down_function : "!VDD + VSS"; /* Flip-Flop “Flop1” is powered
by Primary power supply */
  }
  ff ("IQ2" , "IQN2") {
    next_state : "Q";
    clocked_on : "RET";
    power_down_function : "!VDDS + VSS"; /* Flip-Flop “Flop2” is powered
by Primary power supply */
  }
  clock_condition() {
    clocked_on : "CK"; /* clock must be Low to go into retention mode */
    hold_state : "N"; /* when clock switches (either direction), RET must
be High */
    condition : "RET";
  }
  clear_condition() {
    input : "!RN"; /* When clear deasserts, RET must be high to allow Low
value to be transferred to Flop1 */
    condition : "RET";
```

```
  }
  preset_condition() {
    input : "!SN"; /* When clear deasserts, RET must be high to allow
High value to be transferred to Flop1 */
    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 : "VDDB";
    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 */
```

Retention Cell Library Checks

Library Compiler performs checks for retention cells and issues an error or warning message if certain conditions occur. For a detailed list of library checks for retention cells, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Always-On Cell Modeling

In complex low-power designs, some signals need to be routed through blocks that have been shut down. As a result, a variety of cell categories require “always-on” signal pins. Always-on cells remain powered on by a backup power supply in the region where they are placed even when the main power supply is switched off. The cells also have a secondary backup power pin that supplies the current that is necessary when the main supply is not available.

In order 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. Library Compiler provides the `always_on` attribute to identify always-on cells and pins. The following cell categories support always-on signals: `always_on` cell buffers or inverters, pins on retention cells, pins on switch cells, and pins on isolation cells. There is no restriction on any specific cell.

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, beginning with the D-2010.03 release. 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. 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 `always_on` attribute is supported at the cell level and at the pin level.

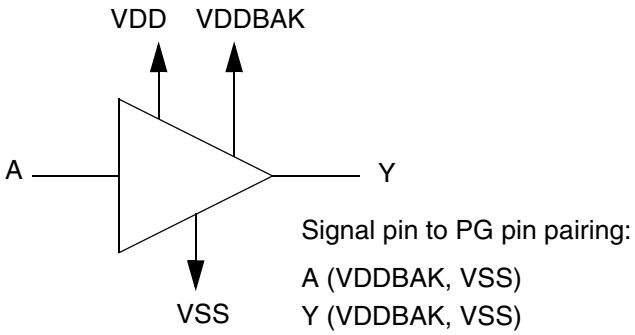
Note:

Some macro cell input pins require that you specify the `always_on` attribute for always-on pins.

Always-On Simple Buffer Example

Figure 2-23 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 2-23 Simple Always-On Cell Buffer



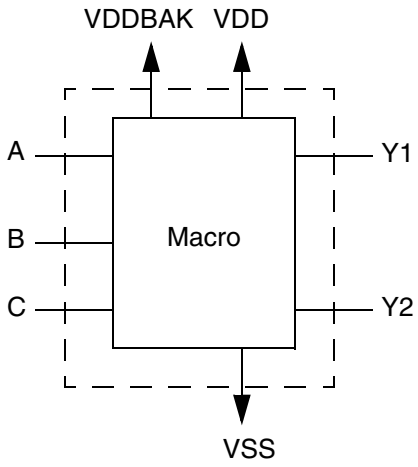

```
library (my_library) {
  …

     voltage_map (VDD, 1.0);
     voltage_map (VDDBAK, 1.0);
     voltage_map (VSS, 0.0);
  ...

   cell(buffer_type_AO) {
     always_on : true;
/* The always-on attribute is not required at the cell
 level if the cell is an always-on cell*/
    /* Other cell level information */
```

```
    pg_pin(VDD) {
      voltage_name : VDD;
      pg_type : primary_power;
    }

    pg_pin(VDDBAK) {
      voltage_name : VDDBAK;
      pg_type : backup_power;
    }

    pg_pin(VSS) {
      voltage_name : VSS;
      pg_type : primary_ground;
    }
    …
    pin (A) {
/* The always-on attribute is not required at the pin
 level if the cell is an always-on cell*/
      related_power_pin : VDDBAK;
      related_ground_pin : VSS;
    /* Other pin level information */
    }
    pin (Y) {
/* The always-on attribute is not required at the pin
 level if the cell is an always-on cell*/
      function : "A";
      related_power_pin : VDDBAK;
      related_ground_pin : VSS;
      power_down_function : "!VDDBAK + VSS";
    /* Other pin level information */
    } /* End Pin group */

  } /* End Cell group */
                        …
} /* End Library group */
```

Macro Cell With an Always-On Pin Example

Figure 2-24 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 2-24 Macro Cell With an Always-On Pin



Signal pin to PG pin pairing:

A (VDDBAK, VSS)
B (VDD, VSS)
C (VDD, VSS)
Y1 (VDDBAK, VSS)
Y2 (VDD, VSS)

```
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 */
```

Always-On Cell Library Checks

Library Compiler performs checks for always-on cells and issues an error or warning message if certain conditions occur. For a detailed list of library checks for always-on cells, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

3

Delay Models

Design Compiler uses the timing parameters and environment attributes described in a technology library to calculate the timing delays for your designs.

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

- CMOS Generic Delay Model
- CMOS Nonlinear Delay Model
- Scalable Polynomial Delay Model
- CMOS Piecewise Linear Delay Model
- Delay Calculation Module (DCM) Delay Model

The delay model you choose applies to all cells in the library; you cannot mix delay models within a single library. Each component of each delay model equation is affected differently by variations in the manufacturing process, operating temperature, and supply voltage. Consequently, separate scaling factors are applied to each component of each delay equation.

You define the appropriate delay model with the following syntax: (`table_lookup` applies to the CMOS nonlinear delay model):

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

CMOS Generic Delay Model

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

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

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

Total Delay Equation

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

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

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

D_I

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

D_S

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

D_C

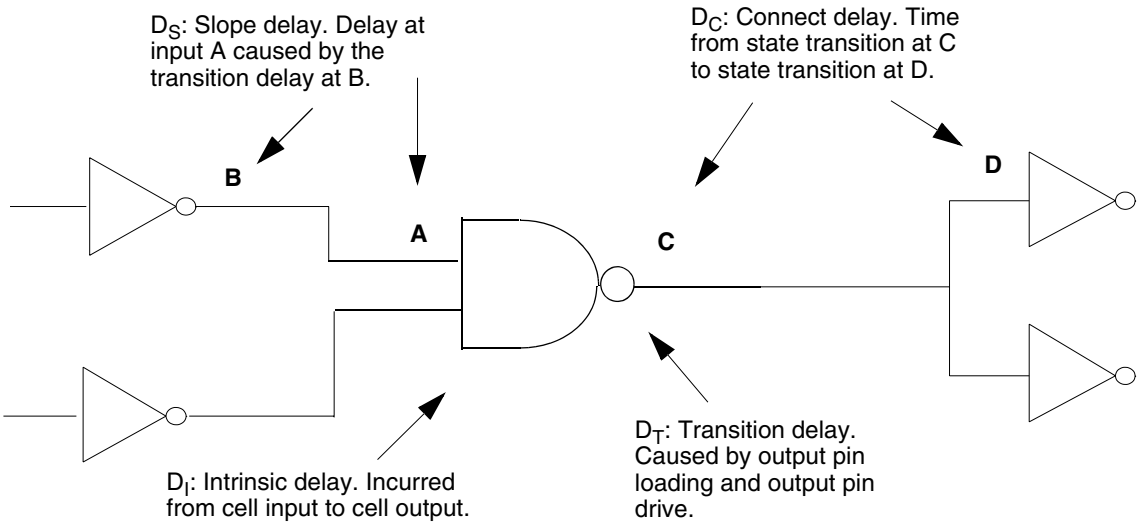
Connect media delay to an input pin (wire delay).

D_T

Transition delay caused by loading of the output pin.

Figure 3-1 shows the components of the total delay.

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



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

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

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

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

Total Delay Scaling

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

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

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

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

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

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

D

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

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

Represent the differences between the process, temperature, and voltage attributes of the prevailing `operating_conditions` group and the library's designated `nom_process`, `nom_temperature`, and `nom_voltage` attributes. Here is one illustration:

$D_{process}$ = process–nom_process

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

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

The total scaled delay equation takes this form:

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

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

Nominal operating conditions and `operating_conditions` groups are also defined at the library level. For a description of these statements, see "Defining Operating Conditions" on page 11-10 of this guide and the *Library Compiler Technology and Symbol Libraries Reference Manual*.

Example 3-2 on page 3-12 shows a complete scaled delay equation for a rise delay.

Slope Delay

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

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

D_S is included in the delay equation to allow additional accurate modeling of technologies that are sensitive to input ramp time (see "Total Delay Equation" on page 3-2).

D is calculated with the transition delay at the previous output pin, plus a slope sensitivity factor, as shown here: $D_S = S_S \times D_{T(prevstage)}$

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

D_S

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

S_S

Slope sensitivity factor. This factor accounts for the time during which the input voltage begins to rise but has not reached the threshold level at which channel conduction begins. The attributes that define it in the `timing` group of the driving pin are `slope_rise` and `slope_fall`.

$D_{T(prevstage)}$

The transition delay calculated at the previous output pin.

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

The total slope delay is calculated by scaling of the constant values by their corresponding k-factors (see "Total Delay Scaling" on page 3-3).

Intrinsic Delay

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

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

Total intrinsic delay is calculated by scaling of these constant values by their corresponding k-factors (see "Total Delay Scaling" on page 3-3).

Transition Delay

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

Here is the equation for transition delay:

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

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

R_{driver}

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

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

C_{wire}

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

C_{pins}

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

number_non_three_state_drivers

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

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

The total transition delay is calculated by scaling of the constant values by their corresponding k-factors (see "Total Delay Scaling" on page 3-3).

Connect Delay

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

Library Compiler and Design Compiler support three cases for an estimated interconnect topology. These three cases represent the three possible values of the `operating_conditions` group attribute `tree_type`. See "operating_conditions Group" on page 11-10 for details.

best_case_tree

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

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

worst_case_tree

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

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

balanced_tree

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

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

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

R_{wire}

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

C_{wire}

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

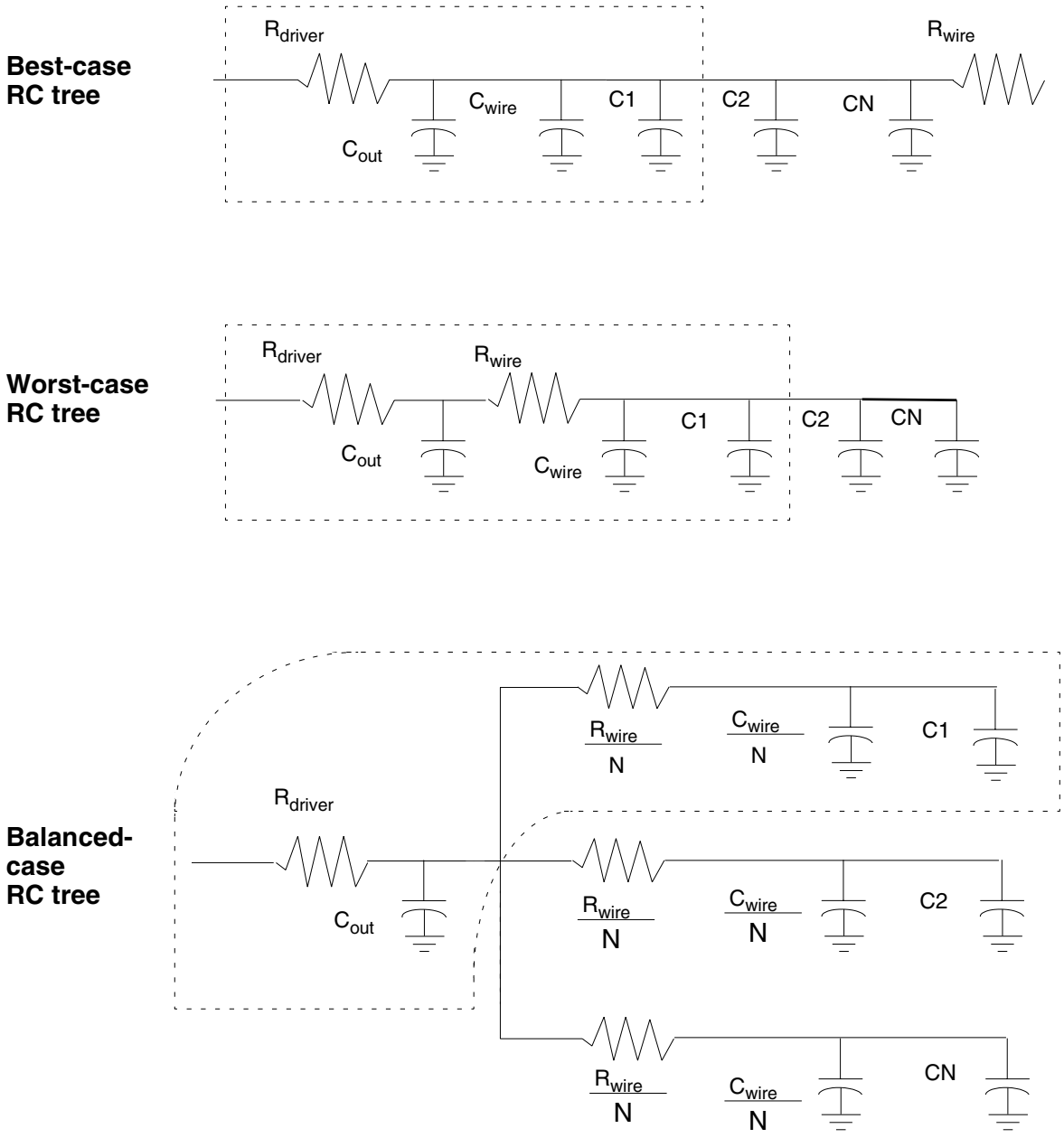
C_{pin}

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

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

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

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



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

Best-case $D_C = 0.0$

Worst-case D_C =

$$\begin{aligned}
&\text{wire_resistance}\ (1 + \Delta_{\text{voltage}}\ \text{k_volt_wire_res}) \\
&\quad (1 + \Delta_{\text{temp}}\ \text{k_temp_wire_res})(1 + \Delta_{\text{process}}\ \text{k_process_wire_res}) \\
&[\ (C_{\text{wire}})(1 + \Delta_{\text{voltage}}\ \text{k_volt_wire_cap}) \\
&\quad (1 + \Delta_{\text{temp}}\ \text{k_temp_wire_cap})(1 + \Delta_{\text{process}}\ \text{k_process_wire_cap}) \\
&+ \sum_{\text{pins}} \{\ (C_{\text{pin}})(1 + \Delta_{\text{voltage}}\ \text{k_volt_pin_cap}) \\
&\quad (1 + \Delta_{\text{temp}}\ \text{k_temp_pin_cap})(1 + \Delta_{\text{process}}\ \text{k_process_pin_cap})\}\ ]
\end{aligned}$$

Balanced-case D_C =

$$\begin{aligned}
&\text{wire_resistance/N}\ (1 + \Delta_{\text{voltage}}\ \text{k_volt_wire_res}) \\
&\quad (1 + \Delta_{\text{temp}}\ \text{k_temp_wire_res})(1 + \Delta_{\text{process}}\ \text{k_process_wire_res}) \\
&[\ (C_{\text{wire}}/N)(1 + \Delta_{\text{voltage}}\ \text{k_volt_wire_cap}) \\
&\quad (1 + \Delta_{\text{temp}}\ \text{k_temp_wire_cap})(1 + \Delta_{\text{process}}\ \text{k_process_wire_cap}) \\
&+ \ (C_{\text{pin}})(1 + \Delta_{\text{voltage}}\ \text{k_volt_pin_cap}) \\
&\quad (1 + \Delta_{\text{temp}}\ \text{k_temp_pin_cap})(1 + \Delta_{\text{process}}\ \text{k_process_pin_cap})]
\end{aligned}$$

Note:
N is the number of load pins on the net.

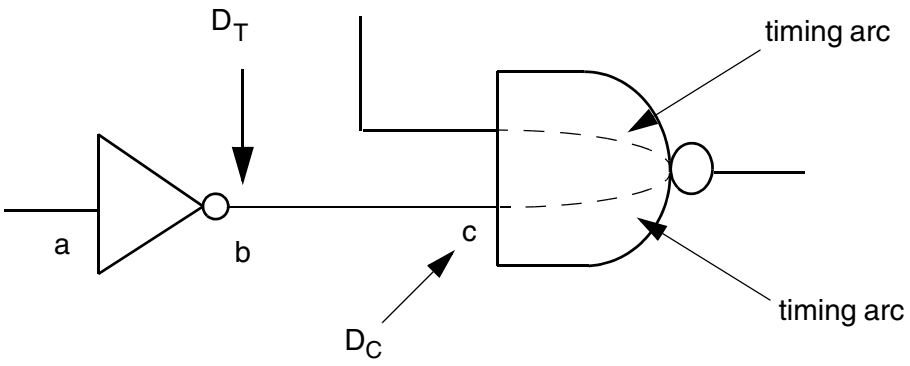
Interconnect Delay

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

$D_{interconnect} = D_T + D_C$

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

Figure 3-3 Interconnect Delay Diagram



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

Example 3-1 Modeling Interconnect Delay

```
cell (inv1) {
   area : 1 ;
   pin (a) {
      direction : input ;
      capacitance : 1 ;
   }
   pin (b) {
      direction : output ;
      capacitance : 0 ;
      timing () {
         related_pin : "a" ;
         rise_resistance : 1.2 ;
         fall_resistance : 0.8 ;
         intrinsic_rise : 1.5 ;
         intrinsic_fall : 1.2 ;
```

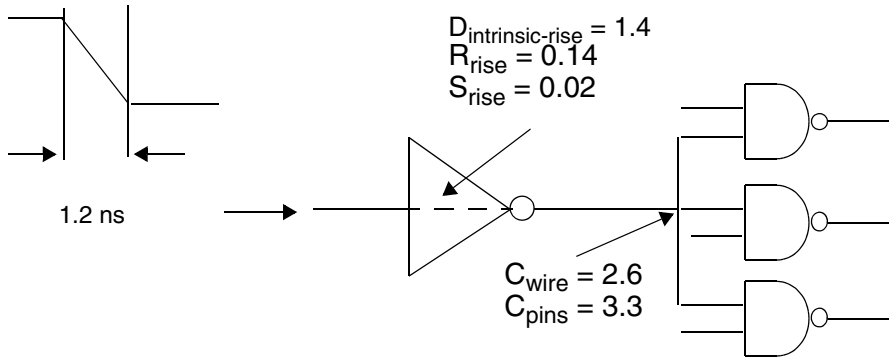
```
      }
   }
}
cell (NAND1) {
   area : 1.5 ;
   pin (in1) {
      direction : input ;
      capacitance : 1.5 ;
   }
   ...
}
```

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

Delay Calculation Example

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

Figure 3-4 Rise Delay Diagram



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

Example 3-2 Rise Delay Computation

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

```
= 2.25
```

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

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

Rise delay =

$$
\begin{aligned}
D_T \quad & + \ \text{rise\_resistance}\,(1+\Delta_{voltage}\,\text{k\_volt\_drive\_rise}) \\
& \quad (1+\Delta_{temp}\,\text{k\_temp\_drive\_rise})(1+\Delta_{process}\,\text{k\_process\_drive\_rise}) \\
& \quad [\,(C_{wire})(1+\Delta_{voltage}\,\text{k\_volt\_wire\_cap}) \\
& \quad\quad (1+\Delta_{temp}\,\text{k\_temp\_wire\_cap})(1+\Delta_{process}\,\text{k\_process\_wire\_cap}) \\
& \quad {\scriptstyle +\Sigma \atop \text{pins}}\ \{(C_{pin})(1+D_{voltage}\,\text{k\_volt\_pin\_cap}) \\
& \quad\quad (1+\Delta_{temp}\,\text{k\_temp\_pin\_cap})(1+\Delta_{process}\,\text{k\_process\_pin\_cap})\}\,]
\end{aligned}
$$

$$
\frac{D_I \quad \begin{aligned} & \text{intrinsic\_rise}\,(1+\Delta_{voltage}\,\text{k\_volt\_intrinsic\_rise}) \\ & \quad (1+\Delta_{temp}\,\text{k\_temp\_intrinsic\_rise})(1+\Delta_{process}\,\text{k\_process\_intrinsic\_} \end{aligned}}{\text{---------- long dashed line ----------}}
$$

$$
\frac{D_S \quad \begin{aligned} & + \ (D_{Tprevstage})(\text{slope\_rise})(1+\Delta_{voltage}\,\text{k\_volt\_slope\_rise}) \\ & \quad (1+\Delta_{temp}\,\text{k\_temp\_slope\_rise})(1+\Delta_{process}\,\text{k\_process\_slope\_rise}) \end{aligned}}{\text{---------- long dashed line ----------}}
$$

$$
\begin{aligned}
&+ \quad \text{wire\_resistance}\,(1 + D_{voltage}\ \text{k\_volt\_wire\_res}) \\
&\qquad (1 + \Delta_{temp}\ \text{k\_temp\_wire\_res})(1 + \Delta_{process}\ \text{k\_process\_wire\_res}) \\
D_C \qquad &\qquad [\,(C_{wire})(1 + \Delta_{voltage}\ \text{k\_volt\_wire\_cap})(1 + \Delta_{temp}\ \text{k\_temp\_wire} \\
&\qquad\qquad (1 + \Delta_{process}\ \text{k\_process\_wire\_cap}) \\
&\overset{+\Sigma}{_{pins}}\ \{(C_{pin})(1 + \Delta_{voltage}\ \text{k\_volt\_pin\_cap}) \\
&\qquad\qquad (1 + \Delta_{temp}\ \text{k\_temp\_pin\_cap})(1 + \Delta_{process}\ \text{k\_process\_pin\_cap}
\end{aligned}
$$

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

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

CMOS Nonlinear Delay Model

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

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

- The total delay equation
- Cell delay (D_{cell})
- Propagation delay ($D_{propagation}$)
- Transition delay ($D_{transition}$)

- Connect delay (D_C)
- CMOS nonlinear delay model calculation
- Process, voltage, and temperature scaling

Total Delay Equation

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

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

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

D_{cell}

The delay contributed by the gate itself, typically defined as the 50 percent input pin voltage to 50 percent output voltage. D_{cell} is computed in two ways, depending on the timing data provided. See "Cell Delay" on page 3-17 for a description of these computation methods.

D_C

The connect delay is either calculated with the `tree_type` attribute in the `operating_conditions` group and the selected `wire_load` model or is read in from a delay feedback file as in the standard delay equation. Connect delay is calculated by the same method used in other delay equations.

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

$D_{transition}$

The time between two reference voltage levels on the output pin. For example, these levels might be 20 percent to 80 percent or 10 percent to 50 percent. Computing $D_{transition}$ involves performing table lookup and interpolation.

The CMOS nonlinear delay model supports two methods of computing D_{cell}. Although the two methods can be intermixed in a technology library, typically you specify the one method that correlates best to the characterized library data.

The two methods are

- Performing table lookup and interpolation in a cell delay table provided in the library
- Using the propagation and transition tables, following this equation:

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

The typical measurement for $D_{propagation}$ is the time from 50 percent input pin voltage until gate output just begins to switch—for example, when the 10 percent output voltage is reached. Thus, with a $D_{transition}$ value of 10 percent to 50 percent output voltage added to $D_{propagation}$, the result is a 50 percent input to 50 percent output cell delay.

If cell delay tables are provided for a timing arc, the total delay equation is

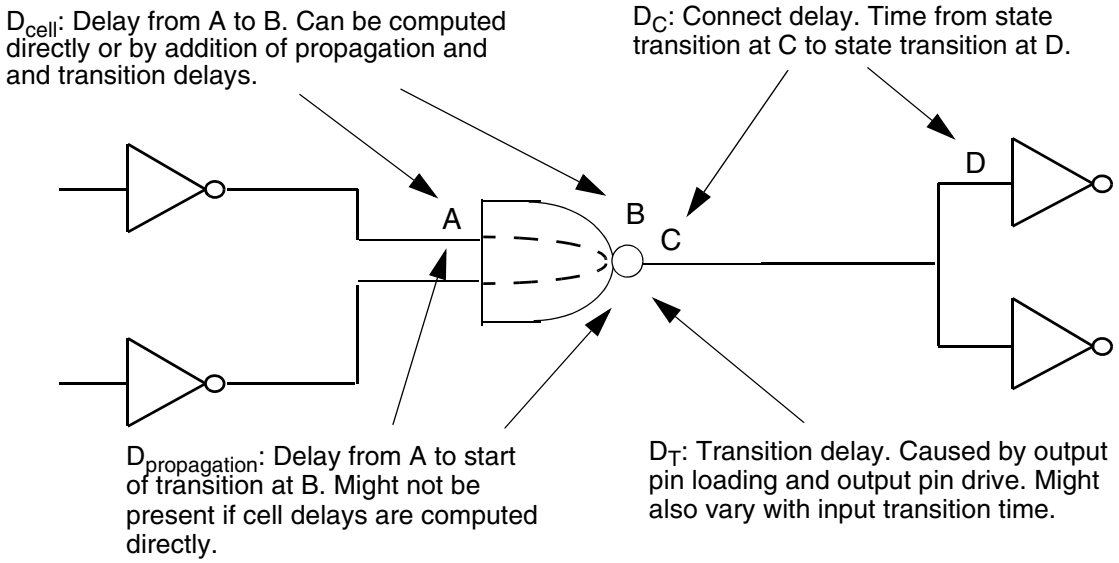
$D_{total} = D_{cell} + D_C$

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

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

Figure 3-5 shows the delay equation components.

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



Cell Delay

Cell delay (D_{cell}) is typically defined as the 50 percent input pin voltage to 50 percent output voltage. Cell delay is usually a function of both output loading and input transition time.

Two groups in the `timing` group define cell delay tables:

```
cell_rise
cell_fall
```

If you define cell delay tables are for a timing arc, do not specify propagation delay tables for that arc.

For more information about how to define delay tables, see Chapter 4, “Timing Arcs.”

Propagation Delay

$D_{propagation}$ is the time from the input transition to completion of a specified percentage (for example, 20 percent) of the output transition. $D_{propagation}$ is often a function of output loading and input transition time.

Two groups in the `timing` group define propagation delay tables:

```
rise_propagation
fall_propagation
```

If propagation delay tables are defined for a timing arc, cell delay tables must not be specified. The presence of propagation delay tables indicates that cell delays are computed by addition of the propagation and transition delays.

For more information about how to define delay tables, see Chapter 4, “Timing Arcs.”

Transition Delay

$D_{transition}$ is the time required for an output pin to change state. It is used as a term in the cell delay calculation if propagation tables are specified. After applicable transition degradation, it is also used to index into delay and transition tables at the next logic stage if the tables are indexed by `input_net_transition`. Transition delay can also be constrained as a design rule.

Computing $D_{transition}$ involves performing table lookup and interpolation. $D_{transition}$ is a function of capacitance at the output pin and can also be a function of input transition time in submicron technology.

Two groups are used to define transition delay tables:

```
rise_transition
fall_transition
```

Transition tables must be specified for all delay arcs.

For more information about how to define delay tables, see Chapter 4, “Timing Arcs.”

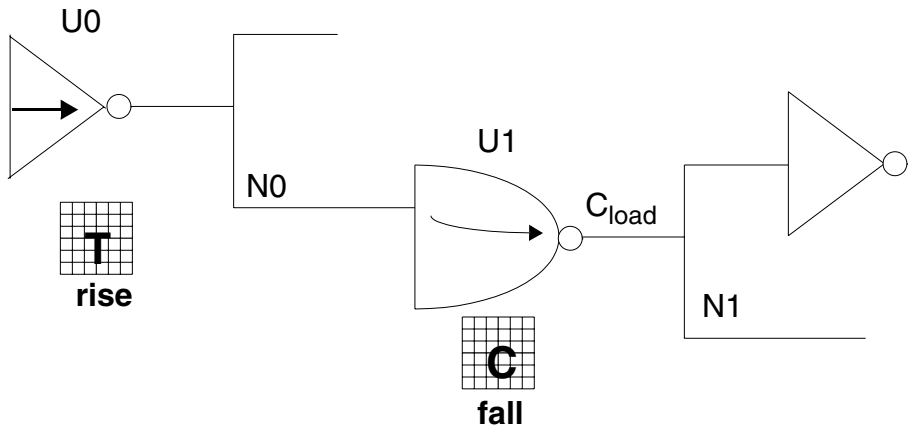
Connect Delay

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

CMOS Nonlinear Delay Model Calculation

Figure 3-6 is a schematic representation of the total delay through a cell. Figure 3-7 displays the result of using a CMOS nonlinear delay model to determine delay through a cell. To determine the fall delay across the timing arc of cell U1 in Figure 3-6, first examine the timing arc and determine whether propagation or cell delay tables are specified. In this case, a cell delay table—a two-dimensional table indexed by input transition time and total output capacitance, is specified.

Figure 3-6 Nonlinear Delay Calculation Schematic



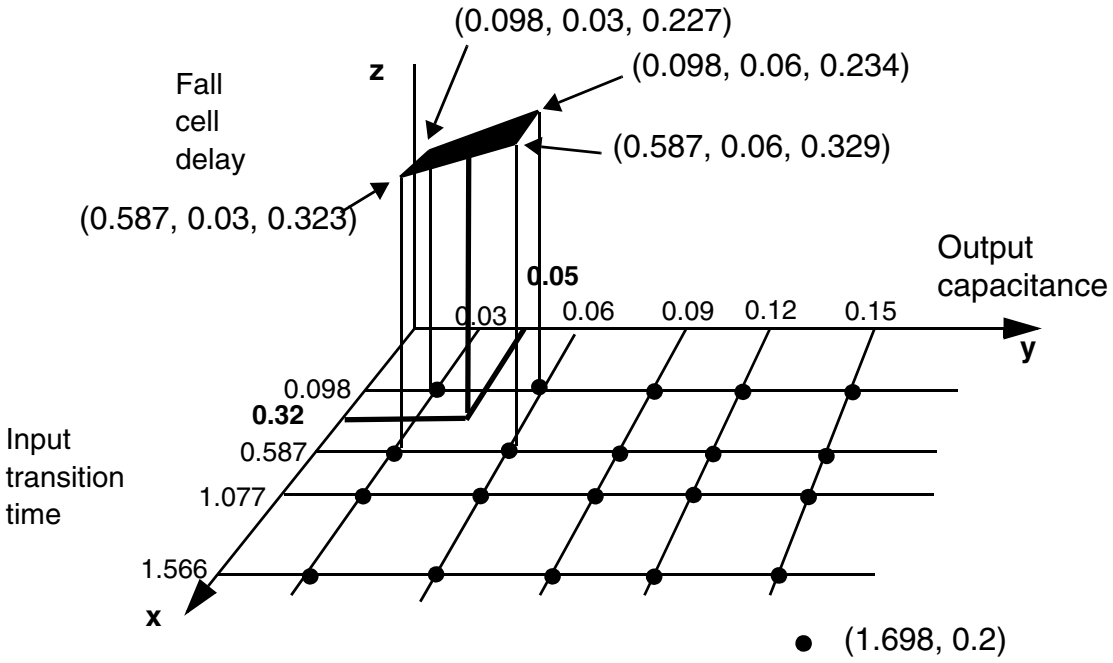
Input transition time is determined by evaluation of the transition delay at the previous gate, U0. Because the timing sense in U1 is negative unate, use the rise transition table for U0 to determine U1’s input transition time. If multiple timing arcs are present at gate U0, the maximum of the rise transition values for those arcs is used as the input transition at U1.

Total output capacitance is calculated by addition of the capacitance introduced by the pins connected to net N1 and the capacitance contributed by the wire itself. For transition, cell, and propagation delays, the `tree_type` attribute of an `operating_conditions` group is not used in the total output capacitance calculation. For wire delay, however, wire capacitance is calculated by use of a `wire_load` model, but it can also be back-annotated.

Performing these calculations determines C_{load} to be 0.05 and the input transition time 0.32. You can now index into U1's fall cell delay table with these two values. Four neighboring table values are determined by examination of the table breakpoints.

For this example, the input transition time, 0.32, falls between the index values of 0.098 and 0.587. The output capacitance 0.05 falls between the index values of 0.03 and 0.06. The index values correspond to the x and y coordinates of the four neighboring table points as shown in Figure 3-7. The corresponding z values are also shown in Figure 3-7.

Figure 3-7 Result of Delay Calculation



You determine the approximation for the surface described by the three coordinates (x, y, z) in Figure 3-7 by solving the A, B, C, and D coefficients of the following equation. Next, insert the coefficient values into the equation to determine which z coordinate relates to the fall propagation delay (see Example 3-3).

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

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

Example 3-3 Determining Coefficients and Solving for the Fall Cell Delay

```
0.227 = A + B * 0.098 + C * 0.03 + D * 0.098 * 0.03
0.234 = A + B * 0.098 + C * 0.06 + D * 0.098 * 0.06
0.323 = A + B * 0.587 + C * 0.03 + D * 0.587 * 0.03
0.329 = A + B * 0.587 + C * 0.06 + D * 0.587 * 0.06
```

Table 3-1 shows the coefficient values for the fall cell delay.

Table 3-1 Coefficient Values for Fall Cell Delay

Coefficient	Value
A	0.2006
B	0.1983
C	0.2399
D	0.0677

Insert the coefficient values and the x, y values that equal (0.32, 0.05) to solve for z (fall cell delay):

```
0.275 = 0.2006 + 0.1983 * 0.32 + 0.2399 * 0.05 + 0.0677 *
0.32 * 0.05
```

In two-dimensional interpolation, the dots in Figure 3-7 represent points you defined in the library source file for the CMOS nonlinear delay model table. This information is extracted from a SPICE modeling simulation. The four points with heights shown on the z-axis are the neighboring points chosen for interpolation. The shaded area is the surface used for interpolation.

If the index in the table cannot cover the actual value for the dimension—for example, if `total_output_net_capacitance` is 0.2 in Figure 3-7—extrapolation is performed to extend the surface formed by the nearest four data points to cover it.

For example, if you have a data point (1.698, 0.2), use data points (1.077, 0.12), (1.077, 0.15), (1.566, 0.12), and (1.566, 0.15) to get a set of A, B, C, and D. Next, insert the data point (1.698, 0.2) to get its fall propagation delay, using the method shown in Example 3-3.

For a one-dimensional table, the first and last line segments are extended to cover the data point if it falls outside the table index range.

Process, Voltage, and Temperature Scaling

When calculating total delay, Design Compiler scales each parameter of D_{total} individually. Each parameter has its own global parameters to model the effects on the nominal case of variation in process, temperature, and voltage.

The equation for a scaled version of an individual parameter is shown here:

$$P_{scaled} = P(1 + \Delta_{process} \times K_{process})(1 + \Delta_{temp} \times K_{temp})(1 + \Delta_{voltage} \times K_{voltage})$$

P

A parameter used in the total delay equation, such as cell, transition, or propagation table delay value; wire resistance or capacitance; or pin capacitance.

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

Represent the differences between the process, temperature, and voltage attributes of the prevailing `operating_conditions` group and the library's designated `nom_process`, `nom_temperature`, and `nom_voltage` attributes. For example,

$D_{process}$ = process – nom_process

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

Represent the scaling factors of the individual components of each parameter.

Scaling factors for process, temperature, and voltage are called k-factors (scaling factors that begin with k_) and are defined at the `library` group level in the technology library. Library Compiler assigns a 0 (zero) as the default value for each scaling attribute not defined in your library.

Nominal operating conditions and `operating_conditions` groups are also defined at the `library` group level. For a description of these statements, see "Defining Operating Conditions" on page 11-10 and the *Library Compiler Technology and Symbol Libraries Reference Manual*..

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

In the nonlinear delay model (as in other Design Compiler models), delay parameters are scaled before delay calculation is performed. This means that post-scaling transition and capacitance values are used to index into delay tables.

In many technology modeling schemes, the overall delay (D_{total})—rather than individual components such as wire resistance, pin capacitance, or transition delay values—is scaled.

To scale D_{total} by using the nonlinear delay model, specify the cell delay tables or propagation delay tables for timing arcs. Next, specify k-factors, to scale the cell table and propagation table values, and wire resistance, to achieve the overall delay scaling.

Scalable Polynomial Delay Model

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

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

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

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

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

Note:

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

Polynomial Representation

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

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

$$P_1Q_2 = (a_0 + a_1x_1) (b_0 + b_1x_2 + b_2x_2^2) =$$
$$A_{00} + A_{10}x_1 + A_{01}x_2 + A_{11}x_1x_2 + A_{02}x_2^2 + A_{12}x_1x_2^2$$

Model Description

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

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

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

You specify polynomials in the expanded decomposed form. Assume m, n, p, q, r, and s as the orders of each of six dimensions (x1—x6). The number of coefficients is

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

The coefficients are expressed as

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

Examples

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

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

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

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

Example 3-5 Sequence of Coefficients in the timing Group

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

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

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

The following features are also supported in the delay model:

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

Scalable Polynomial Calculation

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

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

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

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

Note:

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

Using the Conversion Feature

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

1. Enable the conversion.

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

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

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

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

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

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

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

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

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

or

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

CMOS Piecewise Linear Delay Model

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

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

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

Total Delay Equation

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

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

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

Following are the components of the delay model:

D_I

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

D_S

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

D_C

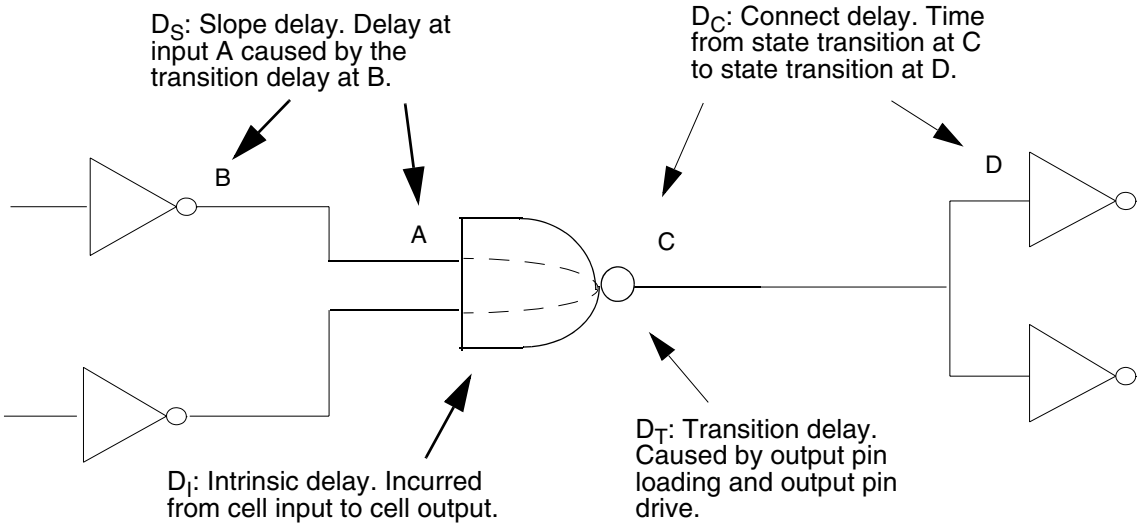
Connect media delay to an input pin (wire delay).

D_T

Transition delay caused by loading of the output pin.

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

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



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

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

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

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

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

Total Delay Scaling

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

$D_{total} = D_1 + D_S + D_C + D_T$

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

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

D

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

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

Represent the differences between the process, temperature, and voltage attributes of the prevailing `operating_conditions` group and the library's designated `nom_process`, `nom_temperature`, and `nom_voltage` attributes. Here is one illustration:

$D_{process}$ = process – nom_process

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

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

The total scaled delay equation takes this form:

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

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

Nominal operating conditions and `operating_conditions` groups are also defined at the library level. For a complete description of these statements, see "Defining Operating Conditions" on page 11-10 and the *Library Compiler Technology and Symbol Libraries Reference Manual*.

For an illustration of total delay scaling, see Example 3-6 on page 3-37.

Piecewise Linear Model

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

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

The syntax of the `piece_define` attribute is

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

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

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

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

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

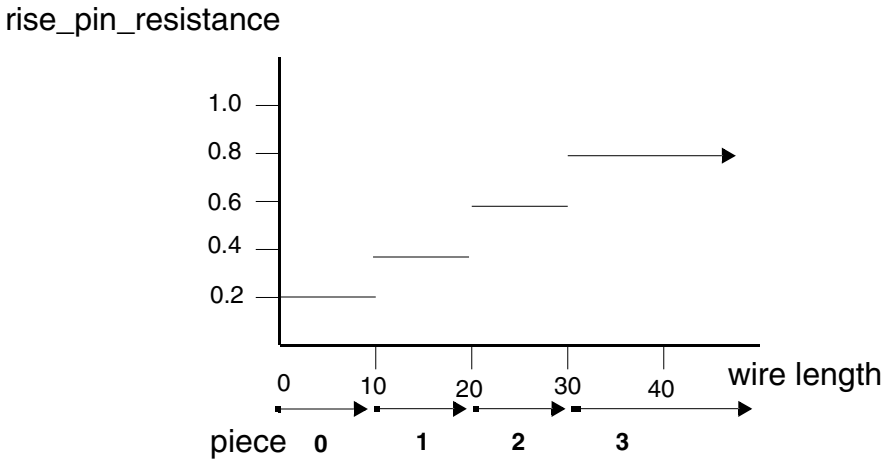
In this model, you must include all ranges of wire length or capacitance you want to associate with unique attribute values.

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

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

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

Figure 3-9 Graph of rise_pin_resistance Values



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

Intrinsic Delay

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

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

The total intrinsic delay is calculated by scaling of these constant values by their corresponding k-factors (see "Total Delay Scaling" on page 3-28).

Slope Delay

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

In some technologies, this delay is a strong function of the ramp time. In other technologies, this delay does not vary—even over a wide range of ramp values. D_S is included in the delay equation to allow more-accurate modeling of technologies that are sensitive to input ramp time. See "Total Delay Scaling" on page 3-28.

D_S is calculated with the transition delay at the previous output pin, plus a slope-sensitivity factor, as shown here: $D_S = S_S \times D_{T(prevstage)}$

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

S_S

Slope sensitivity factor. This factor accounts for the time during which the input voltage begins to rise but has not reached the threshold level at which channel conduction begins. The attributes are defined in the `timing` group of the driving pin as

```
slope_rise
slope_fall
```

$D_{T(prevstage)}$

The transition delay calculated at the previous output pin.

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

D_S

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

The total slope delay is calculated by scaling of the constant values by their corresponding k-factors. See "Total Delay Scaling" on page 3-28.

Transition Delay

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

$$D_T = R_{tdrive_i}\left(C_{pins} + C_{wire} \right) + Y_{adj_i}$$

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

R_{tdrive_i}

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

```
rise_pin_resistance
fall_pin_resistance
```

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

C_{pins}

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

C_{wire}

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

Y_{adj_i}

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

```
rise_delay_intercept
fall_delay_intercept
```

The total transition delay is calculated by scaling of the constant values by their corresponding k-factors. See “Total Delay Scaling” on page 3-28.

Connect Delay

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

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

R_{wire}

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

C_{wire}

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

C_{pin}

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

Library Compiler and Design Compiler support three cases for an estimated interconnect topology. These three cases represent the three possible values of the `operating_conditions` group attribute `tree_type` (see "Defining Operating Conditions" on page 11-10).

best_case_tree

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

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

worst_case_tree

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

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

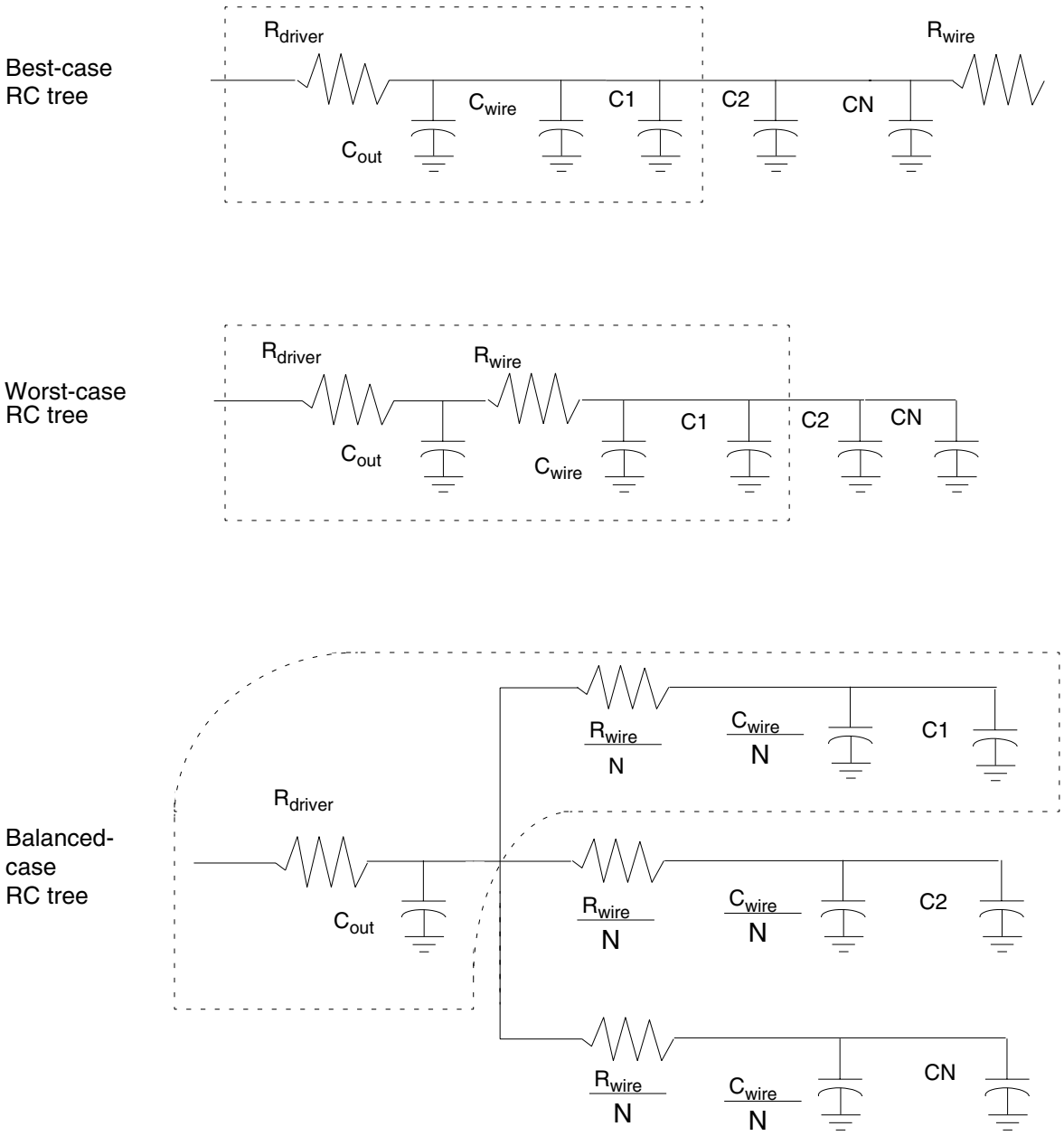
balanced_tree

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

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

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

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



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

Best-case $D_C = 0.0$

Worst-case D_C =

$$\begin{aligned}
&\text{wire_resistance}\ (1 + \Delta_{\text{voltage}}\ \text{k_volt_wire_res}) \\
&\quad (1 + \Delta_{\text{temp}}\ \text{k_temp_wire_res})(1 + \Delta_{\text{process}}\ \text{k_process_wire_res}) \\
&[\ (C_{\text{wire}})(1 + \Delta_{\text{voltage}}\ \text{k_volt_wire_cap}) \\
&\quad (1 + \Delta_{\text{temp}}\ \text{k_temp_wire_cap})(1 + \Delta_{\text{process}}\ \text{k_process_wire_cap}) \\
&+ \sum_{\text{pins}} \{\ (C_{\text{pin}})(1 + \Delta_{\text{voltage}}\ \text{k_volt_pin_cap}) \\
&\quad (1 + \Delta_{\text{temp}}\ \text{k_temp_pin_cap})(1 + \Delta_{\text{process}}\ \text{k_process_pin_cap})\}\ ]
\end{aligned}$$

Balanced-case D_C =

$$\begin{aligned}
&\text{wire_resistance/N}\ (1 + \Delta_{\text{voltage}}\ \text{k_volt_wire_res}) \\
&\quad (1 + \Delta_{\text{temp}}\ \text{k_temp_wire_res})(1 + \Delta_{\text{process}}\ \text{k_process_wire_res}) \\
&[\ (C_{\text{wire}}/\text{N})(1 + \Delta_{\text{voltage}}\ \text{k_volt_wire_cap}) \\
&\quad (1 + \Delta_{\text{temp}}\ \text{k_temp_wire_cap})(1 + \Delta_{\text{process}}\ \text{k_process_wire_cap}) \\
&+ \quad (C_{\text{pin}})(1 + \Delta_{\text{voltage}}\ \text{k_volt_pin_cap}) \\
&\quad (1 + \Delta_{\text{temp}}\ \text{k_temp_pin_cap})(1 + \Delta_{\text{process}}\ \text{k_process_pin_cap})]
\end{aligned}$$

N is the number of load pins on the net.

The total connect delay is calculated by scaling of the constant values by their corresponding k-factors. See “Total Delay Scaling” on page 3-28.

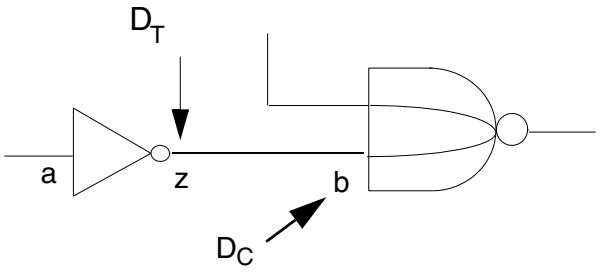
Interconnect Delay

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

$D_{interconnect} = D_T + D_C$

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

Figure 3-11 Interconnect Delay Diagram



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

Example 3-6 Modeling Interconnect Delay

```
library (example) {
   date : "November 12, 2002" ;
   revision : 2.2 ;
   piece_define ( "0 10" ) ;
   cell (inv1) {
      area : 1 ;
      pin (a) {
         direction : input ;
         capacitance : 1 ;
      }
      pin (z) {
         direction : output ;
         capacitance : 0 ;
         timing () {
            related_pin : "a" ;
            intrinsic_rise : 1.5 ;
            intrinsic_fall : 1.2 ;
            rise_pin_resistance (0, 1.2) ;
            fall_pin_resistance (0, 0.8) ;
            rise_delay_intercept (0, 0.89) ;
            fall_delay_intercept (0, 0.71) ;
```

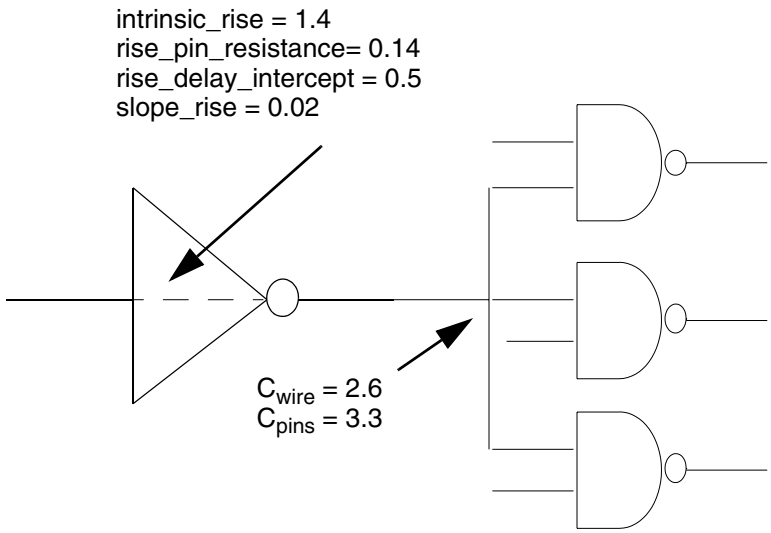
```
            rise_pin_resistance (1, 1.4) ;
            fall_pin_resistance (1, 0.98) ;
            rise_delay_intercept (1, 1.02) ;
            fall_delay_intercept (1, 0.89) ;
         }
      }
   }
   cell (NAND1) {
      area : 1.5 ;
      pin (b) {
         direction : input ;
         capacitance : 1.5 ;
      }
      ...
   }
```

Total interconnect delay is defined as the sum of the scaled versions of D_T and D_C. See "Total Delay Scaling" on page 3-28.

Delay Calculation Example

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

Figure 3-12 Rise Delay Diagram



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

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

Example 3-7 Rise Delay Computation

```
intrinsic_rise = 1.4
rise_pin_resistance = 0.14
rise_delay_intercept = 0.5
slope_rise = 0.02
C_pins = 3 * (1.1) = 3.3
C_wire = 2.6
      DT(fall_previous_stage) = 1.2
D_C = 0.0                          /* for a best-case RC tree */
Rise Delay = Intrinsic + Slope + Transition + Connect
      = 1.4 + (1.2 * 0.02) + [(0.14)(3.3 + 2.6) + 0.5]+ 0.0
      = 1.4 + 0.024        + 1.326                    + 0.0
      = 2.75
```

Figure 3-8 on page 3-27 provides details about the components of the total delay equation.

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

Rise delay =

$$D_I \quad \frac{\text{intrinsic\_rise}\,(1+\Delta_{voltage}\,\text{k\_volt\_intrinsic\_rise})}{(1+\Delta_{temp}\,\text{k\_temp\_intrinsic\_rise})(1+\Delta_{process}\,\text{k\_process\_intrinsic\_rise})}$$

$$D_S \quad + \frac{(D_{Tprevstage})(\text{slope\_rise})(1+\Delta_{voltage}\,\text{k\_volt\_slope\_rise})}{(1+\Delta_{temp}\,\text{k\_temp\_slope\_rise})(1+\Delta_{process}\,\text{k\_process\_slope\_rise})}$$

$$+ \text{ rise\_pin\_resistance } (1 + \Delta_{voltage}\ \text{k\_volt\_rise\_pin\_resistance})$$
$$(1 + \Delta_{temp}\ \text{k\_temp\_rise\_pin\_resistance})(1 + \Delta_{process}\ \text{k\_process\_rise\_pin\_resistance})$$
$$D_T \quad [\ (C_{wire})(1 + \Delta_{voltage}\ \text{k\_volt\_wire\_cap})$$
$$(1 + \Delta_{temp}\ \text{k\_temp\_wire\_cap})(1 + \Delta_{process}\ \text{k\_process\_wire\_cap})$$
$$\overset{+\Sigma}{\text{pins}}\ \{(C_{pin})(1 + D_{voltage}\ \text{k\_volt\_pin\_cap})$$
$$(1 + \Delta_{temp}\ \text{k\_temp\_pin\_cap})(1 + \Delta_{process}\ \text{k\_process\_pin\_cap})\}\ ]$$

$$+ \text{ rise\_delay\_intercept } (1 + \Delta_{voltage}\ \text{k\_voltage\_rise\_delay\_intercept})$$
$$(1 + \Delta_{temp}\ \text{k\_temp\_rise\_delay\_intercept})(1 + \Delta_{process}\ \text{k\_process\_rise\_delay\_intercept})$$

$$+ \text{ wire\_resistance } (1 + D_{voltage}\ \text{k\_volt\_wire\_res})$$
$$(1 + \Delta_{temp}\ \text{k\_temp\_wire\_res})(1 + \Delta_{process}\ \text{k\_process\_wire\_res})$$
$$D_C \quad [\ (C_{wire})(1 + \Delta_{voltage}\ \text{k\_volt\_wire\_cap})(1 + \Delta_{temp}\ \text{k\_temp\_wire\_cap})$$
$$(1 + \Delta_{process}\ \text{k\_process\_wire\_cap})$$
$$\overset{+\Sigma}{\text{pins}}\ \{(C_{pin})(1 + \Delta_{voltage}\ \text{k\_volt\_pin\_cap})$$
$$(1 + \Delta_{temp}\ \text{k\_temp\_pin\_cap})(1 + \Delta_{process}\ \text{k\_process\_pin\_cap})\}\ ]$$

Delay Calculation Module (DCM) Delay Model

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

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

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

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

Selecting the DCM model has the following consequences:

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

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

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

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

4

Timing Arcs

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

For information on describing delay, see these sections:

- Describing State-Dependent Delays

For information on describing constraints, see these sections:

- Setting Setup and Hold Constraints
- Setting Nonsequential Timing Constraints
- Setting Recovery and Removal Timing Constraints
- Setting No-Change Timing Constraints
- Setting Skew Constraints
- Setting Conditional Timing Constraints

For additional information, see these sections:

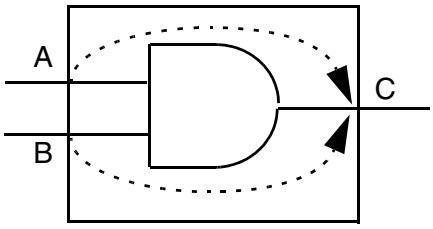
- Timing Arc Restrictions
- Examples of Libraries Using Delay Models
- Describing a Transparent Latch Clock Model
- Checking Timing and Nonlinear Delay Data
- Driver Waveform Support
- Sensitization Support
- Phase-Locked Loop Support

Understanding Timing Arcs

Timing arcs, along with netlist interconnect information, are the paths followed by the path tracer during path analysis. Each timing arc has a startpoint and an endpoint. The startpoint can be an input, output, or I/O pin. The endpoint is always an output pin or an I/O pin. The only exception is a constraint timing arc, such as a setup or hold constraint between two input pins.

Figure 4-1 shows timing arcs AC and BC for an AND gate. All delay information in a library refers to an input-to-output pin pair or an output-to-output pin pair.

Figure 4-1 Timing Arcs



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

Design Compiler uses combinational timing arc information to calculate the physical delays in timing propagation and to trace paths. The timing analyzer uses path-tracing arcs for circuit timing analysis.

Design Compiler uses sequential timing arc information to determine rule-based design optimization constraints. See the *Design Compiler Reference Manual* for more information on optimization constraints.

Combinational Timing Arcs

A combinational timing arc describes the timing characteristics of a combinational element. The timing arc is attached to an output pin, and the related pin is either an input or an output.

A combinational timing arc is of one of the following types:

- combinational
- combinational_rise
- combinational_fall

- three_state_disable
- three_state_disable_rise
- three_state_disable_fall
- three_state_enable
- three_state_enable_rise
- three_state_enable_fall

For information on describing combinational timing types, see "timing Group Attributes" on page 4-24.

Sequential Timing Arcs

Sequential timing arcs describe the timing characteristics of sequential elements. In descriptions of the relationship between a clock transition and data output (input to output), the timing arc is considered a *delay* arc. In descriptions of the relationship between a clock transition and data input (input to input), the timing arc is considered a *constraint* arc.

A sequential timing arc is of one of the following types:

- Edge-sensitive (rising_edge or falling_edge)
- Preset or clear
- Setup or hold (setup_rising, setup_falling, hold_rising, or hold_falling)
- Nonsequential setup or hold (non_seq_setup_rising, non_seq_setup_falling, non_seq_hold_rising, non_seq_hold_falling)
- Recovery or removal (recovery_rising, recovery_falling, removal_rising, or removal_falling)
- No change (nochange_high_high, nochange_high_low, nochange_low_high, nochange_low_low)

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

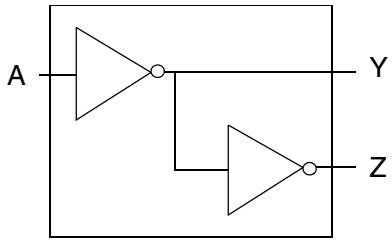
- Skew (skew_rising or skew_falling)

For information on describing sequential timing types, see "timing Group Attributes" on page 4-24.

Modeling Method Alternatives

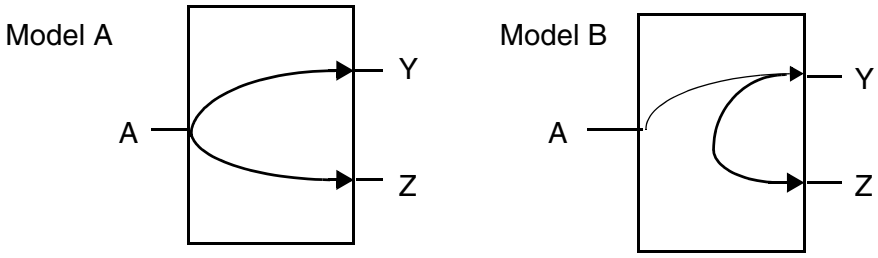
Timing information for combinational cells such as the one in Figure 4-2 can be modeled in one of two ways, as Figure 4-3 shows.

Figure 4-2 Two-Inverter Cell



In Figure 4-3, Model A defines two timing arcs. The first timing arc starts at primary input pin A and ends at primary output pin Y. The second timing arc starts at primary input pin A and ends at primary output pin Z. This is the simple case.

Figure 4-3 Two Modeling Techniques for Two-Inverter Cell



Model B for this cell also has two arcs but is more accurate than Model A. The first arc starts at pin A and ends at pin Y. This arc is modeled like the AY arc in Model A. The second arc is different; it starts with primary output Y and ends with primary output Z, modeling the effect the load on Y has on the delay on Z. Output-to-output timing arcs can be used in either combinational or sequential cells.

Defining the timing Group

The `timing` group contains the information Design Compiler needs in order 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 4-1 on page 4-3.

Note:

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

Naming Timing Arcs Using the timing Group

Within the `timing` group, you can identify the name or names of different timing arcs.

A single timing arc can occur between an identified pin and a single related pin identified with the `related_pin` attribute.

Multiple timing arcs can occur in many ways. The following list shows six possible multiple timing arcs. The following descriptive sections explain how you configure other possible multiple timing arcs:

- Between a single related pin and the identified multiple members of a bundle
- Between multiple related pins and the identified multiple members of a bundle
- Between a single related pin and the identified multiple bits on a bus
- Between multiple related pins and the identified multiple bits of a bus
- Between the identified multiple bits of a bus and the multiple pins of related bus pins (of a designated width) identified with the `related_bus_pins` attribute.
- Between all the bits of a related-bus-equivalent group identified with the `related_bus_equivalent` attribute and an internal pin, and between the internal pin and all the bits of the endpoint `bus` group.

The following sections provide descriptions and examples for various timing arcs.

Timing Arc Between a Single Pin and a Single Related Pin

Identify the timing arc that occurs between a single pin and a single related pin by entering a name in the `timing` group, as shown in the following example:

Example

```
cell (my_inverter) {
   ...
   pin (A) {

      direction : input;
      capacitance : 1;
   }
   pin (B) {
      direction : output
      function : "A'";
      timing (A_B) }
         related_pin : "A";
      ...
      }/* end timing() */
   }/* end pin B */
}/* end cell */
```

The timing arc is as follows:

From pin	To pin	Label
A	B	A_B

Timing Arcs Between a Pin and Multiple Related Pins

This section describes how to identify the timing arcs when a `timing` group is within a `pin` group and the timing arc has more than a single related pin. You identify the multiple timing arcs on a name list entered with the `timing` group as shown in the following example:

Example

```
cell (my_and) {
      ...
   pin (A) {
      direction : input;
      capacitance : 1;

   }
   pin (B) {
      direction : input;
      capacitance : 2;
   }
   pin (C) {
      direction : output
      function : "A B";
      timing (A_C, B_C) {
         related_pin : "A B";
         ...
      }/* end timing() */
   }/* end pin B */
}/* end cell */
```

The timing arcs are as follows:

From pin	To pin	Label
A	C	A_C
B	C	B_C

Timing Arcs Between a Bundle and a Single Related Pin

When the `timing` group is within a `bundle` group that has several members with a single related pin, enter the names of the resulting multiple timing arcs in a name list in the `timing` group.

Library Compiler assumes that the first name in the name list is the arc going from the related pin to the first pin in the bundle member list, the second name in the name list is the arc going from the related pin to the second pin in the bundle member list, and so on. See the following example:

Example

```
...
bundle (Q){
   members (Q0, Q1, Q2, Q3);
   direction : output;
   function : "IQ";
   timing (G_Q0, G_Q1, G_Q2, G_Q3){
      timing_type : rising_edge;
      intrinsic_rise : 0.99;
      intrinsic_fall : 0.96;
      rise_resistance : 0.1458;
      fall_resistance : 0.0653;
      related_pin : "G";
   }
}
```

If G is a pin, as opposed to another `bundle` group, the timing arcs are as follows:

From pin	To pin	Label
G	Q0	G_Q0
G	Q1	G_Q1
G	Q2	G_Q2
G	Q3	G_Q3

If G is another bundle of member size 4 and G0, G1, G2, and G3 are members of bundle G, the timing arcs are as follows:

From pin	To pin	Label
G0	Q0	G_Q0
G1	Q1	G_Q1

G2	Q2	G_Q2
G3	Q3	G_Q3

Timing Arcs Between a Bundle and Multiple Related Pins

When the `timing` group is within a `bundle` group that has several members, each having a corresponding related pin, enter the names of the resulting multiple timing arcs as a name list in the `timing` group.

Library Compiler assumes that the first name in the name list is the arc going from the related pin to the first pin in the bundle member list, the second name in the name list is the arc going from the second related pin to the second pin in the bundle member list, and so on. See the following example:

Example

```
bundle (Q){
   members (Q0, Q1, Q2, Q3);
   direction : output;
   function : "IQ";
   timing (G_Q0, H_Q0, G_Q1, H_Q1, G_Q2, H_Q2, G_Q3, H_Q3){
      timing_type : rising_edge;
      intrinsic_rise : 0.99;
      intrinsic_fall : 0.96;
      rise_resistance : 0.1458;
      fall_resistance : 0.0653;
      related_pin : "G H";
   }
}
```

If G is a pin, as opposed to another `bundle` group, the timing arcs are as follows:

From pin	To pin	Label
G	Q0	G_Q0
H	Q0	H_Q0
G	Q1	G_Q1
H	Q1	H_Q1
G	Q2	G_Q2
H	Q2	H_Q2
G	Q3	G_Q3

H	Q3	H_Q3

If G was another bundle of member size 4 and G0, G1, G2, and G3 are members of bundle G, the timing arcs are as follows:

From pin	To pin	Label
G0	Q0	G_Q0
H	Q0	H_Q0
G1	Q1	G_Q1
H	Q1	H_Q1
G2	Q2	G_Q2
H	Q2	H_Q2
G3	Q3	G_Q3
H	Q3	H_Q3

The same rule applies if H is a size-4 bundle.

Timing Arcs Between a Bus and a Single Related Pin

This section describes how to identify the timing arcs created when a `timing` group is within a `bus` group that has several bits with the same single related pin. You identify the resulting multiple timing arcs by entering a name list with the `timing` group.

Library Compiler assumes that the first name in the name list identifies the arc going from the related pin to the most significant bit (MSB) in the `bus` group, the second name in the name list identifies the arc going from the related pin to the second MSB in the `bus` group, and so on. See the following example:

Example

```
 ...
bus (X){
/*assuming MSB is X[0] */
   bus_type : bus4;
   direction : output;
   capacitance : 1;
   pin (X[0:3]){
   function : "B’";
   timing (B_X0, B_X1, B_X2, B_X3){
```

```
            related_pin : "B";
    }
}
```

If B is a pin, as opposed to another 4-bit bus, the timing arcs are as follows:

From pin	To pin	Label
B	X[0]	B_X0
B	X[1]	B_X1
B	X[2]	B_X2
B	X[3]	B_X3

If B is another 4-bit bus and B[0] is the MSB for bus B, the timing arcs are as follows:

From pin	To pin	Label
B[0]	X[0]	B_X0
B[1]	X[1]	B_X1
B[2]	X[2]	B_X2
B[3]	X[3]	B_X3

Timing Arcs Between a Bus and Multiple Related Pins

Library Compiler assumes that the first name in the name list is the arc going from the first related pin to the most significant bit (MSB) in the `bus` group, the second name in the name list is the arc going from the second related pin to the second MSB in the `bus` group, and so on. See the following example:

This section describes the timing arcs created when a `timing` group is within a `bus` group that has several bits, where each bit has its own related pin. You identify the resulting multiple timing arcs by entering a name list with the `timing` group.

Example

```
bus (X){
/*assuming MSB is X[0] */
   bus_type : bus4;
   direction : output;
   capacitance : 1;
   pin (X[0:3]){
      function : "B'";
      timing (B_X0, C_X0, B_X1, C_X1, B_X2, C_X2, B_X3,C_X3 ){
         related_pin : "B C";
      }
   }
}
```

If B and C are pins, as opposed to another 4-bit bus, the timing arcs are as follows:

From pin	To pin	Label
B	X[0]	B_X0
C	X[0]	C_X0
B	X[1]	B_X1
C	X[1]	C_X1
B	X[2]	B_X2
C	X[2]	C_X2
B	X[3]	B_X3
C	X[3]	C_X3

If B is another 4-bit bus and B[0] is the MSB for bus B, the timing arcs are as follows:

From pin	To pin	Label
B[0]	X[0]	B_X0
C	X[0]	C_X0
B[1]	X[1]	B_X1
C	X[1]	C_X1
B[2]	X[2]	B_X2

C	X[2]	C_X2
B[3]	X[3]	B_X3
C	X[3]	C_X3

The same rule applies if C is a 4-bit bus.

Timing Arcs Between a Bus and Related Bus Pins

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

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

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

Example

```
 ...
/* assuming related_bus_pins is width of 2 bits */
bus (X){
/*assuming MSB is X[0] */
   bus_type : bus4;
   direction : output;
   capacitance : 1;
   pin (X[0:3]){
      function : "B'";
      timing (B0_X0, B0_X1, B0_X2, B0_X3, B1_X0, B1_X1, B1_X2, B1_X3 ){
         related_bus_pins : "B";
      }
   }
}
```

If B is another 2-bit bus and B[0] is its MSB, the timing arcs are as follows:

From pin	To pin	Label
B[0]	X[0]	B0_X0
B[0]	X[1]	B0_X1
B[0]	X[2]	B0_X2
B[0]	X[3]	B0_X3

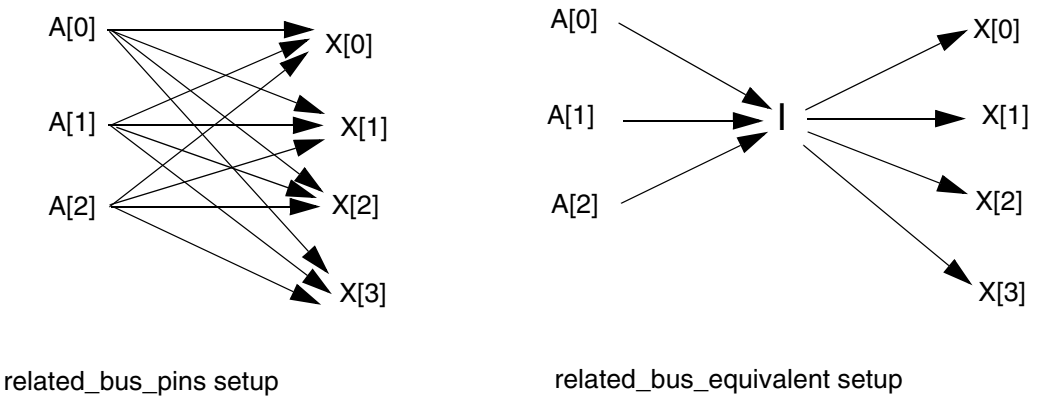
B[1]	X[0]	B1_X0
B[1]	X[1]	B1_X1
B[1]	X[2]	B1_X2
B[1]	X[3]	B1_X3

Timing Arcs Between a Bus and a Related Bus Equivalent

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

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

Figure 4-4 Comparing related_bus_pins Setup With related_bus_equivalent Setup



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

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

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

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

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

Example

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

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

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

Delay Models

The `timing` groups are defined by the `timing` group attributes. The delay model you use determines which set of delay calculation attributes you specify in a `timing` group.

Cell delays are often modeled in a synthesis technology library, with an assortment of delay models. Synthesis tools use these models to predict cell delays during and after optimization. Net delays are estimated during synthesis on the basis of the wire load models provided in the technology library.

Synthesis tools support these delay models (see Chapter 3, "Delay Models," for detailed information):

CMOS generic delay model

This is the standard delay model.

CMOS nonlinear delay model

The nonlinear delay model is characterized by tables that define the timing arcs. To describe delay or constraint arcs with this model,

- Use the library-level `lu_table_template` group to define templates of common information to use in lookup tables.
- Use the templates and the timing groups described in this chapter to create lookup tables.

  Lookup tables and their corresponding templates can be one-dimensional, two-dimensional, or three-dimensional. Delay arcs allow a maximum of two dimensions. Device degradation constraint tables allow only one dimension. Load-dependent constraint modeling requires three dimensions.

CMOS piecewise linear delay model

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

Scalable polynomial delay model

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

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

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

delay_model Attribute

To specify the CMOS delay model, use the `delay_model` attribute in the `library` group.

The `delay_model` attribute must be the first attribute in the library if a `technology` attribute is not present. Otherwise, it should follow the `technology` attribute.

Example

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

Defining the CMOS Nonlinear Delay Model Template

Table templates store common table information that multiple lookup tables can use. A table template specifies the table parameters and the breakpoints for each axis. Assign each template a name so that lookup tables can refer to it.

lu_table_template Group

Define your lookup table templates in the library group.

Syntax

```
lu_table_template(name) {
   variable_1 : value;
   variable_2 : value;
   variable_3 : value;
   index_1 ("float, ..., float");
   index_2 ("float, ..., float");
   index_3 ("float, ..., float");
}
```

Library Compiler includes a predefined lookup table template named scalar for tables having a single value.

Template Variables for Timing Delays

The table template specifying timing delays can have up to three variables (`variable_1`, `variable_2`, and `variable_3`). The variables indicate the parameters used to index into the lookup table along the first, second, and third table axes. The parameters are the input transition time of a constrained pin, the output net length and capacitance, and the output loading of a related pin.

The following is a list of the valid values (divided into sets) that you can assign to a table:

- Set 1:

  ```
  input_net_transition
  ```

- Set 2:

  ```
  total_output_net_capacitance
  output_net_length
  output_net_wire_cap
  output_net_pin_cap
  ```

- Set 3:

```
related_out_total_output_net_capacitance
related_out_output_net_length
related_out_output_net_wire_cap
related_out_output_net_pin_cap
```

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

For every table, the value you assign to a variable must be from a set different from the set from which you assign a value to the other variables. For example, if you want a two-dimensional table and you assign `variable_1` with the value `input_net_transition` from set 1, then you must assign `variable_2` with one of the values from set 2. Table 4-1 lists the combinations of values you can assign to the different variables for the varying dimensional tables specifying timing delays.

Table 4-1 Variable Values for Timing Delays

Template dimension	Variable_1	Variable_2	Variable_3
1	set1		
1	set2		
2	set1	set2	
2	set2	set1	
3	set1	set2	set3
3	set1	set3	set2
3	set2	set1	set3
3	set2	set3	set1
3	set3	set1	set2
3	set3	set2	set1

Template Variables for Load-Dependent Constraints

The table template specifying load-dependent constraints can have up to three variables (`variable_1`, `variable_2`, and `variable_3`). The variables indicate the parameters used to index into the lookup table along the first, second, and third table axes. The parameters are the input transition time of a constrained pin, the transition time of a related pin, and the output loading of a related pin.

The following is a list of the valid values (divided into sets) that you can assign to a table.

- Set 1:

  ```
  constrained_pin_transition
  ```

- Set 2:

  ```
  related_pin_transition
  ```

- Set 3:

  ```
  related_out_total_output_net_capacitance
  related_out_output_net_length
  related_out_output_net_wire_cap
  related_out_output_net_pin_cap
  ```

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

For every table, the value you assign to a variable must be from a set different from the set from which you assign a value to the other variables. For example, if you want a two-dimensional table and you assign `variable_1` with the value `constrained_pin_transition` from set 1, then you must assign `variable_2` with one of the values from set 2.

Table 4-2 lists the combination of values you can assign to the different variables for the varying dimensional tables specifying load-dependent constraints.

Table 4-2 Variable Values for Load-Dependent Constraint Tables

Template dimension	Variable_1	Variable_2	Variable_3
1	set1		
1	set2		
2	set1	set2	
2	set2	set1	
3	set1	set2	set3
3	set1	set3	set2
3	set2	set1	set3
3	set2	set3	set1

Table 4-2 Variable Values for Load-Dependent Constraint Tables (Continued)

Template dimension	Variable_1	Variable_2	Variable_3
3	set3	set1	set2
3	set3	set2	set1

Template Breakpoints

The index statements define the breakpoints for an axis. The breakpoints defined by `index_1` correspond to the parameter values indicated by `variable_1`. The breakpoints defined by `index_2` correspond to the parameter values indicated by `variable_2`. The breakpoints defined by `index_3` correspond to the parameter values indicated by `variable_3`.

The index values are lists of floating-point numbers greater than or equal to 0.0. The values in the list must be in increasing order. The size of each dimension is determined by the number of floating-point numbers in the indexes.

You must define at least one `index_1` in the `lu_table_template` group. For a one-dimensional table, use only `variable_1`.

Creating Lookup Tables

The rules for specifying lookup tables apply to delay arcs as well as to constraints. "Defining Delay Arcs With Lookup Tables" on page 4-51 shows the groups to use as delay lookup tables. See the sections on the various constraints for the groups to use as constraint lookup tables.

This is the syntax for lookup table groups:

```
lu_table(name) {
   index_1 ("float, ..., float");
   index_2 ("float, ..., float");
   index_3 ("float, ..., float");
   values("float, ..., float", ..., "float, ..., float");
}
```

These rules apply to lookup table groups:

- Each lookup table has an associated name for the `lu_table_template` it uses. The name of the template must be identical to the name defined in a library `lu_table_template` group.

- You can overwrite any or all of the indexes in a lookup table template, but the overwrite must occur before the actual definition of values.
- The delay value of the table is stored in a `values` attribute.
  - Transition table delay values must be 0.0 or greater. Propagation tables and cell tables can contain negative delay values.
  - In a one-dimensional table, represent the delay value as a list of `nindex_1` floating-point numbers.
  - In a two-dimensional table, represent the delay value as `nindex_1` x `nindex_2` floating-point numbers.
  - If a table contains only one value, you can use the predefined scalar table template as the template for that timing arc. To use the scalar table template, place the string scalar in your lookup table group statement, as shown in Example 4-5 on page 4-53.
- Each group of floating-point values enclosed in quotation marks represents a row in the table.
  - In a one-dimensional table, the number of floating-point values in the group must equal `nindex_1`.
  - In a two-dimensional table, the number of floating-point values in a group must equal `nindex_2` and the number of groups must equal `nindex_1`.
  - In a three-dimensional table, the total number of groups is `nindex_1` x `nindex_2` and each group contains as many floating-point numbers as `nindex_3`. In a three-dimensional table, the first group represents the value indexed by the (1, 1, 1) to the (1, 1, `nindex_3`) points in the index. The first `nindex_2` groups represent the value indexed by the (1, 1, 1) to the (1, `nindex_2`, `nindex_3`) points in the index. The rest of the groups are grouped in the same order.

Example 4-5 on page 4-53 shows a library that uses the CMOS nonlinear delay model to describe the delay.

Defining the Scalable Polynomial Delay Model Template

Polynomial templates store common format information that equations can use.

poly_template Group

You use a `poly_template` group to specify the equation variables, the variable ranges, the voltage mapping, and the piecewise data. Assign each `poly_template` group a unique name, so that equations in the `timing` group can refer to it.

Syntax

```
poly_template(name_id) {variables(variable_i_enum, ..., variable_n_enum)
variable_i_range(float, float)...variable_n_range(float, float)
   mapping(voltage_enum, power_rail_id)domain(domain_name_id) {calc_mode :
name_id ; variables(variable_i_enum)..., variable_n_enum)
variable_i_range(float, float)...variable_n_range(float, float)
mapping(voltage_enum, power_rail_id)}}
```

poly_template Variables

The `poly_template` group that defines timing delays can have up to *n* variables (variable_*i*, ..., variable_*n*), which you specify in the `variables` complex attribute. The variables you specify represent the following in the equation:

- The input transition time of a constrained pin
- The output net length and capacitance
- The output loading of a related pin
- The default power supply voltage
- The voltage*i* for multivoltage cells
- The temperature
- User parameters (parameter1...parameter5)

The following list shows the valid values, divided into four sets, that you can assign to variables in an equation:

- Set 1:

```
input_net_transition
constrained_pin_transition
```

- Set 2:

```
total_output_net_capacitance
output_net_length
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:

```
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

The delay model you use determines which set of attributes for delay model calculation you specify in a `timing` group. Table 4-3 shows the available delay models and the supported `timing` group attributes for each. See Chapter 3, "Delay Models," for detailed information about the delay models.

Table 4-3 timing Group Attributes in the Delay Models

Purpose	CMOS generic	CMOS piecewise linear	CMOS nonlinear/ scalable polynomial
To specify a default timing arc			`default_timing`
To identify timing arc startpoint	`related_pin` `related_ bus_pins`	`related_pin` `related_bus_pins`	`related_pin` `related_bus_pins`
To describe logical effect of input pin on output pin	`timing_sense`	`timing_sense`	`timing_sense`
To identify an arc as combinational or sequential	`timing_type`	`timing_type`	`timing_type`
To describe intrinsic delay on an output pin	`intrinsic_rise` `intrinsic_fall`	`intrinsic_rise` `intrinsic_fall`	(*inherent*)

To specify transition delay. (Used with propagation delay in CMOS nonlinear delay model.)	`rise_resistance`
	`fall_resistance`
		`rise_pin_resistance`
		`rise_delay_intercept`
		`fall_pin_resistance`
		`fall_delay_intercept`
			`rise_transition`
			`fall_transition`

Table 4-3 timing Group Attributes in the Delay Models (Continued)

Purpose	CMOS generic	CMOS piecewise linear	CMOS nonlinear/ scalable polynomial
To specify propagation delay in total cell delay. (Used with transition delay in CMOS nonlinear delay model.)			`rise_propagation` `fall_propagation`
To specify cell delay independent of transition delay			`cell_rise` `cell_fall`
To specify retain delay within the delay arc			`retaining_rise` `retaining_fall`
To describe incremental delay added to slope of input waveform	`slope_rise` `slope_fall`	`slope_rise` `slope_fall`	
To specify an output or I/O pin for load-dependency model			`related_output_pin`
To specify when a timing arc is active			`mode`

related_pin Simple Attribute

The `related_pin` attribute defines the pin or pins that are the startpoint of the timing arc. The primary use of `related_pin` is for multiple signals in ganged-logic timing. This attribute is a required component of all `timing` groups.

Example

```
pin (B){
   direction : output ;
   function : "A'";
   timing () {
      related_pin : "A" ;
      ...
   }
}
```

You can use the `related_pin` attribute statement as a shortcut for defining two identical timing arcs for a cell. For example, for a 2-input NAND gate with identical delays from both input pins to the output pin, you need to define only one timing arc with two related pins, as shown in the following example.

```
pin (Z) {
   direction : output;
   function : "(A * B)'" ;
   timing () {
      related_pin : "A B" ;
      ... timing information ...
   }
}
```

When you use a bus name in a `related_pin` attribute statement, the bus members or the range of members is distributed across all members of the parent bus. In the following example, the timing arcs described are from each element in bus A to each element in bus B: A(1) to B(1) and A(2) to B(2).

The width of the bus or range must be the same as the width of the parent bus.

```
bus (A) {
   bus_type : bus2;
   ...
}
bus (B) {
   bus_type : bus2;
   direction : output;
   function : "A'";
   timing () {
      related_pin : "A" ;
      ... timing information ...
   }
}
```

related_bus_pins Simple Attribute

The `related_bus_pins` attribute defines the pin or pins that are the startpoint of the timing arc. The primary use of `related_bus_pins` is for module generators.

Example

In this example, the timing arcs described are from each element in bus A to each element in bus B: A(1) to B(1), A(1) to B(2), A(1) to B(3), and so on. The widths of bus A and bus B do not need to be identical.

```
bus (A){
   bus_type : bus2;
   ...
}
```

```
bus (B){
   bus_type : bus4;
   direction : output ;
   function : "A";
   timing () {
      related_bus_pins : "A" ;
      ... timing information ...
   }
}
```

timing_sense Simple Attribute

The `timing_sense` attribute describes the way an input pin logically affects an output pin. A timing analyzer uses this attribute to track the polarity transition of an element during path analysis.

Example

```
timing () {
   timing_sense : positive_unate;
}
```

Design Compiler typically derives the `timing_sense` value from the pin's logic function. For example, the value derived for an AND gate is `positive_unate`, the value for a NAND gate is `negative_unate`, and the value for an XOR gate is `non_unate`.

A function is *unate* if a rising (or falling) change on a positive unate input variable causes the output function variable to rise (or fall) or not change. A rising (or falling) change on a negative unate input variable causes the output function variable to fall (or rise) or not change. For a nonunate variable, further state information is required to determine the effects of a particular state transition.

Do not define the `timing_sense` value of a pin, except when you must override the derived value or you are characterizing a noncombinational gate such as a three-state component. For example, you might define the timing sense manually when you model multiple paths between an input pin and an output pin, as in an XOR gate.

It is possible that one path is positive unate while another is negative unate. In this case, the first timing arc gets a `positive_unate` designation and the second arc gets a `negative_unate` designation.

Note:

When `timing_sense` describes the transition edge used to calculate delay for the `three_state_enable` or `three_state_disable` pin, it has a meaning different from its traditional one. If a 1 value on the control pin of a three-state cell causes a Z value on the output pin, `timing_sense` is `positive_unate` for the `three_state_disable` timing arc and `negative_unate` for the `three_state_enable` timing arc. If a 0 value on the control

pin of a three-state cell causes a Z value on the output pin, `timing_sense` is `negative_unate` for the `three_state_disable` timing arc and positive_unate for the `three_state_enable` timing arc.

If a `related_pin` is an output pin, you must define `timing_sense` for that pin.

timing_type Simple Attribute

The `timing_type` attribute distinguishes between combinational and sequential cells, by defining the type of timing arc. If this attribute is not assigned, the cell is considered combinational.

You must distinguish between combinational and sequential timing types, because each type serves a different purpose.

Design Compiler uses the combinational timing arcs information to calculate the physical delays in timing propagation and to trace paths. The timing analyzer uses path tracing arcs for circuit timing analysis.

Design Compiler uses the sequential timing arcs information to determine rule-based design optimization constraints. More information on optimization constraints is available in the Design Compiler documentation.

Example

```
timing () {
   timing_type : rising_edge;
}
```

Table 4-4 shows the valid values for the `timing_type` attribute:

Table 4-4 Valid Values for the timing_type Attribute

Value	Cells where used	Function
`combinational`	combinational	Use this timing type value to describe 1-to-0 or 0-to-1 timing arcs. This is the default when there is no `timing_type` attribute in the `timing` group.
`combinational_rise` `combinational_fall`	combinational	Use these timing type values to describe 0-to-1 or 1-to-0 timing arcs, respectively.

`three_state_disable`
`three_state_enable`
`three_state_disable_rise`
`three_state_disable_fall`
`three_state_enable_rise`
`three_state_enable_fall`
	combinational	Use these values to describe three-state output pins.

Table 4-4 Valid Values for the timing_type Attribute (Continued)

Value	Cells where used	Function
`rising_edge` `falling_edge`	sequential	Use these values to specify which edge of the input signal causes a transition on the output pin.
`preset` `clear`	sequential	Use these values to affect fall or rise arrival time on a timing arc's endpoint pin.
`hold_rising` `hold_falling`	sequential	Use these values to designate the edge of the related pin for the hold check.
`setup_rising` `setup_falling`	sequential	Use these values to designate the edge of the related pin for the setup check on clocked elements.
`recovery_rising` `recovery_falling`	sequential	Use these values to designate the edge of the related pin for the recovery time check.
`skew_rising` `skew_falling`	sequential	Use these values to specify constraints defining the maximum separation between two clock signals. Use these values for simulation only (not for VITAL models).
`removal_rising` `removal_falling`	sequential	Use these values to designate the edge that defines the removal time.
`minimum_period`	sequential	Use these values to model minimum period constraints for an input pin.
`min_pulse_width`	combinational or sequential	Use these values to model pulse width constraints for an input pin.
`max_clock_tree_path` `min_clock_tree_path`	sequential	Use these values to define the minimum and maximum clock tree path constraints.

`non_seq_setup_rising`
`non_seq_setup_falling`
`non_seq_hold_rising`
`non_seq_hold_falling`
	combinational or sequential	Use these values to specify setup and hold constraints on data arrival unrelated to clock signals.

Table 4-4 Valid Values for the timing_type Attribute (Continued)

Value	Cells where used	Function

`nochange_high_high`
`nochange_high_low`
`nochange_low_high`
`nochange_low_low`
	sequential	Use these values to specify no-change constraints on signal pulses with relation to clock pulses.

The following sections show the `timing_type` attribute values for the following types of timing arcs:

- Combinational
- Sequential
- Nonsequential
- No-change

Values for Combinational Timing Arcs

The timing type and timing sense define the signal propagation pattern. The default timing type is combinational.

Timing type	Timing sense		
	positive_unate	**negative_unate**	**non_unate**
`combinational`	R->R,F->F	R->F,F->R	{R,F}->{R,F}
`combinational_rise`	R->R	F->R	{R,F}->R
`combinational_fall`	F->F	R->F	{R,F}->F
`three_state_disable`	R->{0Z,1Z}	F->{0Z,1Z}	{R,F}->{0Z,1Z}
`three_state_enable`	R->{Z0,Z1}	F->{Z0,Z1}	{R,F}->{Z0,Z1}
`three_state_disable_rise`	R->0Z	F->0Z	{R,F}->0Z
`three_state_disable_fall`	R->1Z	F->1Z	{R,F}->1Z
`three_state_enable_rise`	R->Z1	F->Z1	{R,F}->Z1
`three_state_enable_fall`	R->Z0	F->Z0	{R,F}->Z0

Values for Sequential Timing Arcs

You use sequential timing arcs to model the timing requirements for sequential cells.

`rising_edge`

Identifies a timing arc whose output pin is sensitive to a rising signal at the input pin.

`falling_edge`

Identifies a timing arc whose output pin is sensitive to a falling signal at the input pin.

`preset`

Preset arcs affect only the rise arrival time of the arc's endpoint pin. A preset arc implies that you are asserting a logic 1 on the output pin when the designated related pin is asserted.

`clear`

Clear arcs affect only the fall arrival time of the arc's endpoint pin. A clear arc implies that you are asserting a logic 0 on the output pin when the designated related pin is asserted.

`hold_rising`

Designates the rising edge of the related pin for the hold check.

`hold_falling`

Designates the falling edge of the related pin for the hold check.

`setup_rising`

Designates the rising edge of the related pin for the setup check on clocked elements.

`setup_falling`

Designates the falling edge of the related pin for the setup check on clocked elements.

`recovery_rising`

Uses the rising edge of the related pin for the recovery time check. The clock is rising-edge-triggered.

`recovery_falling`

Uses the falling edge of the related pin for the recovery time check. The clock is falling-edge-triggered.

`skew_rising`

The timing constraint interval is measured from the rising edge of the reference pin (specified in `related_pin`) to a transition edge of the parent pin in the `timing` group. The `intrinsic_rise` value is the maximum skew time between the reference pin rising and the parent pin rising. The `intrinsic_fall` value is the maximum skew time between the reference pin falling and the parent pin falling.

`skew_falling`

The timing constraint interval is measured from the falling edge of the reference pin (specified in `related_pin`) to a transition edge of the parent pin in the `timing` group. The `intrinsic_rise` value is the maximum skew time between the reference pin falling and the parent pin rising. The `intrinsic_fall` value is the maximum skew time between the reference pin falling and the parent pin falling.

`removal_rising`

Used when the cell is a low-enable latch or a rising-edge-triggered flip-flop. For active-low asynchronous control signals, define the removal time with the `intrinsic_rise` attribute. For active-high asynchronous control signals, define the removal time with the `intrinsic_fall` attribute.

`removal_falling`

Used when the cell is a high-enable latch or a falling-edge-triggered flip-flop. For active-low asynchronous control signals, define the removal time with the `intrinsic_rise` attribute. For active-high asynchronous control signals, define the removal time with the `intrinsic_fall` attribute.

`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 “Sample Library With timing_type Statements” (Example 4-14 on page 4-94).

`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. Library Compiler checks that you have not specified a `related_pin` attribute.

`min_clock_tree_path`

Used in `timing` groups under a clock pin. Defines the minimum clock tree path constraint. Library Compiler checks that you have not specified a `related_pin` attribute.

Values for Nonsequential Timing Arcs

In some nonsequential cells, the setup and hold timing constraints are specified on the data pin with a nonclock pin as the related pin. The signal of a pin must be stable for a specified period of time before and after another pin of the same cell range state for the cell to function as expected.

`non_seq_setup_rising`

Defines (with `non_seq_setup_falling`) the timing arcs used for setup checks between pins with nonsequential behavior. The related pin in a timing arc is used for the timing check. The `_rising` designation tells Design Compiler that the rising edge of the related pin is active for the setup check.

`non_seq_setup_falling`

Defines (with `non_seq_setup_rising`) the timing arcs used for setup checks between pins with nonsequential behavior. The related pin in a timing arc is used for the timing check. The `_falling` designation tells Design Compiler that the falling edge of the related pin is active for the setup check.

`non_seq_hold_rising`

Defines (with `non_seq_hold_falling`) the timing arcs used for hold checks between pins with nonsequential behavior. The related pin in a timing arc is used for the timing check. The `_rising` designation tells Design Compiler that the rising edge of the related pin is active for the hold check.

`non_seq_hold_falling`

Defines (with `non_seq_hold_rising`) the timing arcs used for hold checks between pins with nonsequential behavior. The related pin in a timing arc is used for the timing check. The `_falling` designation tells Design Compiler that the falling edge of the related pin is active for the hold check.

Values for No-Change Timing Arcs

You use no-change timing arcs to model the timing requirement for latch devices with latch-enable signals. The four no-change timing types define the pulse waveforms of both the constrained signal and the related signal in standard CMOS and nonlinear CMOS delay models. The information is used in static timing verification during synthesis.

`nochange_high_high`

Indicates a positive pulse on the constrained pin and a positive pulse on the related pin.

`nochange_high_low`

Indicates a positive pulse on the constrained pin and a negative pulse on the related pin.

`nochange_low_high`

Indicates a negative pulse on the constrained pin and a positive pulse on the related pin.

`nochange_low_low`

Indicates a negative pulse on the constrained pin and a negative pulse on the related pin.

mode Complex Attribute

You define the `mode` attribute within a `timing` group. A `mode` attribute pertains to an individual timing arc. The timing arc is active when `mode` is instantiated with a name and a value. You can specify multiple instances of the mode attribute, but only one instance for each timing arc.

Syntax

```
mode (mode_name, mode_value);
```

Library Compiler issues an error message if the *mode_name* and *mode_value* strings are not already defined by a `mode_definition` group in the cell.

Example

```
timing() {
   mode(rw, read);
}
```

Example 4-1 shows a `mode` instance description.

Example 4-1 A mode Instance Description

```
pin(my_outpin) {
   direction : output;
      timing() {
         related_pin : b;
         timing_sense : non_unate;
         mode(rw, read);
         cell_rise(delay3x3) {
            values("1.1, 1.2, 1.3", "2.0, 3.0, 4.0", "2.5, 3.5, 4.5");
         }
         rise_transition(delay3x3) {
            values("1.0, 1.1, 1.2", "1.5, 1.8, 2.0", "2.5, 3.0, 3.5");
         }
         cell_fall(delay3x3) {
            values("1.1, 1.2, 1.3", "2.0, 3.0, 4.0", "2.5, 3.5, 4.5");
         }
         fall_transition(delay3x3) {
            values("1.0, 1.1, 1.2", "1.5, 1.8, 2.0", "2.5, 3.0, 3.5");
         }
      }
   }
}
```

Example 4-2 shows multiple `mode` descriptions.

Example 4-2 Multiple mode Descriptions

```
library (MODE_EXAMPLE) {
   delay_model              : "table_lookup";
   time_unit                : "1ns";
```

```
voltage_unit             : "1V";
current_unit             : "1mA";
pulling_resistance_unit  : "1kohm";
leakage_power_unit       : "1nW" ;
capacitive_load_unit       (1, pf);
nom_process              :  1.0;
nom_voltage              :  1.0;
nom_temperature          : 125.0;
slew_lower_threshold_pct_rise  :  10 ;
slew_upper_threshold_pct_rise  :  90 ;
input_threshold_pct_fall       :  50 ;
output_threshold_pct_fall      :  50 ;
input_threshold_pct_rise       :  50 ;
output_threshold_pct_rise      :  50 ;
slew_lower_threshold_pct_fall  :  10 ;
slew_upper_threshold_pct_fall  :  90 ;
slew_derate_from_library       :  1.0 ;
cell (mode_example) {
   mode_definition(RAM_MODE) {
      mode_value(MODE_1) {
      }
      mode_value(MODE_2) {
      }
      mode_value(MODE_3) {
      }
      mode_value(MODE_4) {
      }
   }
   interface_timing : true;
   dont_use         : true;
   dont_touch       : true;
   pin(Q) {
      direction            : output;
      max_capacitance     : 2.0;
      three_state          : "!OE";
      timing() {
         related_pin     : "CK";
         timing_sense    : non_unate
         timing_type     : rising_edge
         mode(RAM_MODE,"MODE_1 MODE_2");
         cell_rise(scalar) {
            values( " 0.0 ");
         }
         cell_fall(scalar) {
            values( " 0.0 ");
         }
         rise_transition(scalar) {
            values( " 0.0 ");
         }
         fall_transition(scalar) {
            values( " 0.0 ");
         }
      }
```

```
      timing() {
         related_pin     : "OE";
         timing_sense    : positive_unate
         timing_type     : three_state_enable
         mode(RAM_MODE, " MODE_2 MODE_3");
         cell_rise(scalar) {
            values( " 0.0 ");
         }
         cell_fall(scalar) {
            values( " 0.0 ");
         }
         rise_transition(scalar) {
            values( " 0.0 ");
         }
         fall_transition(scalar) {
            values( " 0.0 ");
         }
      }
      timing() {
         related_pin     : "OE";
         timing_sense    : negative_unate
         timing_type     : three_state_disable
         mode(RAM_MODE, MODE_3);
         cell_rise(scalar) {
            values( " 0.0 ");
         }
         cell_fall(scalar) {
            values( " 0.0 ");
         }
         rise_transition(scalar) {
            values( " 0.0 ");
         }
         fall_transition(scalar) {
            values( " 0.0 ");
         }
      }
   }
   pin(A) {
      direction            : input;
      capacitance          : 1.0;
      max_transition       : 2.0;
      timing() {
         timing_type       : setup_rising;
         related_pin       : "CK";
         mode(RAM_MODE, MODE_2);
         rise_constraint(scalar) {
            values( " 0.0 ");
         }
         fall_constraint(scalar) {
            values( " 0.0 ");
         }
      }
      timing() {
```

```
      timing_type       : hold_rising;
      related_pin       : "CK";
      mode(RAM_MODE, MODE_2);
      rise_constraint(scalar) {
         values( " 0.0 ");
      }
      fall_constraint(scalar) {
         values( " 0.0 ");
      }
   }
}
pin(OE) {
   direction            : input;
   capacitance          : 1.0;
   max_transition       : 2.0;
}
pin(CS) {
   direction            : input;
   capacitance          : 1.0;
   max_transition       : 2.0;
   timing() {
      timing_type       : setup_rising;
      related_pin       : "CK";
      mode(RAM_MODE, MODE_1);
      rise_constraint(scalar) {
         values( " 0.0 ");
      }
      fall_constraint(scalar) {
         values( " 0.0 ");
      }
   }
   timing() {
      timing_type       : hold_rising;
      related_pin       : "CK";
      mode(RAM_MODE, MODE_1);
      rise_constraint(scalar) {
         values( " 0.0 ");
      }
      fall_constraint(scalar) {
         values( " 0.0 ");
      }
   }
}
pin(CK) {
   timing() {
      timing_type : "min_pulse_width";
      related_pin : "CK";
      mode(RAM_MODE , MODE_4);
      fall_constraint(scalar) {
         values( " 0.0 ");
      }
      rise_constraint(scalar) {
         values( " 0.0 ");
```

```
            }
         }
         timing() {
            timing_type : "minimum_period";
            related_pin : "CK";
            mode(RAM_MODE , MODE_4);
            rise_constraint(scalar) {
               values( " 0.0 ");
            }
            fall_constraint(scalar) {
               values( " 0.0 ");
            }
         }
         clock                : true;
         direction            : input;
         capacitance          : 1.0;
         max_transition       : 1.0;
      }
      cell_leakage_power : 0.0;
   }
}
```

Describing Three-State Timing Arcs

Three-state arcs describe a three-state output pin in a cell.

Describing Three-State-Disable Timing Arcs

To designate a three-state-disable timing arc when defining a three-state pin,

1. Assign `related_pin` to the enable pin of the three-state function.
2. Define the 0-to-Z propagation time with the `intrinsic_rise` statement.
3. Define the 1-to-Z propagation time with the `intrinsic_fall` statement.
4. Include the `timing_type` : `three_state_disable` statement.

Example

```
timing () {
   related_pin : "OE" ;
   timing_type : three_state_disable ;
   intrinsic_rise : 1.0 ; /* 0 to Z time */
   intrinsic_fall : 1.0 ; /* 1 to Z time */
}
```

Note:

The `timing_sense` attribute, which describes the transition edge used to calculate delay for a timing arc, has a nontraditional meaning when it is included in a `timing` group that also contains a `three_state_disable` attribute. See "timing_sense Simple Attribute" on page 4-27 for more information.

Example 4-3 describes the timing model for an inverter with a three-state output and active-high enable in a CMOS generic library.

Describing Three-State-Enable Timing Arcs

To designate a three-state-enable timing arc when defining a three-state pin,

1. Assign `related_pin` to the enable pin of the three-state function.
2. Define the Z-to-1 propagation time with the `intrinsic_rise` statement.
3. Define the Z-to-0 propagation time with the `intrinsic_fall` statement.
4. Include the `timing_type` : `three_state_enable` statement.

Example

```
timing () {
   related_pin : "OE" ;
   timing_type : three_state_enable ;
   intrinsic_rise : 1.0 ; /* Z-to-1 time */
   intrinsic_fall : 1.0 ; /* Z-to-0 time */
}
```

Note:

The `timing_sense` attribute that describes the transition edge used to calculate delay for a timing arc has a nontraditional meaning when it is included in a `timing` group that also contains a `three_state_enable` attribute. See "timing_sense Simple Attribute" on page 4-27 for more information.

Example 4-3 shows a three-state cell with both disable and enable timing arcs.

Example 4-3 Three-State Cell With Disable and Enable Timing Arcs

```
library(example){
   date : "January 14, 2002";
   revision : 2000.01;
   technology (cmos);
   ...
   cell(TRI_INV2) {
      area : 3;
      pin(A) {
         direction : input;
         capacitance : 2;
      }
      pin(E) {
         direction : input;
         capacitance : 2;
      }
      pin(Z) {
         direction : output;
         function : "A'";
         three_state : "E'";
         timing() {
            intrinsic_rise : 0.39;
            intrinsic_fall : 0.75;
```

```
            rise_resistance : 0.15;
            fall_resistance : 0.06;
            related_pin : "A";
         }
         timing() {
            timing_type : three_state_enable;
            intrinsic_rise : 0.39;
            intrinsic_fall : 0.75;
            rise_resistance : 0.15;
            fall_resistance : 0.06;
            related_pin : "E";
         }
         timing() {
            timing_type : three_state_disable;
            intrinsic_rise : 0.25;
            intrinsic_fall : 0.35;
            related_pin : "E";
         }
      }
   }
}
```

Describing Edge-Sensitive Timing Arcs

Edge-sensitive timing arcs, such as the arc from the clock on a flip-flop, are identified by the following values of the `timing_type` attribute in the `timing` group.

rising_edge

Identifies a timing arc whose output pin is sensitive to a rising signal at the input pin.

falling_edge

Identifies a timing arc whose output pin is sensitive to a falling signal at the input pin.

These arcs are path-traced; the path tracer propagates only the active edge (rise or fall) path values along the timing arc.

See “timing_type Simple Attribute” on page 4-28 for information about the `timing_type` attribute.

The following example shows the timing arc for the QN pin in a JK flip-flop in a CMOS library using a CMOS generic delay model.

Example

```
pin(QN) {
   direction : output ;
   function : "IQN" ;
   timing() {
      related_pin : "CP" ;
```

```
        timing_type : rising_edge ;
        intrinsic_rise : 1.29 ;
        intrinsic_fall : 1.61 ;
        rise_resistance : 0.1723 ;
        fall_resistance : 0.0553 ;
    }
}
```

The QN pin makes a transition after the clock signal rises.

Describing Preset and Clear Timing Arcs

In normal operation, preset arcs affect only the rise arrival time on the arc's endpoint pin and clear arcs affect only the fall arrival time. You can use these timing arcs for asynchronous reset and clear pins on a flip-flop or a level-sensitive latch. Accordingly, a rise time must be defined for a preset arc and a fall time must be defined for a clear arc. Depending on the delay model defined in the library, the rise time for a preset arc is defined by creation of a rise delay table or by setting of the `intrinsic_rise` value. The fall timing for a clear arc is defined similarly. Generally, fall timing for preset arcs and rise timing for clear arcs should not exist.

In some cases, however, a transition on a preset pin can cause a falling transition at the arc's endpoint pin and a transition on a clear pin can cause a rising transition. For example, for the preset arc, consider what happens when a preset signal is de-asserted on a level-sensitive latch while the latch input is at logic 0 and the latch is in its transparent mode. In such a case, the output of the latch is a falling transition caused by the de-assertion of the preset signal. For the clear arc case, consider what happens when the clear signal on an edge-triggered flip-flop is de-asserted while the preset signal is still asserted and the clear signal dominates the preset signal. In such a case, the flip-flop output is a rising transition caused by the de-assertion of the clear signal.

In normal operation, the Design Compiler and PrimeTime tools ignore paths with a falling transition at a preset arc endpoint pin and paths with a rising transition at a clear arc endpoint pin. This is done without a warning message. To force PrimeTime to consider such paths, a fall timing must be defined for a preset arc and a rise timing must be defined for a clear arc. However, these rise and fall timing values should be defined only for a case similar to one of the two examples described above.

Describing Preset Arcs

Preset arcs affect only the rise arrival time of the arc's endpoint pin. A preset arc means that you are asserting a logic 1 on the output pin when the designated `related_pin` is asserted.

To designate a preset arc,

1. Select the preset value for the `timing_type` attribute.

```
timing_type : preset;
```

2. Assign the appropriate value to the `timing_sense` attribute. The valid values are

positive_unate

Indicates that the rise arrival time of the arc's source pin is used to calculate the arc's delay. This calculation produces the rise arrival time on the arc's endpoint pin. In the case of slope delays, the source pin's rise transition time is added to the arc's delay. The source pin is active-high.

negative_unate

Indicates that the fall arrival time of the arc's source pin is used to calculate the arc's delay. This calculation produces the rise arrival time on the arc's endpoint pin. In the case of slope delays, the source pin's fall transition time is added to the arc's delay. The source pin is active-low.

non_unate

Indicates that the maximum of the rise and fall arrival times of the arc's source pin is used to calculate the arc's delay. This calculation produces the maximum arrival time on the arc's endpoint pin. In the case of slope delays, the maximum of the source pin's rise and fall transition times is added to the arc's delay.

```
timing_sense : negative_unate;
```

If you are specifying a preset arc for a cell that has both clear and preset pins, define the fall time for the preset arc to accurately model the dynamic timing behavior (for simulation tools) during the time when both clear and preset signals are active.

Example 4-15 on page 4-99 and Example 4-16 on page 4-101 show how to specify preset arcs.

Describing Clear Arcs

Clear arcs affect only the fall arrival time of the arc's endpoint pin. A clear arc means that you are asserting a logic 0 on the output pin when the designated `related_pin` is asserted.

In normal operation, a clear arc causes a falling transition at the output pin. In some cases, however, a transition on a clear pin can cause a rising transition at the output pin. For example, consider what happens when CDN is active-low clear and SDN is active-low preset:

```
SDN = 0
CDN transition from 0 to 1
```

The result is

```
Q transition from 0 to 1
```

In such a case, you can define a rise delay for the clear arc.

To designate a clear arc,

1. Select the clear value for the `timing_type` attribute.

   ```
   timing_type : clear;
   ```

2. Assign the appropriate value to the `timing_sense` attribute. These are valid values:

positive_unate

Indicates that the fall arrival time of the arc's source pin is used to calculate the arc's delay. This calculation produces the fall arrival time on the arc's endpoint pin. In the case of slope delays, the source pin's fall transition time is added to the arc's delay. The source pin is active-low.

negative_unate

Indicates that the rise arrival time of the arc's source pin is used to calculate the arc's delay. This calculation produces the fall arrival time on the arc's endpoint pin. In the case of slope delays, the source pin's rise transition time is added to the arc's delay. The source pin is active-high.

non_unate

Indicates that the maximum of the rise and fall arrival times of the arc's source pin is used in calculating the arc's delay. This calculation produces the maximum fall arrival time on the arc's endpoint pin. In the case of slope delays, the maximum of the source pin's rise and fall transition times is added to the arc's delay.

```
timing_sense : positive_unate;
```

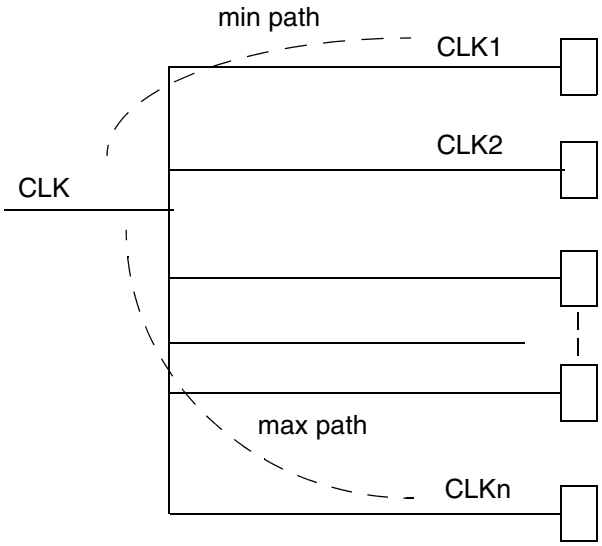
If you are specifying a clear arc for a cell that has both clear and preset pins, define the rise time for the clear arc to accurately model the dynamic timing behavior (for simulation tools) when both clear and preset signals are active.

Example 4-15 on page 4-99 and Example 4-16 on page 4-101 show how to specify clear arcs.

Describing Clock Insertion Delay

Arrival timing paths are the timing paths from an input clock pin to the clock pins that are internal to a cell. The arrival timing paths describe the minimum and the maximum timing constraint for a pin driving an internal clock tree for each input transition, as shown in Figure 4-5.

Figure 4-5 Minimum and Maximum Clock Tree Paths



The `max_clock_tree_path` and `min_clock_tree_path` attributes let you define the maximum and minimum clock tree path constraints.

The clock tree path for any one clock can have up to eight values depending on the unateness of the pins and the fastest and slowest paths.

You can use either lookup tables or scalable polynomials to model the cell delays. Lookup tables are indexed only by the input transition time of the clock. Polynomials can include only the following variables in piecewise domains: input_net_transition, voltage, voltage*i*, and temperature.

For `timing` groups whose `timing_sense` attribute is set to `non_unate` and whose only variable is input_net_transition, use pairs of lookup tables or polynomials to model both positive unate and negative unate.

Describing Intrinsic Delay

The intrinsic delay of an element is the zero-load (fixed) component of the total delay equation. Intrinsic delay attributes have different meanings, depending on whether they are for an input or an output pin. See “Intrinsic Delay” on page 3-6.

When describing an output pin, the intrinsic delay attributes define the fixed delay from input to output pin. These values are used to calculate the intrinsic delay of the total delay equation.

When describing an input pin, such as in a setup or hold timing arc, intrinsic attributes define the timing requirements for that pin. Pin D in Example 4-15 on page 4-99 illustrates the intrinsic delay attributes used as timing constraints. Timing constraints are not used in the delay equation.

In the CMOS Generic Delay Model

The `intrinsic_rise` and `intrinsic_fall` attributes describe the intrinsic delay of a pin when you use the CMOS generic delay model. See “CMOS Generic Delay Model” on page 3-2 for details.

intrinsic_rise Simple Attribute

On an output pin, this value defines

- The 0-to-1 propagation time for a combinational cell
- The 0-to-Z propagation time for a three-state-disable timing type
- The Z-to-1 propagation time for a three-state-enable timing type

On an input pin, this value defines a setup, hold, or recovery timing requirement for a logic 0-to-1 transition.

Example

```
intrinsic_rise : 1.0;
```

intrinsic_fall Simple Attribute

On an output pin, this value defines

- The 1-to-0 propagation time for a combinational cell
- The 1-to-Z propagation time for a three-state-disable timing type
- The Z-to-0 propagation time for a three-state-enable timing type

On an input pin, this value defines a setup, hold, or recovery timing requirement for a logic 1-to-0 transition.

Example

This example shows the `timing` group for a 2-input NAND gate with an intrinsic rise delay of 0.58 and an intrinsic fall delay of 0.69.

```
pin (Z) {
   direction : output ;
   function : "(A * B)'" ;
   timing () {
      related_pin : "A B" ;
      intrinsic_rise : 0.58 ;
      intrinsic_fall : 0.69 ;
      ...
   }
}
```

In the CMOS Piecewise Linear Delay Model

You describe the intrinsic delay in the CMOS piecewise linear delay model the same way you describe it in the CMOS generic delay model. See “intrinsic_rise Simple Attribute” on page 4-46 and “intrinsic_fall Simple Attribute” on page 4-47 for details.

In the CMOS Nonlinear Delay Model

The description of intrinsic delay is inherent in the lookup tables you create for this delay model. See “Defining the CMOS Nonlinear Delay Model Template” on page 4-18 to find out how to create and use templates for lookup tables in a library, using the CMOS nonlinear delay model.

In the Scalable Polynomial Delay Model

The description of intrinsic delay is inherent in the polynomials you create for this delay model. See "Defining the Scalable Polynomial Delay Model Template" on page 4-22 to learn how to create and use templates for scalable polynomials in a library, using the CMOS scalable polynomial nonlinear delay model.

Describing Transition Delay

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

The components of the total delay calculation depend on the timing delay model used. Include the transition delay attributes that apply to the delay model you are using.

In the CMOS Generic Delay Model

Use the following `timing` group attributes exclusively for generic delay models. In the attribute statements, the value is a positive floating-point number for the delay time per load unit.

rise_resistance Simple Attribute

This value represents the output resistance, or drive capability, for a logic 0-to-1 transition.

fall_resistance Simple Attribute

This value represents the output resistance, or drive capability, for a logic 1-to-0 transition.

Note:

You cannot specify a resistance unit in the library. Instead, the resistance unit is derived from the ratio of the `time_unit` value to the `capacitive_load_unit` value.

Example

This example is a combinational timing description of a NAND gate in a library that uses a standard CMOS generic delay model.

```
cell (NAND2) {
  area : 1 ;
  pin(A, B) {
     direction   : input ;
     capacitance : 1 ;
  }
  pin(Z) {
```

```
      direction : output ;
      function : "(A * B)'" ;
      timing() {
         intrinsic_rise : 0.17 ;
         intrinsic_fall : 0.10 ;
         rise_resistance : 0.65 ;
         fall_resistance : 0.18 ;
         related_pin     : "A" ;
      }
      timing() {
         intrinsic_rise : 0.21 ;
         intrinsic_fall : 0.12 ;
         rise_resistance : 0.65 ;
         fall_resistance : 0.18 ;
         related_pin     : "B" ;
      }
   }
}
```

In the CMOS Piecewise Linear Delay Model

When using a piecewise linear delay model, you must include the `piece_define` attribute statement in the `library` group; the `piece_type` attribute statement is optional (see the *Library Compiler Methodology and Modeling Functionality in Technology Libraries User Guide*). See "CMOS Piecewise Linear Delay Model" on page 3-26 for the delay equations.

The transition delay for piecewise linear equations is modeled with delay-intercept attributes and pin-resistance attributes. Delay-intercept attributes define the intercept for vendors that use slope- or intercept-type timing equations. Pin-resistance attributes describe the resistance encountered during logic transitions.

rise_delay_intercept Complex Attribute

This attribute defines the intercept for vendors using slope- or intercept-type timing equations. The value of this attribute is added to the rising edge in the delay equation.

fall_delay_intercept Complex Attribute

This attribute defines the intercept for vendors using slope- or intercept-type timing equations. The value of this attribute is added to the falling edge in the delay equation.

rise_pin_resistance Complex Attribute

This attribute defines the drive resistance applied to pin loads in the rising edge in the transition delay equation.

fall_pin_resistance Complex Attribute

This attribute defines the drive resistance applied to pin loads in the falling edge in the transition delay equation.

Syntax

This is the syntax for the following `timing` group attribute statements in which *`piece`* is an integer defining the wire segment in accordance with the `piece_define` statement and *`value`* is a floating-point number.

```
attribute_name (piece, "value");
```

Example 4-4 shows how to define a NOR gate in a library that uses a CMOS piecewise linear delay model.

Example 4-4 Combinational Timing Description (Piecewise Linear Delay Model)

```
library (example) {
date : "August 14, 2002" ;
revision : 2000.05;
piece_define ( "0 15 40" ) ;
cell(NOR2) {
   area : 1;
   pin(A, B) {
      direction : input;
      capacitance : 1;
   }
   pin(Z) {
      direction : output;
      function : "(A+B)'";
      timing() {
         intrinsic_rise : 0.38;
         intrinsic_fall : 0.15;
         rise_delay_intercept(0,"1.0");  /* piece 0 */
         rise_delay_intercept(1,"0.0");  /* piece 1 */
         rise_delay_intercept(2,"-1.0"); /* piece 2 */
         fall_delay_intercept(0,"1.0");  /* piece 0 */
         fall_delay_intercept(1,"0.0");  /* piece 1 */
         fall_delay_intercept(2,"-1.0"); /* piece 2 */
         rise_pin_resistance(0,"0.25");  /* piece 0 */
         rise_pin_resistance(1,"0.50");  /* piece 1 */
         rise_pin_resistance(2,"1.00");  /* piece 2 */
         fall_pin_resistance(0,"0.25");  /* piece 0 */
        fall_pin_resistance(1,"0.50");  /* piece 1 */
        fall_pin_resistance(2,"1.00");  /* piece 2 */
         related_pin : "A B";
      }
   }
}
```

In the CMOS Nonlinear Delay Model

Transition time is the time it takes for an output signal to make a transition between the high and low logic states. With nonlinear delay models, it is computed by table lookup and interpolation. Transition delay is a function of capacitance at the output pin and input transition time.

Defining Delay Arcs With Lookup Tables

These `timing` group attributes provide valid lookup tables for delay arcs:

- `cell_rise`
- `cell_fall`
- `rise_propagation`
- `fall_propagation`
- `retaining_rise`
- `retaining_fall`
- `retain_rise_slew`
- `retain_fall_slew`

  Note:

  For `timing` groups with timing type clear, only fall groups are valid. For `timing` groups with timing type preset, only rise groups are valid.

There are two methods for defining delay arcs. Choose the method that best fits your library data characterization. See Chapter 3, "Delay Models," and the *Library Compiler Technology and Symbol Libraries Reference Manual* for details about the way delays are computed with lookup tables.

Method 1

To specify cell delay independently of transition delay, use one of these `timing` group attributes as your lookup table:

- `cell_rise`
- `cell_fall`

Method 2

To specify transition delay as a term in the total cell delay, use one of these `timing` group attributes as your lookup table:

- `rise_propagation`

- `fall_propagation`

cell_rise and cell_fall Groups

The cell delay is usually a function of both output loading and input transition time.

These groups define cell delay lookup tables (independently of transition delay) in CMOS nonlinear timing models. If you define cell delay tables for a timing arc, you cannot specify propagation delay tables for that arc.

Example

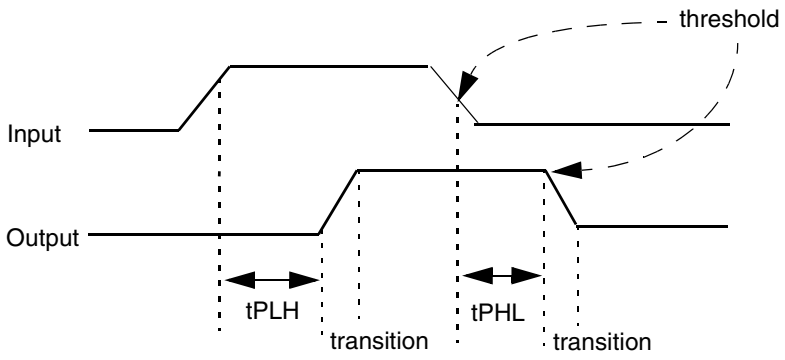
```
cell (general) {
   pin(a) {
      direction: output;
      timing() {
         cell_rise(basic_template) {
            values ("0.0, 0.13, 0.17, 0.19", "0.21, 0.23, \
                     0.30, 0.41", "0.22, 0.31, 0.35, 0.47",\
                     "0.33, 0.37, 0.45, 0.50");
            }
         }
      }
   }
}
```

rise_propagation and fall_propagation Groups

The propagation delay is the delay between the thresholds in the transition at the input of a gate and the start of the output transition, as shown in Figure 4-6.

If you define propagation delay tables for a timing arc, you cannot specify cell delay tables for that arc.

Figure 4-6 Propagation Delay Time Waveforms



tPLH - propagation delay time for a low-to-high transition

tPHL - propagation delay time for a high-to-low transition

Example

```
pin(Z) {
   ...
   timing() {
      ...
      rise_propagation(scalar) {
         values ("0.12");
      }
      fall_propagation(scalar) {
         values ("0.12");
      }
      rise_transition(tran_template) {
         values ("0.1, 0.15, 0.20, 0.29");
      }
      fall_transition (tran_template) {
         values ("0.1, 0.15, 0.20, 0.29");
      }
      related_pin : "A B" ;
   }
}
```

See "Specifying Delay Scaling Attributes" on page 11-23 for information about calculating delay factors. For information about including propagation delay in total cell delay calculations, see "Propagation Delay" on page 3-17.

Example 4-5 shows the use of lookup tables and templates for describing cell delay with the CMOS nonlinear delay model.

Example 4-5 Using Templates in the CMOS Nonlinear Delay Model

```
library( vendor_a ) {
   /* Use CMOS nonlinear delay model */

   delay_model : table_lookup;
  . . .

  /* Define template of size 4 x 4*/

   lu_table_template(basic_template) {
      variable_1 : input_net_transition;
      variable_2 : total_output_net_capacitance;
      index_1 ("0.0, 0.5, 1.5, 2.0");
      index_2 ("0.0, 2.0, 4.0, 6.0");
   }

   /* Define library-level one-dimensional lu_table of size 4 */

   lu_table_template(one_dimensional) {
      variable_1 : input_net_transition;
      index_1 ("0.0, 0.5, 1.5, 2.0");
   }
   . . .

   /* Define a cell with pins containing lu_table groups that */
```

```
   /* inherit the library-level template information */

   cell (general) {
      . . .
      pin(a) {
         direction: output;
         timing() {
         . . .

         /* Inherit the 'basic_template' template */

            cell_rise(basic_template) {

            /* Specify all the values */

               values ("0.0, 0.13, 0.17, 0.19", "0.21, 0.23, 0.30, \
                         0.41", "0.22, 0.31, 0.35, 0.47", "0.33, \
                     0.37, 0.45, 0.50");
               }
            }
       . . .
         }
      }
      pin(b) {
         direction: output;
         timing() {
          . . .
            /* Inherit the 'one-dimensional' template */

            cell_rise(one_dimensional) {

               /*Specify all the values within a pair of ""*/

               values ("0.1, 0.15, 0.20, 0.29");
            }
         }
      . . .
      }
      pin(c) {
         direction: output;
         timing() {
          . . .
            /* Use the predefined 'scalar' template */

            cell_rise(scalar) {

               /* Specify the value */
               values ("0.12");
            }
         }
      . . .
      }
   . . .
   }

  /* The rest of the library. */
   . . .
}
```

retaining_rise and retaining_fall Groups

The retaining delay is the time during which an output port retains its current logical value after a voltage rise or fall at a related input port.

The retaining delay is part of the arc delay (I/O path delay); therefore, its time cannot exceed the arc delay time. Because retaining delay is part of the arc delay, the retaining delay tables are placed within the timing arc.

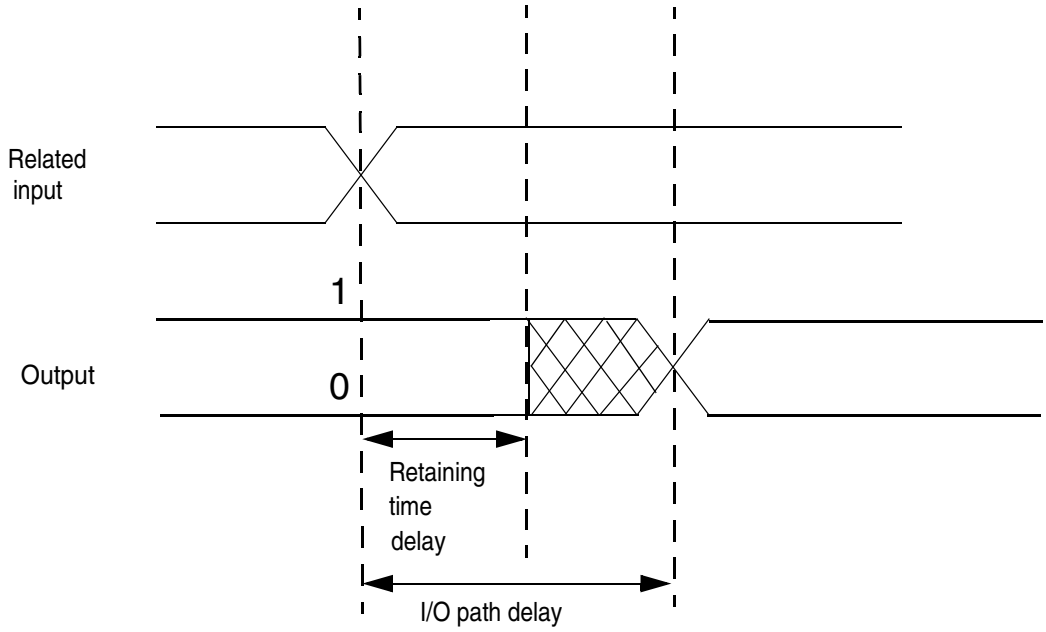
The value you enter for the `retaining_rise` attribute determines how long the output pin retains its current value, 0, after the value at the related input port has changed.

The value you enter for the `retaining_fall` attribute determines how long the output will retain its current value, 1, after the value at the related input port has changed.

Note:
Retaining time works only on a nonlinear delay model.

Figure 4-7 shows retaining delay in regard to changes in a related input port.

Figure 4-7 Retaining Time Delay



Example 4-6 shows how to use the `retaining_rise` and `retaining_fall` attributes.

Example 4-6 Retaining Time Delay

```
library(foo) {
   ...
   lu_table_template (retaining_table_template){
   ...
```

```
        variable_1: total_output_net_capacitance;
        variable_2: input_net_transition;
        index_1 ("0.0, 1.5");
        index_2 ("1.0, 2.1");
    }
    ...
    cell (cell_name){
    ...
        pin (A) {
            direction : output;
            ...
            timing(){
                related_pin : "B"
                ...
                retaining_rise (retaining_table_template){
                    values ("0.00, 0.23", "0.11, 0.28");
                }
                retaining_fall (retaining_table_template){
                    values ("0.01, 0.30", 0.12, 0.18");
                }
        }/*end of pin() */
        ...
    }/*end of cell() */
    ...
}/*end of library() */
```

See "Specifying Delay Scaling Attributes" on page 11-23 for information about calculating delay factors. For information about including propagation delay in total cell delay calculations, see "Propagation Delay" on page 3-17.

retain_rise_slew and retain_fall_slew Groups

These groups let you specify a slew table for the retain arc that is separate from the table of the parent delay arc. This retain arc represents the time it takes until an output pin starts losing its current logical value after a related input pin is changed. This decaying of the output logic value happens not only at a different time than the propagation of the final logical value but also at a different rate.

The retain delay is part of the arc delay (I/O path delay), and therefore its time cannot exceed the arc delay time. Because the retain delay is part of the arc delay, the retain delay tables are placed within the timing arc.

The value you enter for the `retain_rise_slew` attribute determines how long the output pin will retain its current value, 0, after the value at the related input port has changed.

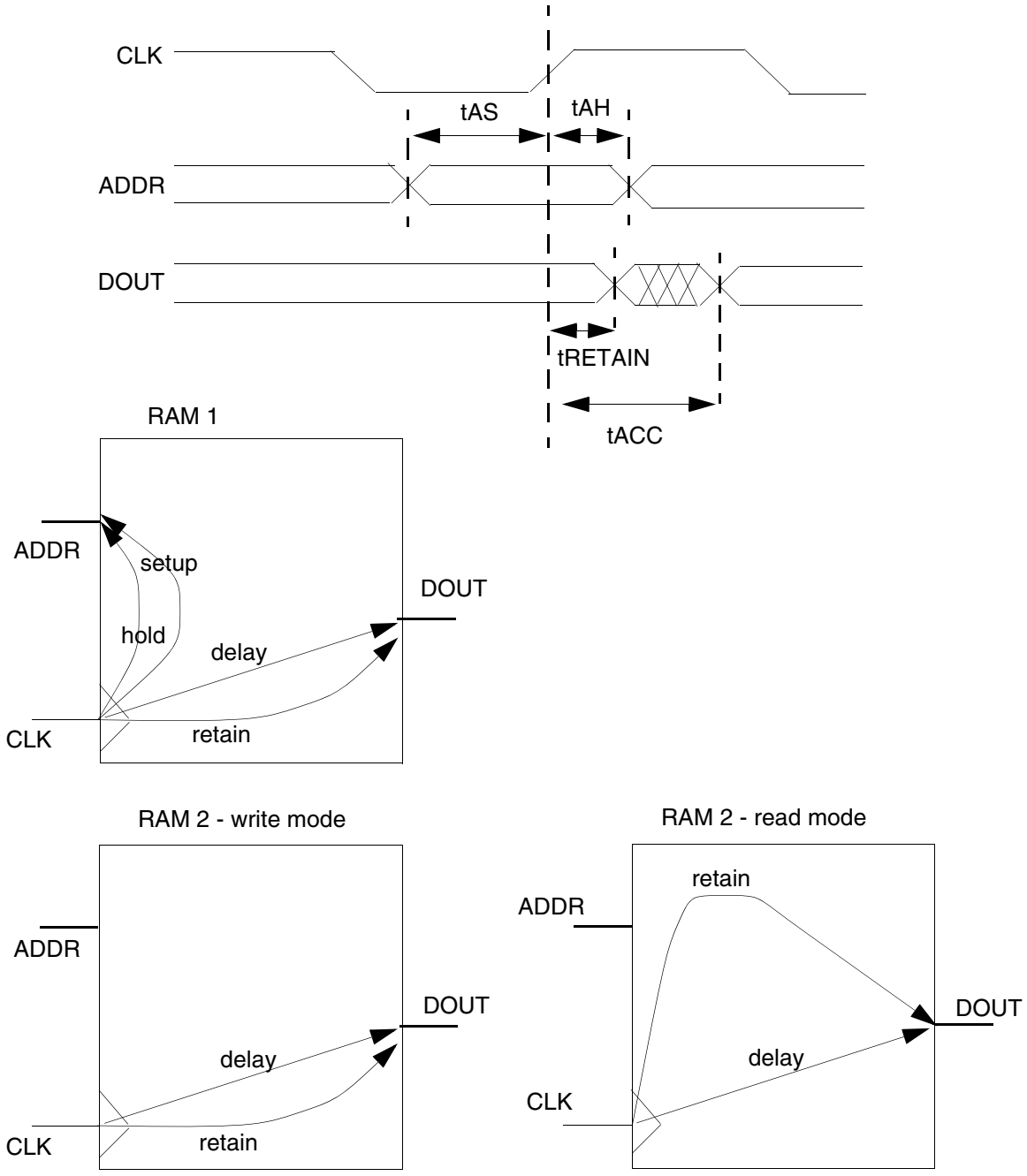
The value you enter for the `retain_fall_slew` attribute determines how long the output will retain its current value, 1, after the value at the related input port has changed.

Note:

Retaining time works only on a nonlinear delay model.

Figure 4-8 shows a timing diagram of synchronous RAM.

Figure 4-8 Timing Diagram of Synchronous RAM



Example

```
library(library_name) {
  ...
```

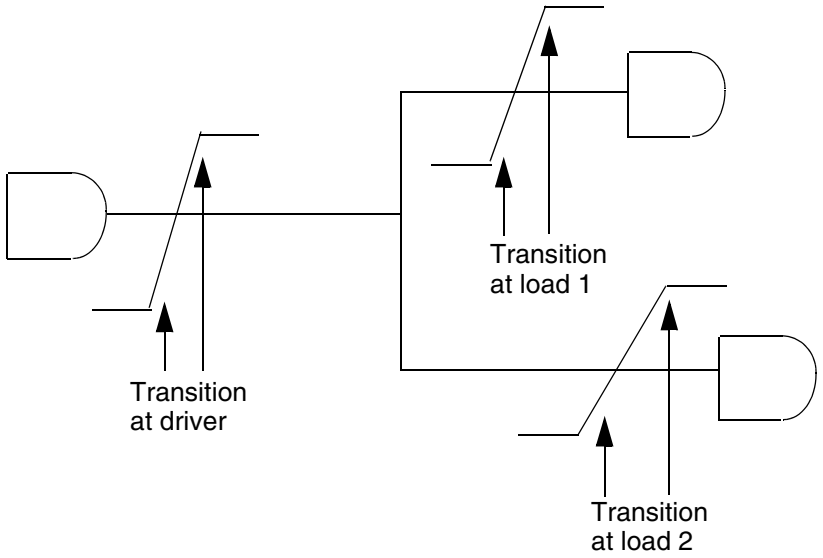
```
    lu_table_template (retaining_table_template){
        ...
        variable_1: total_output_net_capacitance;
        variable_2: input_net_transition;
        index_1 ("0.0, 1.5");
        index_2 ("1.0, 2.1");
    }
    ...
    cell (cell_name){
        ...
        pin (A) {
            direction : output;
            ...
            timing(){
                related_pin : "B"
                ...
                retaining_rise (retaining_table_template){
                    values ("0.00, 0.23", "0.11, 0.28");
                }
                retaining_fall (retaining_table_template){
                    values ("0.01, 0.30", 0.12, 0.18");
                }
                retain_rise_slew(retaining_time_template){
                    values("0.01,0.02");
                }
                retain_fall_slew(retaining_time_template){
                    values("0.01,0.02");
                }
        }/*end of pin() */
        ...
    }/*end of cell() */
        ...
}/*end of library() */
```

See “Specifying Delay Scaling Attributes” on page 11-23 for information about calculating delay factors. For information about including propagation delay in total cell delay calculations, see “Propagation Delay” on page 3-17.

Modeling Transition Time Degradation

Current nonlinear delay models are based on the assumption that the transition time at the load pin is the same as the transition time created at the driver pin. In reality, the net acts as a low-pass filter and the transition flattens out as it propagates from the driver of the net to each load, as shown in Figure 4-9. The higher the interconnect load, the greater the flattening effect and the greater the transition delay.

Figure 4-9 Transition Time Degradation



To model the degradation of the transition time as it propagates from an output pin over the net to the next input pin, include these library-level groups in your library:

- `rise_transition_degradation`
- `fall_transition_degradation`

These groups contain the values describing the degradation functions for rise and fall transitions in the form of lookup tables. The lookup tables are indexed by

- Transition time at the net driver
- Connect delay between the driver and a load

These are the supported values for transition degradation (`variable_1` and `variable_2`):

- output_pin_transition
- connect_delay

You can assign either `connect_delay` or `output_pin_transition` to `variable_1` or `variable_2`, as long as the index and table values are consistent with the assignment.

The values you use in the table compute the degradation transition according to the following formula:

```
degraded_transition =
table_lookup(f(output_pin_transition, connect_delay))
```

For more information about function `f`, see the “CMOS Nonlinear Delay Model Calculation” on page 3-18.

Design Compiler uses transition degradation tables when it indexes into any delay table in a library that uses the table parameters `input_net_transition`, `constrained_pin_transition`, or `related_pin_transition` in an `lu_table_template` group. Transition degradation tables in a library have no effect on computations related to cells from other libraries.

The k-factors for process, voltage, and temperature are not supplied for the new tables. The `output_pin_transition` value and the `connect_delay` value are computed at the current, rather than nominal, operating conditions.

Example 4-7 shows the use of the degradation tables. In this example, `trans_deg` is the name of the template for the transition degradation.

Example 4-7 Using Degradation Tables

```
library (simple_tlu) {
delay_model : table_lookup;

/* define the table templates */

lu_table_template(prop) {
   variable_1 : input_net_transition ;
   variable_2 : total_output_net_capacitance ;
   index_1("0, 1, 2");
   index_2("0, 1, 2");
}

lu_table_template(tran) {
   variable_1 : total_output_net_capacitance ;
   variable_2 : input_net_transition ;
   index_1("0, 1, 2");
   index_2("0, 1, 2");
}

lu_table_template(constraint) {
   variable_1 : constrained_pin_transition ;
   index_1("0, 1, 2");
   variable_2 : related_pin_transition ;
   index_2("0, 1, 2");
}

lu_table_template(trans_deg) {
   variable_1 : output_pin_transition ;
   index_1("0, 1");
   variable_2 : connect_delay ;
   index_2("0, 1");
}

/* the new degradation tables */

rise_transition_degradation(trans_deg) {
   values("0.0, 0.6", "1.0, 1.6");
```

```
}
fall_transition_degradation(trans_deg) {
   values("0.0, 0.8", "1.0, 1.8");
}

/* other library level defaults */

default_inout_pin_cap : 1.0;
...
k_process_fall_transition : 1.0;
...

nom_process : 1.0;
nom_temperature: 25.0;
nom_voltage : 5.0;

operating_conditions(BASIC_WORST) {
   process : 1.5 ;
   temperature : 70 ;
   voltage : 4.75 ;
   tree_type : "worst_case_tree" ;
}

/* list of cell descriptions */
cell(AN2) {
....
}
```

Any linear function of `output_pin_transition` and `connect_delay` can be represented with four table points, because the Design Compiler interpolation and extrapolation mechanism is linear. Larger tables are required to represent more-complex degradation functions, and breakpoints must be chosen so that the interpolation error is acceptable.

See "CMOS Nonlinear Delay Model" on page 3-14 for more information about calculating CMOS nonlinear timing models.

Modeling Load Dependency

"Describing Transition Delay" on page 4-48 describes how to model the transition time dependency of a constrained pin and its related pin on timing constraints. You can further model the effect of unbuffered output on timing constraints by modeling load dependency.

Load-dependent constraints are allowed in the CMOS nonlinear delay model and in the scalable polynomial delay model.

In the CMOS Nonlinear Delay Model

This is the procedure for modeling load dependency.

1. In the `timing` group of the output pin, set the `timing_type` attribute to one of the values shown in Table 4-4.
2. Use the `related_output_pin` attribute in the `timing` group to specify which output pin to use to calculate the load dependency.
3. Create a three-dimensional table template that uses two variables and indexes to model transition time and the third variable and index to model load. The variable values for representing output loading on the `related_output_pin` are

   ```
   related_out_total_output_net_capacitance
   related_out_output_net_length
   related_out_output_net_wire_cap
   related_out_output_net_pin_cap
   ```

   See "Defining the CMOS Nonlinear Delay Model Template" on page 4-18.
4. Create a three-dimensional lookup table, using the table template and the `index_3` attribute in the lookup table group. (See "Creating Lookup Tables" on page 4-21 .) The following groups are valid lookup tables for output load modeling:

   - `rise_constraint`
   - `fall_constraint`

   See "Setting Setup and Hold Constraints" on page 4-70 for information about these groups.

Example 4-8 is an example of a library that includes a load-dependent model.

Example 4-8 Load-Dependent Model in a Library

```
library(load_dependent) {
   delay_model : table_lookup;
   ...
   lu_table_template(constraint) {
      variable_1 :  constrained_pin_transition;
      variable_2 :  related_pin_transition;
      variable_3 :  related_out_total_output_net_capacitance;
      index_1 ("1, 5, 10");
      index_2 ("1, 5, 10");
      index_3 ("1, 5, 10");
   }
   cell(selector) {
      ...
      pin(d) {
         direction : input ;
         capacitance : 4 ;
         timing() {
```

```
            related_pin : "sel";
            related_output_pin : "so";
            timing_type : non_seq_hold_rising;
            rise_constraint(constraint) {
               values("1.5, 2.5, 3.5", "1.6, 2.6, 3.6", "1.7, 2.7, 3.7", \
                  "1.8, 2.8, 3.8", "1.9, 2.9, 3.9", "2.0, 3.0, 4.0", \
                  "2.1, 3.1, 4.1", "2.2, 3.2, 4.2", "2.3, 3.3, 4.3");
            }
            fall_constraint(constraint) {
               values("1.5, 2.5, 3.5", "1.6, 2.6, 3.6", "1.7, 2.7, 3.7", \
                  "1.8, 2.8, 3.8", "1.9, 2.9, 3.9", "2.0, 3.0, 4.0", \
                  "2.1, 3.1, 4.1", "2.2, 3.2, 4.2", "2.3, 3.3, 4.3");
            }
         }
         ...
      }
      ...
}
```

In the CMOS Scalable Polynomial Delay Model

This is the procedure for modeling load dependency.

1. In the `timing` group of the output pin, set the `timing_type` attribute to one of the values shown in Table 4-4.
2. Specify the output pin used to figure the load dependency with the `related_output_pin` attribute described below.
3. Create a three-dimensional table template that uses two variables to model transition time and a third variable, `poly_template`, to model load. The variable values for representing output loading on the `related_output_pin` are

   ```
   related_out_total_output_net_capacitance
   related_out_output_net_length
   related_out_output_net_wire_cap
   related_out_output_net_pin_cap
   ```

   See "Defining the Scalable Polynomial Delay Model Template" on page 4-22.
4. Create a three-dimensional lookup table, using the table template and the `index_3` attribute in the lookup table group.

   Express the delay equation in terms of scalable polynomial delay delay coefficients, using the `variable_3` variable and the `variable_3_range` attribute in the `poly_template` group.

   - `rise_constraint`
   - `fall_constraint`

See "Setting Setup and Hold Constraints" on page 4-70 for information about these groups.

Example 4-9 is an example of a library that includes a load-dependent model.

Example 4-9 Load-Dependent Model

```
library(load_dependent) {
    ...
    technology (cmos) ;
    delay_model : polynomial ;
    ...
    poly_template ( const ) {
        variables (constrained_pin_transition, related_pin_transition, \
                  related_out_total_output_net_capacitance);
           variable_1_range (0.0000, 4.0000);
           variable_2_range (0.0000, 4.0000);
           variable_3_range (0.0000, 4.0000);
    }
    ...
    cell(example) {
        ...
        pin(D) {
           direction : input ;
           capacitance : 1.00 ;
           timing() {
              timing_type : setup_rising ;
              fall_constraint(const) {
                 orders ("2, 1, 1")
                 coefs ("1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0");
              }
              rise_constraint(const){
                 orders ("2, 1, 1")
                 coefs ("1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0");
              }
              related_pin : "CP" ;
              related_output_pin : "Q";
           }
           timing() {
              timing_type : hold_rising ;
              rise_constraint(const) {
                 orders ("1, 1, 1")
                 coefs ("1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0");
              }
              fall_constraint(const){
                 orders ("1, 1, 1")
                 coefs ("1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0");
              }
              related_pin : "CP" ;
              related_output_pin : "Q";
           }
        }
        ...
  }  /* end cell */
    ...
}  /* end library */
```

Describing Slope Sensitivity

The slope delay of an element is the incremental time delay due to slowing changing input signals. Slope delay is calculated with the transition delay at the previous output pin with a slope sensitivity factor.

A slope sensitivity factor accounts for the time during which the input voltage begins to rise but has not reached the threshold level at which channel conduction begins. It is defined in the `timing` group of the driving pin.

The value in the attribute statements for slope sensitivity is a positive floating-point number that results in slope delay when multiplied by the transition delay.

In the CMOS Generic Delay Model and Piecewise Linear Delay Model

Use these slope sensitivity attributes for CMOS generic or piecewise linear technology only.

slope_rise Simple Attribute

This value represents the incremental delay to add to the slope of the input waveform for a logic 0-to-1 transition.

Example

```
slope_rise : 0.0;
```

slope_fall Simple Attribute

This value represents the incremental delay to add to the slope of the input waveform for a logic 1-to-0 transition.

Example

```
slope_fall: 0.0;
```

Describing State-Dependent Delays

These timing attributes describe the delay values for specified conditions.

In the `timing` group of a technology library, you can specify state-dependent delays that correspond to entries in Open Verilog International Standard Delay Format (OVI SDF 2.1) syntax. See the *Library Compiler Technology and Symbol Libraries Reference Manual* for syntax information about the attributes used to define timing check conditions and edge information.

To define a state-dependent timing arc, use these attributes:

- `when`
- `sdf_cond`

For state-dependent timing, each `timing` group requires both the `sdf_cond` and the `when` attributes.

You must define mutually exclusive conditions for state-dependent timing arcs. *Mutually exclusive* means that no more than one condition (defined in the `when` attribute) can be met at any time. Use the `default_timing` attribute to specify a default timing arc in the case of multiple timing arcs with `when` attributes.

See "Generating an SDF File" on page 4-97 for information about conditional path delays in an SDF file.

when Simple Attribute

The `when` attribute is a Boolean expression in the `timing` group that specifies the condition on which a timing arc depends to activate a path. Conditional timing lets you control the output pin of a cell with respect to the various *states* of the input pins.

See Table 4-5 for the valid Boolean operators.

Table 4-5 Valid Boolean Operators

Operator	Description
’	invert previous expression
!	invert following expression
^	logical XOR
*	logical AND

Table 4-5 Valid Boolean Operators (Continued)

Operator	Description
&	logical AND
space	logical AND
+	logical OR
|	logical OR
1	signal tied to logic 1
0	signal tied to logic 0

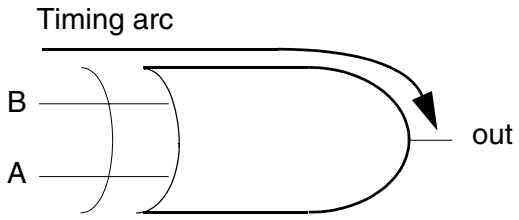
The order of precedence of the operators is left to right, with inversion performed first, then XOR, then AND, then OR.

Example

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

Figure 4-10 shows an XOR gate.

Figure 4-10 XOR Gate With State-Dependent Timing Arc



Example 4-10 shows how to use the `when` attribute for an XOR gate. In the description of the XOR cell, pin A sets conditional timing for the output pin “out” when you define the timing arc for `related_pin` B. In this example, when you set conditional timing for `related_pin` A with the `when` : “B” statement, the output pin gets the negative_unate value of A when the condition is B.

There are limitations on the pin types that can be used with different types of cells with the `when` attribute.

For a combinational cell, these pins are valid with the `when` attribute:

- Pins in the `function` attribute for regular combinational timing arcs
- Pins in the `three_state` attribute for the endpoint of the timing arc

For a sequential cell, valid pins or variables to use with the `when` attribute are determined by the timing type of the arc.

- For timing types `rising_edge` and `falling_edge`: Pins in these attributes are allowed in the `when` attribute:
  - `next_state`
  - `clocked_on`
  - `clocked_on_also`
  - `enable`
  - `data_in`
- For timing type `clear`
  - If the pin's function is the first state variable in the `flip-flop` or `latch` group, the pin that defines the clear condition in the `flip-flop` or `latch` group is allowed in the `when` construct.
  - If the pin's function is the second state variable in the `flip-flop` or `latch` group, the pin that defines the preset condition in the `flip-flop` or `latch` group is allowed in the `when` construct.
- For timing type `preset`
  - If the pin's function is the first state variable in the `flip-flop` or `latch` group, the pin that defines the preset condition in the `flip-flop` or `latch` group is allowed in the `when` construct.
  - If the pin's function is the second state variable in the `flip-flop` or `latch` group, the pin that defines the clear condition in the `flip-flop` or `latch` group is allowed in the `when` construct.

See "timing_type Simple Attribute" on page 4-28 for more information.

All input pins in a black box cell (a cell without a `function` attribute) are allowed in the `when` attribute.

sdf_cond Simple Attribute

Defined in the state-dependent timing group, the `sdf_cond` attribute supports Standard Delay Format (SDF) file generation and condition matching during back-annotation.

Example

```
sdf_cond : "SE ==1'B1";
```

The `sdf_cond` attribute should be logically equivalent to the `when` attribute in the same timing arc. If these two Boolean expressions are not equivalent, back-annotation is not performed properly.

Note:

Library Compiler does not parse the `sdf_cond` expression, nor does it check for logical equivalency between these two expressions.

Example 4-10 is a 2-input XOR gate. It represents a commonly used state-dependent delay case. The intrinsic delay between pin A and pin OUT is 1.3 for rising and 1.5 for falling when pin B = 1. There is an additional timing arc between the same two pins that has `intrinsic_rise` 1.4 and `intrinsic_fall` 1.6 `when` pin B = 0.

Example 4-10 XOR Cell With State-Dependent Timing

```
cell(XOR) {
   pin(A) {
      direction : input;
   ...
   }
   pin(B) {
      direction : input;
   ...
   }
   pin(out) {
      direction : output;
      function : "A ^ B";
      timing() {
         related_pin : "A";
         timing_sense : negative_unate;
         when : "B";
         sdf_cond : " B == 1'B1 ";
         intrinsic_rise : 1.3;
         intrinsic_fall : 1.5;
      }
      timing() {
         related_pin : "A";
         timing_sense : positive_unate;
         when : "!B";
         sdf_cond : " B == 1'B0 ";
         intrinsic_rise : 1.4;
         intrinsic_fall : 1.6;
      }
      timing() {/* default timing arc */
         related_pin : "A";
         timing_sense : non_unate;
         intrinsic_rise : 1.4;
         intrinsic_fall : 1.6;
      }
      timing() {
         related_pin : "B";
         timing_sense : negative_unate;
```

```
            when : "A";
            sdf_cond : "A == 1'B1 ";
            intrinsic_rise : 1.3;
            intrinsic_fall : 1.5;
         }
         timing() {
            related_pin : "B";
            timing_sense : positive_unate;
            when : "!A";
            sdf_cond : "A == 1'B0 ";
            intrinsic_rise : 1.4;
            intrinsic_fall : 1.6;
         }
         timing() {/* default timing arc */
            related_pin : "B";
            timing_sense : non_unate;
            intrinsic_rise : 1.4;
            intrinsic_fall : 1.6;
         }
      }
   }
```

Setting Setup and Hold Constraints

Signals arriving at an input pin have ramp times. Therefore, you must ensure that the data signal has stabilized before latching its value by defining setup and hold arcs as timing requirements. No path tracing takes place along these timing arcs; they are used solely for constraint verification.

- Setup constraints describe the minimum time allowed between the arrival of the data and the transition of the clock signal. During this time, the data signal must remain constant. If the data signal makes a transition during the setup time, an incorrect value may be latched.
- Hold constraints describe the minimum time allowed between the transition of the clock signal and the latching of the data. During this time, the data signal must remain constant. If the data signal makes a transition during the hold time, an incorrect value may be latched.

By combining a setup time and a hold time, you can ensure the stability of data that is latched.

Figure 4-11 shows setup and hold timing for a rising-edge-triggered flip-flop. The timing checks for flip-flops use the activating edge of the clock, which is the rising edge in this case.

Figure 4-11 Setup and Hold Constraints for Rising-Edge-Triggered Flip-Flop

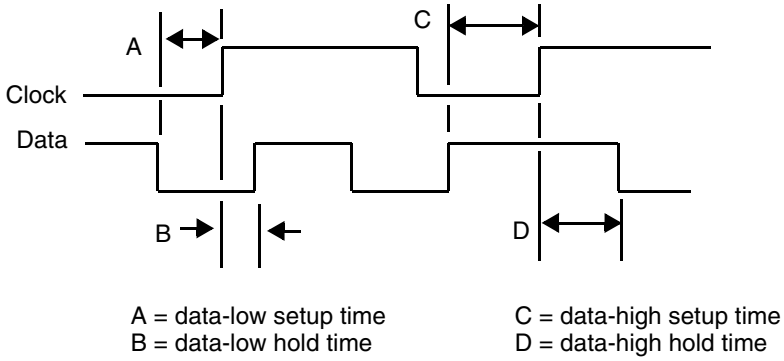
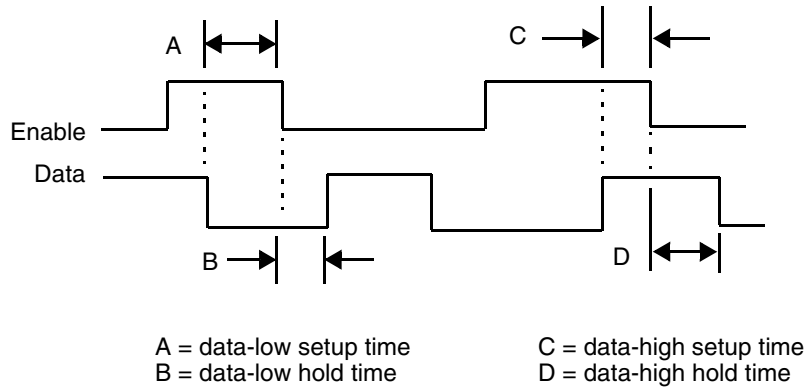


Figure 4-12 illustrates setup and hold timing for a high-enable latch. The timing checks for latches generally use the deactivating edge of the enable signal, which is the falling edge in this case. However, the method used depends on the vendor.

Figure 4-12 Setup and Hold Constraints for High-Enable Latch



In the CMOS Generic Delay Model and Piecewise Linear Delay Model

Use setup and hold constraints only between data pins and clock pins.

Setup Constraints

The values you can assign to a `timing_type` attribute to define timing arcs used for setup checks on clocked elements are

setup_rising

Designates the rising edge of the related pin for the setup check.

setup_falling

Designates the falling edge of the related pin for the setup check.

To define a setup constraint,

1. Assign one of the constraint values to the `timing_type` attribute.
2. Specify a related pin in the `timing` group.

   The `related_pin` attribute in a timing arc with a `setup_rising` or `setup_falling` timing type identifies the pin used for the timing check. The rising or falling designation tells Design Compiler which edge of the related pin is active for the setup check.

Example

This example describes the setup arc for the data pin on the rising-edge-triggered D flip-flop shown in Figure 4-11. The `intrinsic_rise` value represents setup time C, and the `intrinsic_fall` value represents setup time A. The syntax shown assumes a generic or piecewise linear delay model.

```
timing() {
   timing_type : setup_rising ;
   intrinsic_rise : 1.5 ;
   intrinsic_fall : 1.5 ;
   related_pin : "Clock" ;
}
```

The following example describes the setup constraint for the data pin on the high-enable latch shown in Figure 4-12. The `intrinsic_rise` value represents setup time C, and the `intrinsic_fall` value represents setup time A.

```
timing() {
      timing_type : setup_falling ;
      intrinsic_rise : 1.5 ;
      intrinsic_fall : 1.5 ;
      related_pin : "Enable" ;
```

```
}
```

Note:

The VHDL library generator accepts only one setup or hold value. For example, if you use both the `intrinsic_rise` and the `intrinsic_fall` attributes, the library generator uses the maximum of the two values.

Hold Constraints

The values you can assign to the `timing_type` attribute to define timing constraints used for hold checks on clocked elements are

hold_rising

This value designates the rising edge of the related pin for the hold check.

hold_falling

This value designates the falling edge of the related pin for the hold check.

To define a hold constraint,

1. Assign one of the constraint values to the `timing_type` attribute.
2. Specify a related pin in the `timing` group.

   The `related_pin` attribute in a timing arc with a `hold_rising` or `hold_falling` timing type identifies the pin used for the timing check. The rising or falling designation tells Design Compiler which edge of the related pin is active for the hold check.

Example

This example shows the hold constraint for pin D on a rising-edge-triggered D flip-flop. The `intrinsic_rise` value represents hold time B in Figure 4-11, and the `intrinsic_fall` value is hold time D.

```
timing() {
      timing_type : hold_rising;
      intrinsic_rise : 0.5 ;
      intrinsic_fall : 0.5 ;
      related_pin : "Clock" ;
}
```

The following example shows the hold constraint for the data pin on a high-enable latch. The `intrinsic_rise` value represents hold time B in Figure 4-12, and the `intrinsic_fall` value represents hold time D.

```
timing() {
   timing_type : hold_falling ;
   intrinsic_rise : 1.5 ;
   intrinsic_fall : 1.5 ;
```

```
   related_pin : "Enable" ;
}
```

Setup and Hold Timing Constraints

You can describe a complete setup and hold timing constraint by combining a setup constraint and a hold constraint. The following example shows setup and hold timing constraints on pin K in a JK flip-flop.

```
pin(K) {
   direction : input ;
   capacitance : 1.3 ;
   timing() {
      timing_type : setup_rising ;
      intrinsic_rise : 1.5 ;
      intrinsic_fall : 1.5 ;
      related_pin : "CP" ;
   }
   timing() {
      timing_type : hold_rising ;
      intrinsic_rise : 0.0 ;
      intrinsic_fall : 0.0 ;
      related_pin : "CP" ;
   }
}
```

In the CMOS Nonlinear Delay Model

The CMOS nonlinear timing model can support timing constraints that are sensitive to clock or data-input transition times. Each constraint is defined by a `timing` group with two lookup tables:

- `rise_constraint` group
- `fall_constraint` group

rise_constraint and fall_constraint Groups

These constraint tables replace the `intrinsic_rise` and `intrinsic_fall` attributes used in the other delay models. The format of the lookup table template and the format of the lookup table are the same as described previously in “Defining the CMOS Nonlinear Delay Model Template” on page 4-18 and “Creating Lookup Tables” on page 4-21.

These are valid variable values for the timing constraint template:

`constrained_pin_transition`

Value for the transition time of the pin that owns the `timing` group.

`related_pin_transition`

Value for the transition time of the `related_pin` defined in the `timing` group.

For each `timing` group containing one of the following `timing_type` attribute values, at least one lookup table is required:

- `setup_rising`
- `setup_falling`
- `hold_rising`
- `hold_falling`
- `skew_rising`
- `skew_falling`
- `non_seq_setup_rising`
- `non_seq_setup_falling`
- `non_seq_hold_rising`
- `non_seq_hold_falling`
- `nochange_high_high`
- `nochange_high_low`
- `nochange_low_high`
- `nochange_low_low`

For each `timing` group with one of the following `timing_type` attribute values, only one lookup table is required:

- `recovery_rising`
- `recovery_falling`
- `removal_rising`
- `removal_falling`

Example 4-11 shows how to use tables to specify setup constraints for a flip-flop.

Example 4-11 CMOS Nonlinear Timing Model Using Constraint

```
library( vendor_b ) {

   /* 1. Use delay lookup table */
   delay_model : table_lookup;

   /* 2. Define template of size 3 x 3*/
```

```
    lu_table_template(constraint_template) {
        variable_1 : constrained_pin_transition;
        variable_2 : related_pin_transition;
        index_1 ("0.0, 0.5, 1.5");
        index_2 ("0.0, 2.0, 4.0");
    }
    . . .
    cell(dff) {
        pin(d) {
            direction: input;
            timing() {
                related_pin : "clk";
                timing_type : setup_rising;

                /* Inherit the constraint_template template */
                rise_constraint(constraint_template) {

                    /* Specify all the values */
                    values ("0.0, 0.13, 0.19",  \
                    "0.21, 0.23, 0.41", \
                    "0.33, 0.37, 0.50");
                }
                fall_constraint(constraint_template) {
                    values ("0.0, 0.14, 0.20",  \
                    "0.22, 0.24, 0.42", \
                    "0.34, 0.38, 0.51");
                }
            }
            . . .
        }
    }
}
```

In the Scalable Polynomial Delay Model

Example 4-12 shows how to specify constraint in a scalable polynomial delay model.

Example 4-12 CMOS Scalable Polynomial Delay Model Using Constraint

```
library(vendor_b) {
    /* Use polynomial delay model */
    delay_model : polynomial;
    /* Define template */
    poly_template ( constraint_template_poly ) {
        variables (constrained_pin_transition, related_pin_transition);
            variable_1_range (0.01, 3.00);
            variable_2_range (0.01, 3.00);
        }
    .......
    cell(dff) {
        pin(D) {
            direction : input ;
            timing() {
                related_pin : "CP" ;
                timing_type : setup_rising ;
                rise_constraint ( constraint_template_poly ) {
                    orders("1, 1");
```

```
                /* (a0 + a1x1 )(b0 + b1x2) = A00 + A10x1 + A01x2 + A11x1x2  */
                coefs ("0.2487 +0.0520 -0.0268 -0.0053 " ) ;
                variable_1_range (0.01, 3.00);
                variable_2_range (0.01, 3.00);
             }
             fall_constraint ( constraint_template_poly ) {
                orders("1, 2");
                /* (a0 + a1x1 )(b0 + b1x2 + b2x2^2 )  = */
                /* A00 + A10x1 + A01x2 + A11x1x2 + A02x2^2 + A12x1x2^2 */
                coefs ("0.2732 +0.0668 +0.0216 -0.0002 -0.0003 +0.0000 " ) ;
                variable_1_range (0.01, 3.00);
                variable_2_range (0.01, 3.00);
             }
          }
        .......
       }
   }
 .....
}
```

Identifying Interdependent Setup and Hold Constraints

To reduce slack violation, use pairs of `interdependence_id` attributes to identify interdependent pairs of setup and hold constraint tables. Interdependence data is supported in conditional constraint checking. The `interdependence id` increases independently for each condition. Interdependence data can be specified in pin or bus and bundle groups. For details, see the Library Compiler Reference Manual: Technology and Symbol Libraries.

Setting Nonsequential Timing Constraints

You can set constraints requiring that the data signal on an input pin remain stable for a specified amount of time before or after another pin in the same cell changes state. These cells are termed nonsequential cells, because the related pin is not a clock signal.

All Synopsys delay models supporting sequential setup and hold constraint modeling also support nonsequential setup and hold modeling.

Scaling of nonsequential setup and hold constraints based on the environment use k-factors for sequential setup and hold constraints.

The values you can assign to a `timing_type` attribute to model nonsequential setup and hold constraints are

`non_seq_setup_rising`

Designates the rising edge of the related pin for the setup check.

`non_seq_setup_falling`

Designates the falling edge of the related pin for the setup check.

`non_seq_hold_rising`

Designates the rising edge of the related pin for the hold check.

`non_seq_hold_falling`

Designates the falling edge of the related pin for the hold check.

To model nonsequential setup and hold constraints for a cell,

1. Assign a value to the `timing_type` attribute in a `timing` group of an input or I/O pin.
2. Specify a related pin with the `related_pin` attribute in the `timing` group. The related pin in a timing arc is the pin used for the timing check.

   Use any pin in the same cell, except for output pins, and the constrained pin itself as the related pin.

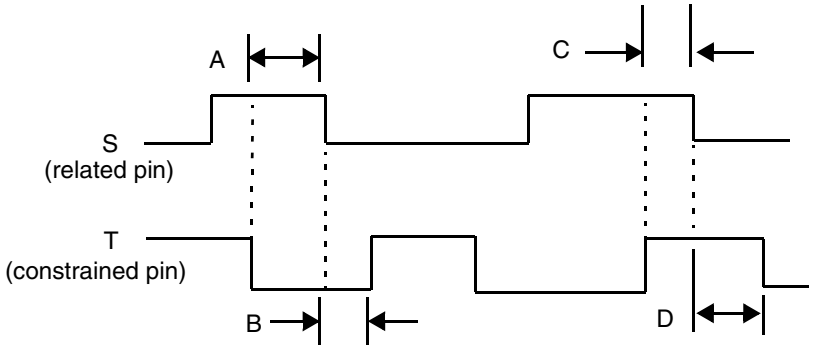
   You can use both rising and falling edges as the active edge of the related pin for one cell.

Example

```
pin(T) {
   timing() {
      timing_type : non_seq_setup_falling;
      intrinsic_rise : 1.5;
      intrinsic_fall : 1.5;
      related_pin : "S";
   }
}
```

Figure 4-13 shows the waveforms for the nonsequential timing arc described in the preceding example. In this timing arc, the constrained pin is T and its related pin is S. The intrinsic rise value describes setup time C or hold time B. The intrinsic fall value describes setup time A or hold time D.

Figure 4-13 Nonsequential Setup and Hold Constraints



A = `intrinsic_fall` in setup time
B = `intrinsic_rise` in hold time
C = `intrinsic_rise` in setup time
D = `intrinsic_fall` in hold time

Setting Recovery and Removal Timing Constraints

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

Recovery Constraints

The recovery timing arc describes the minimum allowable time between the control pin transition to the inactive state and the active edge of the synchronous clock signal (time between the control signal going inactive and the clock edge that latches data in).

The asynchronous control signal must remain constant during this time, or else an incorrect value may appear at the outputs.

Figure 4-14 shows the recovery timing arc for a rising-edge-triggered flip-flop with active-low clear.

Figure 4-15 shows the recovery timing arc for a low-enable latch with active-high preset.

Figure 4-14 Recovery Timing Constraint for a Rising-Edge-Triggered Flip-Flop

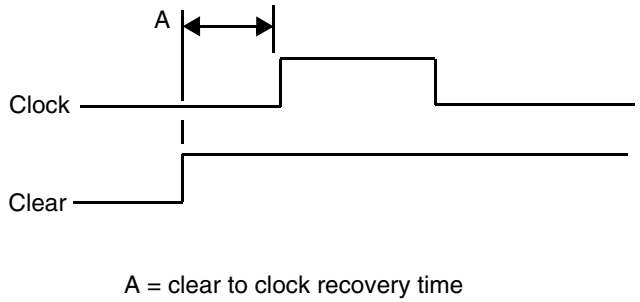
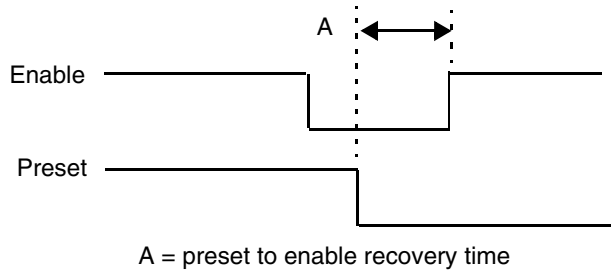


Figure 4-15 Recovery Timing Constraint for a Low-Enable Latch



The values you can assign to a `timing_type` attribute to define a recovery time constraint are

`recovery_rising`

Uses the rising edge of the related pin for the recovery time check; the clock is rising-edge-triggered.

`recovery_falling`

Uses the falling edge of the related pin for the recovery time check; the clock is falling-edge-triggered.

To define a recovery time constraint for an asynchronous control pin,

1. Assign a value to the `timing_type` attribute.

   Use `recovery_rising` for rising-edge-triggered flip-flops and low-enable latches; use `recovery_falling` for negative- edge-triggered flip-flops and high-enable latches.

2. Identify the synchronous clock pin as the `related_pin`.

   For active-low control signals, define the recovery time with the `intrinsic_rise` statement.

For active-high control signals, define the recovery time with the `intrinsic_fall` statement.

Example

This example shows a recovery timing arc for the active-low clear signal in a rising-edge-triggered flip-flop. The `intrinsic_rise` value represents clock recovery time A in Figure 4-14.

```
pin (Clear) {
   direction : input ;
   capacitance : 1 ;
   timing() {
      related_pin : "Clock" ;
      timing_type : recovery_rising;
      intrinsic_rise : 1.0 ;
   }
}
```

The following example shows a recovery timing arc for the active-high preset signal in a low-enable latch. The `intrinsic_fall` value represents clock recovery time A in Figure 4-15.

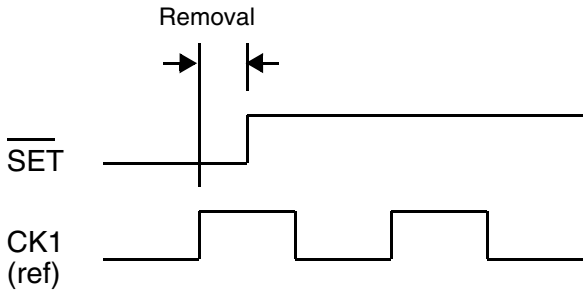
```
pin (Preset) {
   direction : input ;
      capacitance : 1 ;
      timing() {
      related_pin : "Enable" ;
      timing_type : recovery_rising;
      intrinsic_fall : 1.0 ;
   }
}
```

Removal Constraint

This constraint is also known as the asynchronous control signal hold time.

The removal constraint describes the minimum allowable time between the active edge of the clock pin while the asynchronous pin is active and the inactive edge of the same asynchronous control pin (see Figure 4-16).

Figure 4-16 Timing Diagram for Removal Constraint



The values you can assign to a `timing_type` attribute to define a removal constraint are

`removal_rising`

Use when the cell is a low-enable latch or a rising-edge-triggered flip-flop.

`removal_falling`

Use when the cell is a high-enable latch or a falling-edge-triggered flip-flop.

To define a removal constraint,

1. Assign a value to the `timing_type` attribute.
2. Identify the synchronous clock pin as the `related_pin`.
3. For active-low asynchronous control signals, define the removal time with the `intrinsic_rise` attribute.

   For active-high asynchronous control signals, define the removal time with the `intrinsic_fall` attribute.

Example

```
pin ( SET ) {
   ....
   timing() {
      timing_type : removal_rising;
      related_pin : " CK1 ";
      intrinsic_rise : 1.0 ;
      }
}
```

Setting No-Change Timing Constraints

You can model no-change timing checks to use in static timing verification during synthesis.

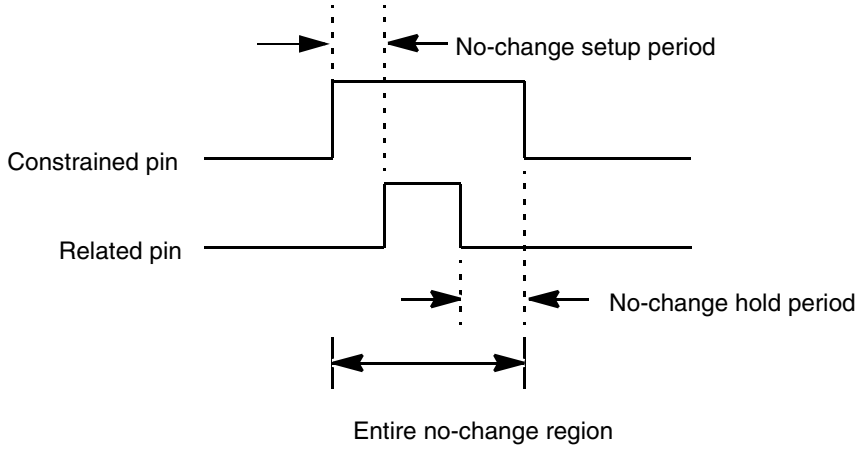
A no-change timing check checks a constrained signal against a level-sensitive related signal. The constrained signal must remain stable during an established setup period, for the width of the related pulse, and during an established hold period.

For example, you can use the no-change timing check to model the timing requirements of latch devices with latch enable signals. To ensure correct latch sampling, the latch enable signal must remain stable during the clock pulse and the setup and hold time around the clock pulse.

You can also use the no-change timing check to model the timing requirements of memory devices. To guarantee correct read/write operations, the address or data must remain stable during a read/write enable pulse and the setup and hold margins around the pulse.

Figure 4-17 shows a no-change timing check between a constrained pin and its level-sensitive related pin.

Figure 4-17 No-Change Timing Check



The values you can assign to a `timing_type` attribute to define a no-change timing constraint are

`nochange_high_high`

Specifies a positive pulse on the constrained pin and a positive pulse on the related pin.

`nochange_high_low`

Specifies a positive pulse on the constrained pin and a negative pulse on the related pin.

`nochange_low_high`

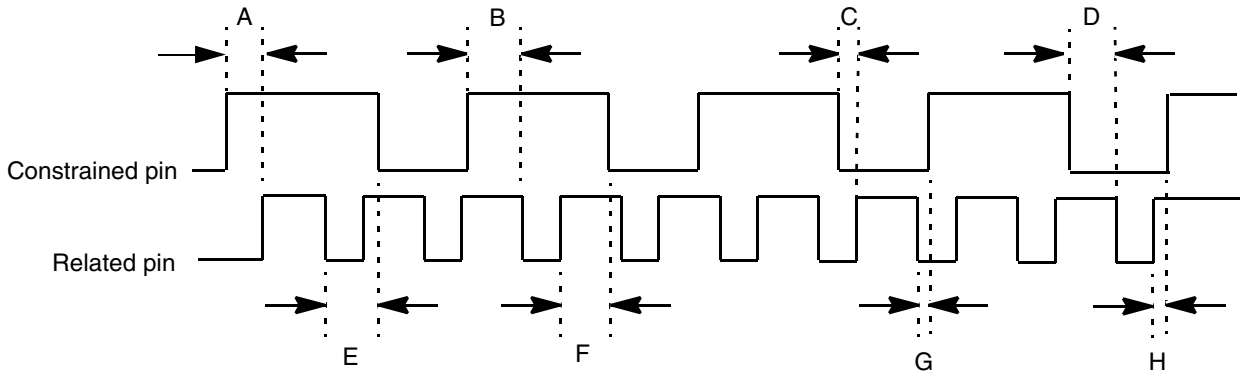
Specifies a negative pulse on the constrained pin and a positive pulse on the related pin.

`nochange_low_low`

Specifies a negative pulse on the constrained pin and a negative pulse on the related pin.

Figure 4-18 shows the waveforms for these constraints.

Figure 4-18 No-Change Setup and Hold Constraint Waveforms



	Setup	Hold
`nochange_high_high`	A	E
`nochange_high_low`	B	F
`nochange_low_high`	C	G
`nochange_low_low`	D	H

To model no-change timing constraints,

1. Assign a value to the `timing_type` attribute.
2. Specify a related pin with the `related_pin` attribute in the `timing` group.

   The final high or low designation in the `timing_type` attribute tells Design Compiler which pulse of the related pin is active for the setup or hold constraint.The related pin in the timing arc is the pin used for the timing check.
3. Specify delay attribute values according to the delay model you use, as summarized in Table .

Note:

With no-change timing constraints, conditional timing constraints have different interpretations than they do with other constraints. See "Setting Conditional Timing Constraints" on page 4-90 for more information.

No-change constraint	Setup attribute for generic delay and nonlinear delay models	Hold attribute for generic delay and nonlinear delay models
nochange_high_high	`intrinsic_rise` / `rise_constraint`	`intrinsic_fall`/ `fall_constraint`
nochange_high_low	`intrinsic_rise` / `rise_constraint`	`intrinsic_fall`/ `fall_constraint`
nochange_low_high	`intrinsic_fall` / `fall_constraint`	`intrinsic_rise` / `rise_constraint`
nochange_low_low	`intrinsic_fall` / `fall_constraint`	`intrinsic_rise` / `rise_constraint`

In the CMOS Generic Delay Model

In the CMOS generic delay model, specify setup time with `intrinsic_rise` and hold time with `intrinsic_fall`.

This is the syntax for the no-change timing check in the CMOS generic delay model:

```
timing () {
   timing_type : nochange_high_high | nochange_high_low |
                 nochange_low_high | nochange_low_low;
   related_pin : related_pinname;
   intrinsic_rise : float; /* constrained signal rising */
   intrinsic_fall : float; /* constrained signal falling */
}
```

In the CMOS Nonlinear Delay Model

In the CMOS nonlinear delay model, specify setup time with `rise_constraint` and hold time with `fall_constraint`.

This is the syntax for the no-change timing check in the CMOS nonlinear delay model:

```
timing () {timing_type : nochange_high_high | nochange_high_low |
nochange_low_high | nochange_low_low ;related_pin : related_pinname;
   rise_constraint (template_name_id) { /* constrained signal rising */
values (float, ..., float) ;}fall_constraint (template_name_id) { /*
constrained signal falling */values (float, ..., float) ;}}
```

Example

This is an example of a no-change timing check in a technology library using the CMOS nonlinear delay model:

```
library (the_lib) {
   delay_model : polynomial;
   k_process_nochange_rise : 1.0;  /* no-change scaling factors */
   k_process_nochange_fall : 1.0;
   k_volt_nochange_rise : 0.0;
   k_volt_nochange_fall : 0.0;
   k_temp_nochange_rise : 0.0;
   k_temp_nochange_fall : 0.0;

   cell (the_cell) {
      pin(EN {
         timing () {
            timing_type : nochange_high_low;
            related_pin : CLK;
            rise_constraint (polynomial) { /* setup time */
               values (2.98);
            }
            fall_constraint (polynomial) { /* hold time */
               values (0.98);
            }
         }
       ...
      }
     ...
   }
  ...
}
```

In the CMOS Scalable Polynomial Delay Model

In the CMOS scalable polynomial delay model, specify setup time with `rise_constraint` and hold time with `fall_constraint`.

This is the syntax for the no-change timing check in the CMOS scalable polynomial delay model:

```
timing () {timing_type : nochange_high_high | nochange_high_low |
nochange_low_high | nochange_low_low;related_pin : related_pinname;
   rise_constraint (poly_template_name_id) { /* constrained signal rising
*/    orders(integer, integer) ;coefs(float, ..., float) ;
variable_n_range(float, float) ;} fall_constraint (poly_template_name_id)
{     /* constrained signal falling */ orders(integer, integer) ;
coefs(float, ..., float) ;variable_n_range(float, float) ;} }
```

Example

This is an example of a technology library no-change timing check using the CMOS scalable polynomial delay model:

```
library (the_lib) {
   delay_model : table_lookup;
   k_process_nochange_rise : 1.0; /* no-change scaling factors */
   k_process_nochange_fall : 1.0;
   k_volt_nochange_rise : 0.0;
   k_volt_nochange_fall : 0.0;
   k_temp_nochange_rise : 0.0;
   k_temp_nochange_fall : 0.0;
   ...
   cell (the_cell) {
      pin(EN {
         timing () {
            timing_type : nochange_high_low;
            related_pin : CLK;
            rise_constraint (poly_template) { /* setup time */
               orders ("1, 1");
               coefs ("0.2487 +0.0520 -0.0268 -0.0053 " );
               variable_1_range (0.01, 3.00);
               variable_2_range (0.01, 3.00);
            }
            fall_constraint (poly_template) { /* hold time */
               orders ("1, 1");
               coefs ("0.2487 +0.0520 -0.0268 -0.0053 " );
               variable_1_range (0.01, 3.00);
               variable_2_range (0.01, 3.00);
            }
         }
      ...
       }
    ...
   }
...
}
```

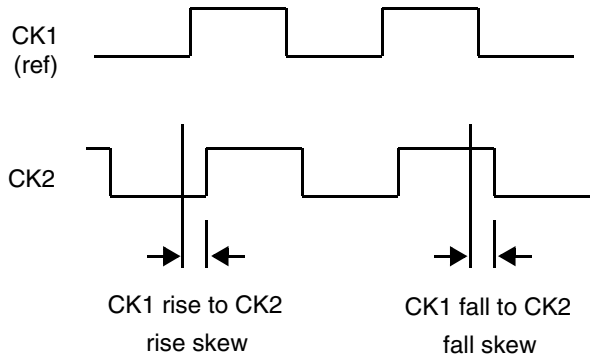
Setting Skew Constraints

The skew constraint defines the maximum separation time allowed between two clock signals.

The VHDL library generator uses the skew constraint for simulation.

Figure 4-19 is a timing diagram showing skew constraint.

Figure 4-19 Timing Diagram for Skew Constraint



The values you can assign to a `timing_type` attribute to define a skew constraint are

skew_rising

The timing constraint interval is measured from the rising edge of the reference pin (specified in `related_pin`) to a transition edge of the parent pin of the `timing` group. The `intrinsic_rise` value is the maximum skew time between the reference pin rising and the parent pin rising. The `intrinsic_fall` value is the maximum skew time between the reference pin falling and the parent pin falling.

skew_falling

The timing constraint interval is measured from the falling edge of the reference pin (specified in `related_pin`) to a transition edge of the parent pin of the `timing` group. The `intrinsic_rise` value is the maximum skew time between the reference pin rising and the parent pin rising. The `intrinsic_fall` value is the maximum skew time between the reference pin falling and the parent pin falling.

To set skew constraint,

1. Assign a value to the `timing_type` attribute.
2. Define only one of these attributes in the `timing` group:
   - `intrinsic_rise`

- intrinsic_fall

3. Use the related_pin attribute in the timing group to specify a reference clock pin. Only the following attributes in a skew timing group are used (all others are ignored):

   - timing_type
   - related_pin
   - intrinsic_rise
   - intrinsic_fall

Example

This example shows how to model constraint CK1 rise to CK2 rise skew:

```
pin (CK2) {
   ....
   timing() {
      timing_type : skew_rising;
      related_pin : " CK1 ";
      intrinsic_rise : 1.0 ;
   }
}
```

Setting Conditional Timing Constraints

A conditional timing constraint describes a check Design Compiler should perform when a specified condition is met. You can specify conditional timing checks in `pin`, `bus`, and `bundle` groups.

Use the following attributes and groups to specify conditional timing checks.

Attributes:

- `when`
- `sdf_cond`
- `when_start`
- `sdf_cond_start`
- `when_end`
- `sdf_cond_end`
- `sdf_edges`

Groups:

- `min_pulse_width`
- `minimum_period`

when and sdf_cond Simple Attributes

The `when` attribute defines enabling conditions for timing checks such as setup, hold, and recovery. Use `when` to model timing check conditions for VITAL models.

If you define `when`, you must define `sdf_cond`.

Using the `when` and `sdf_cond` pair is a short way of specifying `when_start`, `sdf_cond_start`, `when_end`, and `sdf_cond_end` when the start condition is identical to the end condition. Library Compiler applies `when` and `sdf_cond` attribute values to both start and end conditions. Library Compiler ignores the values for `when` and `sdf_cond` if the same `timing` group contains definitions for `when_start`, `sdf_cond_start`, `when_end`, and `sdf_cond_end`.

“Describing State-Dependent Delays” on page 4-66 describes the `sdf_cond` and `when` attributes in defining state-dependent timing arcs.

when_start Simple Attribute

In a `timing` group, `when_start` defines a timing check condition specific to a start event in VHDL initiative toward ASIC libraries (VITAL) models in Synopsys logic format. The expression in this attribute contains any pin, including input, output, I/O, and internal pins. You must use real pin names. Bus and bundle names are not allowed.

Example

```
when_start : "SIG_A"; /*SIG_A must be a declared pin */
```

The `when_start` attribute requires an `sdf_cond_start` attribute in the same `timing` group.

The end condition is considered always true if a `timing` group contains `when_start` but no `when_end`.

sdf_cond_start Simple Attribute

In a `timing` group, `sdf_cond_start` defines a timing check condition specific to a start event in VITAL models in OVI SDF 2.1 syntax.

Example

```
sdf_cond_start : "SIG_A";
```

The `sdf_cond_start` attribute requires a `when_start` attribute in the same `timing` group.

The end condition is considered always true if a `timing` group contains `sdf_cond_start` but no `sdf_cond_end`.

when_end Simple Attribute

In a `timing` group, `when_end` defines a timing check condition specific to an end event in VITAL models in Synopsys logic format. The expression in this attribute contains any pin, including input, output, I/O, and internal pins. Pins must use real pin names. Bus and bundle names are not allowed.

Example

```
when_end : "CD * SD";
```

The `when_end` attribute requires an `sdf_cond_end` attribute in the same `timing` group.

The start condition is considered always true if a `timing` group contains `when_end` but no `when_start`.

sdf_cond_end Simple Attribute

In a `timing` group, `sdf_cond_end` defines a timing check condition specific to an end event in VITAL models in OVI SDF 2.1 syntax.

Example

```
sdf_cond_end : "SIG_0 == 1'b1";
```

The `sdf_cond_end` attribute requires a `when_end` attribute in the same `timing` group.

The start condition is considered always true if a `timing` group contains `sdf_cond_end` but no `sdf_cond_start`.

sdf_edges Simple Attribute

The `sdf_edges` attribute defines edge-specific information for both the start pins and the end pins in VITAL models. Edge types can be `noedge`, `start_edge`, `end_edge`, or `both_edges`. The default is `noedge`.

Example

```
sdf_edges : both_edges;
```

min_pulse_width Group

In a `pin`, `bus`, or `bundle` group, the `min_pulse_width` group models the enabling conditional minimum pulse width check. In the case of a pin, the timing check is performed on the pin itself, so the related pin must be the same.

The attributes in this group are `constraint_high`, `constraint_low`, `when`, and `sdf_cond`.

If the `pin` group contains a `min_pulse_width` group and either a `min_pulse_width_high` or `min_pulse_width_low` attribute, Library Compiler ignores the attribute.

min_pulse_width Example

The following example shows the `min_pulse_width` group with the `constraint_high` and `constraint_low` attributes specified.

Example 4-13 min_pulse_width Example

```
min_pulse_width() {
  constraint_high : 3.0; /* min_pulse_width_high */
  constraint_low : 3.5; /* min_pulse_width_low */
  when : "SE";
  sdf_cond : "SE == 1'B1";
}
```

constraint_high and constraint_low Simple Attributes

At least one of these attributes must be defined in the `min_pulse_width` group. The `constraint_high` attribute defines the minimum length of time the pin must remain at logic 1. The `constraint_low` attribute defines the minimum length of time the pin must remain at logic 0.

when and sdf_cond Simple Attributes

These attributes define the enabling condition for the timing check: `when` in Synopsys logic format and `sdf_cond` in OVI SDF 2.1 syntax. Both attributes are required in the `min_pulse_width` group. All pins of the `cell`, `input`, `output`, `inout`, or `internal` group can be used with the `when` attribute.

---

minimum_period Group

In a `pin`, `bus`, or `bundle` group, the `minimum_period` group models the enabling conditional minimum period check. In the case of a pin, the check is performed on the pin itself, so the related pin must be the same. The attributes in this group are `constraint`, `when`, and `sdf_cond`.

If the `pin` group contains a `minimum_period` group and a `min_period` attribute, Library Compiler ignores the `min_period` attribute.

minimum_period Example

```
minimum_period() {
  constraint : 9.5; /* min_period */
  when : "SE";
  sdf_cond : "SE == 1'B1";
}
```

constraint Simple Attribute

This required attribute defines the minimum clock period for the pin.

when and sdf_cond Simple Attributes

These attributes define the enabling condition for the timing check: `when` in Synopsys logic format and `sdf_cond` in OVI SDF 2.1 syntax. Both attributes are required in the `minimum_period` group. All pins of the `cell`, `input`, `output`, `inout`, or `internal` group can be used with the `when` attribute.

min_pulse_width and minimum_period Example

Example 4-14 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 4-14 Sample Library With timing_type Statements

```
library(sample) {

   technology (cmos) ;
   delay_model : table_lookup ;

   /* 2-D table template */
   lu_table_template ( mpw ) {
      variable_1 : constrained_pin_transition;
   /* You can replace the constrained_pin_transition value with
   related_pin_transition, but you cannot specify both values. */
      variable_2 : related_out_total_output_net_capacitance;
      index_1("1, 2, 3");
      index_2("1, 2, 3");
   }

   /* 1-D table template */
   lu_table_template( f_ocap ) {
      variable_1 : total_output_net_capacitance;
      index_1 (" 0.0000, 1.0000 ");
   }

   cell( test ) {
      area : 200.000000 ;
      dont_use : true ;
      dont_touch : true ;

      pin ( CK ) {
        direction : input;
        rise_capacitance : 0.00146468;
        fall_capacitance : 0.00145175;
        capacitance : 0.00146468;
        clock : true;

        timing ( mpw_constraint) {
          related_pin : "CK";
          timing_type : min_pulse_width;
          related_output_pin : "Z";

          fall_constraint ( mpw) {
```

```
            index_1("0.1, 0.2, 0.3");
            index_2("0.1, 0.2");
            values( "0.10 0.11", \
              "0.12 0.13" \
              "0.14 0.15");
          }

          rise_constraint ( mpw) {
            index_1("0.1, 0.2, 0.3");
            index_2("0.1, 0.2");
            values( "0.10 0.11", \
              "0.12 0.13" \
              "0.14 0.15");

        timing ( mpw_constraint) {
          related_pin : "CK";
          timing_type : minimum_period;
          related_output_pin : "Z";

          fall_constraint ( mpw) {
            index_1("0.2, 0.4, 0.6");
            index_2("0.2, 0.4");
            values( "0.20 0.22", \
              "0.24 0.26" \
              "0.28 0.30");
          }

          rise_constraint ( mpw) {
            index_1("0.2, 0.4, 0.6");
            index_2("0.2, 0.4");
            values( "0.20 0.22", \
              "0.24 0.26" \
              "0.28 0.30");
         }
      }
   }
   ...
   } /* end of arc */
  } /* end of cell */
} /* end of library */
```

Checking min_pulse_width and minimum_period Constraints

Library Compiler performs `min_pulse_width` and `minimum_period` constraint checks and issues an error message if the following conditions are not met:

- A two-dimensional `min_pulse_width` or `minimum_period` lookup table template must have the `related_out_total_output_net_capacitance` value defined in the index.

- The `related_output_pin` attribute must be specified on the clock pin that has the two-dimensional `min_pulse_width` or `minimum_period` lookup table with the additional output load index.

Using Conditional Attributes With No-Change Constraints

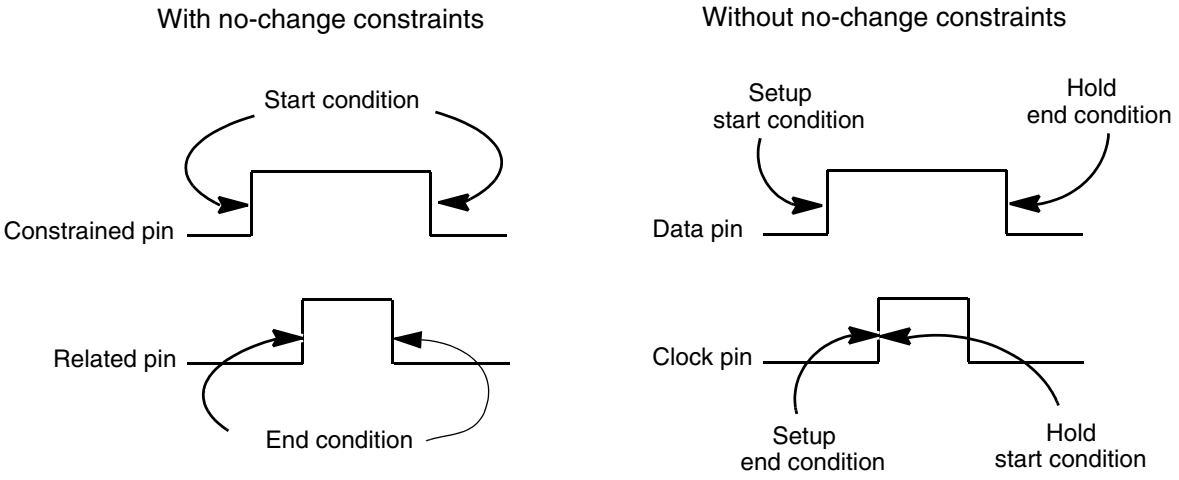
As shown in Table 4-6, conditional timing check attributes have different interpretations when you use them with no-change timing constraints. See "Setting No-Change Timing Constraints" on page 4-83 for a description of no-change timing constraint values.

Table 4-6 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 4-20 shows the different interpretation of setup and hold conditions when used with a no-change timing constraint.

Figure 4-20 Interpretation of Conditional Timing Constraints With No-Change Constraints



Generating an SDF File

SDF file information applies to state-dependent delays as well as to conditional timing constraints.

You can generate an SDF file by first compiling your design, using Design Compiler, and then using the `write_timing` command to write out an SDF file. See the Synopsys man pages for more information on the `write_timing` command.

An SDF file generated by Synopsys tools may contain conditional path delays. You create these path delays by putting the expression string, which you define in the `conditional_port_expr`, into the SDF file of the delay entry. When you generate a VHDL library, the string is written out for back-annotation reference, along with the delay value for the cell component.

When back-annotation is required for simulation, each `conditional_port_expr` in the SDF condition path delay is matched to a string in the VHDL library for an appropriate place to put the delay values that maintain the conditional path delay.

Note:

You must ensure that the string in the `sdf_cond` attribute, defined in the technology library, matches the `conditional_port_expr` statement of the corresponding state-dependent timing arc in SDF files. Library Compiler ignores all white spaces during the string matching.

A common procedure is to use Library Compiler to compile a file that contains one or more `sdf_cond` attributes. This creates a VHDL library containing state-dependent timing information that can be used for SDF condition matching by the Synopsys VHDL Simulation tool to match design rise and fall delays that third-party vendors might consider more accurate. The intrinsic rise and fall delays defined in the library source file are ultimately replaced by the rise and fall delays defined in the design.

Timing Arc Restrictions

The following section describes timing arc limitations.

Impossible Transitions

The information in this section applies to the table lookup and all other delay models and only to combinational and three-state timing arcs. Certain output transitions cannot result from a single input change when `function`, `three_state`, and `x_function` share input.

In the following table, Y is the function of A, B, and C.

A	B	C	Y
0	0	0	Z
0	0	1	1
0	1	0	0
0	1	1	X
1	0	0	0
1	0	1	1
1	1	0	Z
1	1	1	X

No isolated signal change on C can cause the 0-to-1 or 1-to-0 transitions on Y. Therefore, there is no combinational arc from C to Y, although the two are functionally related. Further, no isolated change on A will cause the 1-to-Z or Z-to-1 transitions on Y.

`three_state_enable` has no rising value (Z to1), and `three_state_disable` has no falling value (1 to Z).

Library Compiler detects such invalid timing arcs if they are specified. For the impossible 0-to-1 and 1-to-0 transitions, Library Compiler deletes the timing arc. For the impossible 0-to-Z, Z-to-0,
1-to-Z, and Z-to-1 transitions, Library Compiler puts either the predefined `non_rising` or `non_falling` attribute on the timing arc.

Examples of Libraries Using Delay Models

This section contains examples of libraries using the following CMOS delay models: generic delay, piecewise linear delay, nonlinear delay, and scalable polynomial delay.

CMOS Generic Delay Model

In the CMOS D flip-flop description in Example 4-15, pin D contains setup and hold constraints; pins Q and QN contain preset, clear, and edge-sensitive arcs.

Example 4-15 D Flip-Flop Description (CMOS Generic Delay Model)

```
library (example){
   date : "January 14, 2002";
   revision : 2000.01;
   delay_model : generic_cmos;
   technology (cmos);
   cell( DFLOP_CLR_PRE ) {
      area :  11 ;
      ff ( IQ , IQN ) {
         clocked_on : " CLK ";
         next_state : " D ";
         clear : " CLR' " ;
         preset : " PRE' " ;
         clear_preset_var1 : L ;
         clear_preset_var2 : L ;
      }
      pin ( D ){
         direction : input;
         capacitance : 1 ;
         timing () {
            related_pin : "CLK" ;
            timing_type : hold_rising;
            intrinsic_rise : 0.12;
            intrinsic_fall : 0.12 ;
         }
         timing (){
            related_pin : "CLK" ;
            timing_type : setup_rising ;
            intrinsic_rise : 2.77 ;
            intrinsic_fall : 2.77 ;
         }
```

```
}
pin ( CLK ){
   direction : input;
   capacitance : 1 ;
}
pin ( PRE )    {
   direction : input ;
   capacitance : 2 ;
}
pin ( CLR ){
   direction : input ;
   capacitance : 2 ;
}
pin ( Q ) {
   direction : output;
   function : "IQ" ;
   timing () {
      related_pin : "PRE" ;
      timing_type : preset ;
      timing_sense : negative_unate ;
      intrinsic_rise : 0.65 ;
      rise_resistance : 0.047 ;
      slope_rise : 1.0 ;
   }
   timing (){
      related_pin : "CLR" ;
      timing_type : clear ;
      timing_sense : positive_unate ;
      intrinsic_fall : 1.45 ;
      fall_resistance : 0.066 ;
      slope_fall : 0.8 ;
   }
   timing (){
      related_pin : "CLK" ;
      timing_type : rising_edge;
      intrinsic_rise : 1.40 ;
      intrinsic_fall : 1.91 ;
      rise_resistance : 0.071 ;
      fall_resistance : 0.041 ;
   }
}
pin ( QN ){
   direction : output ;
   function : "IQN" ;
   timing (){
      related_pin : "PRE" ;
      timing_type : clear ;
      timing_sense : positive_unate ;
      intrinsic_fall : 1.87 ;
      fall_resistance : 0.053 ;
      slope_fall : 1.2 ;
   }
   timing (){
```

```
            related_pin : "CLR" ;
            timing_type : preset ;
            timing_sense : negative_unate ;
            intrinsic_rise : 0.68 ;
            rise_resistance : 0.054 ;
            slope_rise : 1.0 ;
         }
         timing (){
            related_pin : "CLK" ;
            timing_type : rising_edge;
            intrinsic_rise : 2.37 ;
            intrinsic_fall : 2.51 ;
            rise_resistance : 0.036 ;
            fall_resistance : 0.041 ;
         }
      }
   }
}
```

CMOS Piecewise Linear Delay Model

In the CMOS piecewise linear D flip-flop description in Example 4-16, pin D contains setup and hold constraints; pins Q and QN contain preset, clear, and edge-sensitive arcs.

Example 4-16 D Flip-Flop Description (CMOS Piecewise Linear Delay Model)

```
library (example){
date : "January 14, 2002";
revision : 2000.01;
delay_model : piecewise_cmos;
technology (cmos);
piece_define("0,15,40");
cell( DFLOP_CLR_PRE ) {
   area : 11 ;
   ff ( IQ , IQN ) {
      clocked_on : " CLK ";
      next_state : " D ";
      clear : " CLR' " ;
      preset : " PRE' " ;
      clear_preset_var1 : L ;
      clear_preset_var2 : L ;
   }
   pin ( D ){
      direction : input;
      capacitance : 1 ;
      timing () {
         related_pin : "CLK" ;
         timing_type : hold_rising;
         intrinsic_rise : 0.12;
         intrinsic_fall : 0.12 ;
      }
      timing (){
```

```
      related_pin : "CLK" ;
      timing_type : setup_rising ;
      intrinsic_rise : 2.77 ;
      intrinsic_fall : 2.77 ;
   }
}
pin ( CLK ){
   direction : input;
   capacitance : 1 ;
}
pin ( PRE )    {
   direction : input ;
   capacitance : 2 ;
}
pin ( CLR ){
   direction : input ;
   capacitance : 2 ;
}
pin ( Q ) {
   direction : output;
   function : "IQ" ;
   timing () {
      related_pin : "PRE" ;
      timing_type : preset ;
      timing_sense : positive_unate ;
      intrinsic_rise : 0.65 ;
      rise_delay_intercept (0,0.054);  /* piece 0 */
      rise_delay_intercept (1,0.0);  /* piece 1 */
      rise_delay_intercept (2,-0.062); /* piece 2 */
      rise_pin_resistance (0,0.25); /* piece 0 */
      rise_pin_resistance(1,0.50); /* piece 1 */
      rise_pin_resistance (2,1.00); /* piece 2 */
   }
   timing (){
      related_pin : "CLR" ;
      timing_type : clear ;
      timing_sense : negative_unate ;
      intrinsic_fall : 1.45 ;
      fall_delay_intercept (0,1.0); /* piece 0 */
      fall_delay_intercept (1,0.0); /* piece 1 */
      fall_delay_intercept (2,-1.0); /* piece 2 */
      fall_pin_resistance (0,0.25); /* piece 0 */
      fall_pin_resistance (1,0.50); /* piece 1 */
      fall_pin_resistance (2,1.00); /* piece 2 */
   }
   timing (){
      related_pin : "CLK" ;
      timing_type : rising_edge;
      intrinsic_rise : 1.40 ;
      intrinsic_fall : 1.91 ;
      rise_pin_resistance (0,0.25); /* piece 0 */
      rise_pin_resistance (1,0.50); /* piece 1 */
      rise_pin_resistance (2,1.00); /* piece 2 */
```

```
            fall_pin_resistance (0,0.15); /* piece 0 */
            fall_pin_resistance (1,0.40); /* piece 1 */
            fall_pin_resistance (2,0.90); /* piece 2 */
         }
      }
      pin ( QN ){
         direction : output ;
         function : "IQN" ;
         timing (){
            related_pin : "PRE" ;
            timing_type : clear ;
            timing_sense : negative_unate ;
            intrinsic_fall : 1.87 ;
            fall_delay_intercept (0,1.0); /* piece 0 */
            fall_delay_intercept (1,0.0); /* piece 1 */
            fall_delay_intercept (2,-1.0); /* piece 2 */
            fall_pin_resistance (0,0.25); /* piece 0 */
            fall_pin_resistance (1,0.50); /* piece 1 */
            fall_pin_resistance (2,1.00); /* piece 2 */
         }
         timing (){
            related_pin : "CLR" ;
            timing_type : preset ;
            timing_sense : positive_unate ;
            intrinsic_rise : 0.68 ;
            rise_delay_intercept (0,0.054);  /* piece 0 */
            rise_delay_intercept (1,0.0);  /* piece 1 */
            rise_delay_intercept (2,-0.062); /* piece 2 */
            rise_pin_resistance (0,0.25);  /* piece 0 */
            rise_pin_resistance(1,0.50);  /* piece 1 */
            rise_pin_resistance (2,1.00);  /* piece 2 */
         }
         timing (){
            related_pin : "CLK" ;
            timing_type : rising_edge;
            intrinsic_rise : 2.37 ;
            intrinsic_fall : 2.51 ;
            rise_pin_resistance (0,0.25); /* piece 0 */
            rise_pin_resistance (1,0.50); /* piece 1 */
            rise_pin_resistance (2,1.00); /* piece 2 */
            fall_pin_resistance (0,0.15); /* piece 0 */
            fall_pin_resistance (1,0.40); /* piece 1 */
            fall_pin_resistance (2,0.90); /* piece 2 */
         }
      }
}
```

CMOS Nonlinear Delay Model

In the nonlinear library description in Example 4-17, pin D contains setup and hold constraints; pins Q and QN contain preset, clear, and edge-sensitive arcs.

Example 4-17 D Flip-Flop Description (CMOS Nonlinear Delay Model)

```
library (NLDM){
date : "January 14, 2002";
revision : 2000.01;
delay_model : table_lookup;
technology (cmos);
/* Define template of size 2 x 2*/
lu_table_template(cell_template) {
   variable_1 : input_net_transition;
   variable_2 : total_output_net_capacitance;
   index_1 ("0.0, 1.5");
   index_2 ("0.0, 4.0");
}
/* Define one-dimensional lu_table of size 4 */
lu_table_template(tran_template) {
   variable_1 : total_output_net_capacitance;
   index_1 ("0.0, 0.5, 1.5, 2.0");
}
cell( DFLOP_CLR_PRE ) {
   area :  11 ;
   ff ( IQ , IQN ) {
      clocked_on : " CLK ";
      next_state : " D ";
      clear : " CLR' " ;
      preset : " PRE' " ;
      clear_preset_var1 : L ;
      clear_preset_var2 : L ;
   }
   pin ( D ){
      direction : input;
      capacitance : 1 ;
      timing () {
         related_pin : "CLK" ;
         timing_type : hold_rising;
         rise_constraint(scalar) {
            values("0.12");
         }
         fall_constraint(scalar) {
            values("0.29");
         }
      }
      timing (){
         related_pin : "CLK" ;
         timing_type : setup_rising ;
         rise_constraint(scalar) {
            values("2.93");
         }
         fall_constraint(scalar) {
            values("2.14");
         }
      }
   }
   pin ( CLK ){
```

```
      direction : input;
      capacitance : 1 ;
   }
   pin ( PRE )    {
      direction : input ;
      capacitance : 2 ;
   }
   pin ( CLR ){
      direction : input ;
      capacitance : 2 ;
   }
   pin ( Q ) {
      direction : output;
      function : "IQ" ;

      timing () {
         related_pin : "PRE" ;
         timing_type : preset ;
         timing_sense : negative_unate ;
         cell_rise(cell_template) {
         values("0.00, 0.23", "0.11, 0.28");
         }
         rise_transition(tran_template) {
         values("0.01, 0.12, 0.15, 0.40");
         }
      }
      timing (){
         related_pin : "CLR" ;
         timing_type : clear ;
         timing_sense : positive_unate ;
         cell_fall(cell_template) {
         values("0.00, 0.24", "0.15, 0.26");
         }
         fall_transition(tran_template) {
         values("0.03, 0.15, 0.18, 0.38");
         }
      }
      timing (){
         related_pin : "CLK" ;
         timing_type : rising_edge;
         cell_rise(cell_template) {
         values("0.00, 0.25", "0.11, 0.28");
         }
         rise_transition(tran_template) {
         values("0.01, 0.08, 0.15, 0.40");
         }
         cell_fall(cell_template) {
         values("0.00, 0.33", "0.11, 0.38");
         }
         fall_transition(tran_template) {
         values("0.01, 0.11, 0.18, 0.40");
         }
      }
```

```
    }
    pin ( QN ){
       direction : output ;
       function : "IQN" ;
       timing (){
          related_pin : "PRE" ;
          timing_type : clear ;
          timing_sense : positive_unate ;
          cell_fall(cell_template) {
          values("0.00, 0.23", "0.11, 0.28");
          }
          fall_transition(tran_template) {
          values("0.01, 0.12, 0.15, 0.40");
          }
       }
       timing (){
          related_pin : "CLR" ;
          timing_type : preset ;
          timing_sense : negative_unate ;
          cell_rise(cell_template) {
          values("0.00, 0.23", "0.11, 0.28");
          }
          rise_transition(tran_template) {
          values("0.01, 0.12, 0.15, 0.40");
          }
       }
       timing (){
          related_pin : "CLK" ;
          timing_type : rising_edge;
          cell_rise(cell_template) {
          values("0.00, 0.25", "0.11, 0.28");
          }
          rise_transition(tran_template) {
          values("0.01, 0.08, 0.15, 0.40");
          }
          cell_fall(cell_template) {
          values("0.00, 0.33", "0.11, 0.38");
          }
          fall_transition(tran_template) {
          values("0.01, 0.11, 0.18, 0.40");
          }
       }
    }
  }
}
```

CMOS Scalable Polynomial Delay Model

In the scalable polynomial delay model in Example 4-18, pin D contains setup and hold constraints; pins Q and QN contain preset, clear, and edge-sensitive arcs.

Example 4-18 D Flip-Flop Description (CMOS Scalable Polynomial Delay Model)

```
library (SPDM) {
   technology (cmos);
   date : "September 19, 2002" ;
   revision : 2002.01 ;
   delay_model : polynomial ;
   /* Define template of 2D polynomial */
   poly_template(cell_template) {
      variables(input_net_transition, total_output_net_capacitance) ;
      variable_1_range(0.0, 1.5) ;
      variable_2_range(0.0, 4.0) ;
   }
   /* Define template of 1D polynomial */
   poly_template(tran_template) {
      variables(total_output_net_capacitance);
      variable_1_range(0.0, 2.0) ;
   }
   cell(DFLOP_CLR_PRE) {
   area : 11;
   ff(IQ, IQN) {
      clocked_on : "CLK" ;
      next_state : "D" ;
      clear : "CLR'";
      preset : "PRE'" ;
      clear_preset_var1 : L ;
      clear_preset_var2 : L ;
   }
   pin(D) {
      direction : input ;
      capacitance : 1 ;
      timing () {
         related_pin : "CLK" ;
         timing_type : hold_rising ;
         rise_constraint(scalar) {
            values("0.12") ;
         }
         fall_constraint(scalar) {
            values("0.29") ;
         }
      } /* end timing */
      timing () {
         related_pin : "CLK" ;
         timing_type : setup_rising ;
         rise_constraint(scalar) {
            values("2.93") ;
         }
         fall_constraint(scalar) {
            values("2.14") ;
         }
      } /* end timing */
   } /* end pin D */
   pin(CLK) {
      direction : input;
      capacitance : 1 ;
   }
   pin(PRE) {
      direction : input;
      capacitance : 2 ;
```

```
}
pin(CLR) {
   direction : input;
   capacitance : 2 ;
}
pin(Q) {
   direction : output ;
   function : "IQ" ;
   timing () {
      related_pin : "PRE" ;
      timing_type : preset ;
      timing_sense : negative_unate ;
      cell_rise(cell_template) {
         orders("1, 1") ;
         coefs("0.1632, 3.0688, 0.0013, 0.0320") ;
         variable_1_range (0.01, 3.00);
         variable_2_range (0.01, 3.00);
      }
      rise_transition(tran_template) {
         orders("1");
         coefs("0.2191, 1.7580") ;
         variable_1_range (0.01, 3.00);
         variable_2_range (0.01, 3.00);
      }
   } /* end timing */
   timing () {
      related_pin : "CLR" ;
      timing_type : clear ;
      timing_sense : positive_unate ;
      cell_fall(cell_template) {
         orders("1, 1") ;
         coefs("0.0542, 6.3294, 0.0214, -0.0310") ;
         variable_1_range (0.01, 3.00);
         variable_2_range (0.01, 3.00);
      }
      fall_transition(tran_template) {
         orders("1") ;
         coefs("0.0652, 2.9232") ;
         variable_1_range (0.01, 3.00);
         variable_2_range (0.01, 3.00);
      }
   } /* end timing */
   timing () {
      related_pin : "CLK" ;
      timing_type : rising_edge ;
      cell_rise(cell_template) {
         orders("1, 1") ;
         coefs("0.1687, 3.0627, 0.0194, 0.0155") ;
         variable_1_range (0.01, 3.00);
         variable_2_range (0.01, 3.00);
      }
      rise_transition(tran_template) {
         orders("1");
         coefs("0.2130, 1.7576") ;
      }
      cell_fall(cell_template) {
         orders("1, 1") ;
         coefs("0.0539, 6.3360, 0.0194, -0.0289") ;
```

```
            variable_1_range (0.01, 3.00);
            variable_2_range (0.01, 3.00);
        }
        fall_transition(tran_template) {
            orders("1");
            coefs("0.0647, 2.9220") ;
            variable_1_range (0.01, 3.00);
            variable_2_range (0.01, 3.00);
        }
    } /* end timing */
} /* end pin Q */
pin(QN) {
    direction : output ;
    function : "IQN" ;
    timing () {
        related_pin : "PRE" ;
        timing_type : clear ;
        timing_sense : positive_unate ;
        cell_fall(cell_template) {
            orders ("1, 1");
            coefs("0.1605, 3.0639, 0.0325, 0.0104") ;
            variable_1_range (0.01, 3.00);
            variable_2_range (0.01, 3.00);
        }
        fall_transition(tran_template) {
            orders ("1") ;
            coefs("0.1955, 1.7535") ;
            variable_1_range (0.01, 3.00);
            variable_2_range (0.01, 3.00);
        }
    } /* end timing */
    timing () {
        related_pin : "CLR" ;
        timing_type : preset ;
        timing_sense : negative_unate ;
        cell_rise(cell_template) {
            orders ("1, 1") ;
            coefs("0.0540, 6.3849, 0.0211, -0.0720" );
            variable_1_range (0.01, 3.00);
            variable_2_range (0.01, 3.00);
        }
        rise_transition(tran_template) {
            orders ("1") ;
            coefs("0.0612, 2.9541") ;
            variable_1_range (0.01, 3.00);
            variable_2_range (0.01, 3.00);
        }
    } /* end timing */
    timing () {
        related_pin : "CLK" ;
        timing_type : rising_edge ;
        cell_rise(cell_template) {
            orders ("1, 1") ;
            coefs ("0.2407, 3.1568, 0.0129, 0.0143") ;
            variable_1_range (0.01, 3.00);
            variable_2_range (0.01, 3.00);
        }
        rise_transition(tran_template) {
```

```
            orders ("1") ;
            coefs ("0.3355, 1.7578") ;
            variable_1_range (0.01, 3.00);
            variable_2_range (0.01, 3.00);
         }
         cell_fall(cell_template) {
            orders("1, 1") ;
            coefs("0.0742, 6.3452, 0.0260, -0.0938") ;
            variable_1_range (0.01, 3.00);
            variable_2_range (0.01, 3.00);
         }
         fall_transition(tran_template) {
            orders ("1") ;
            coefs("0.0597, 2.9997") ;
            variable_1_range (0.01, 3.00);
            variable_2_range (0.01, 3.00);
         }
      } /* end timing */
   } /* end pin QN */
} /* end library */
```

Clock Insertion Delay Example

```
library( vendor_a ) {
   /* 1. Use delay polynomial to mix both lookup table and polynomials*/
   delay_model :polynomial;
   /* 2. Define library-level one-dimensional lu_table of size 4 */
   lu_table_template(lu_template) {
      variable_1 :input_net_transition;
      index_1 ("0.0, 0.5, 1.5, 2.0");
   }
   /* 3. Define library-level poly_template with only one variable*/
   poly_template(poly_template) {
      variables(input_net_transition);
      variable_1_range (0.0, 2.0);
   }
   /* 4. Define a cell and pins within it which has clock tree path*/
   cell (general) {
      ...
      pin(clk) {
         direction:input;
         timing() {
            timing_type :max_clock_tree_path;
            timing_sense :positive_unate;
            cell_rise(lu_template) {
               values ("0.1, 0.15, 0.20, 0.29");
            }
            cell_fall(lu_template) {
               values ("0.2, 0.25, 0.30, 0.39");
            }
            rise_transition(poly_template) {
               orders("2");
               coefs("0.1, 0.2, 0.3");
            }
            fall_transition(poly_template) {
               orders("2");
```

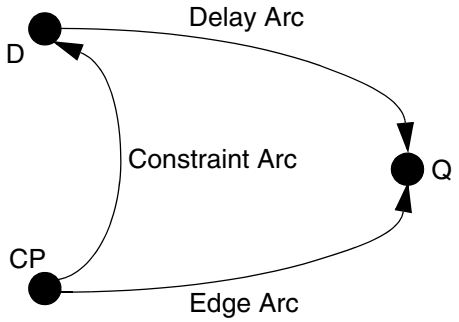
```
                coefs("0.4, 0.5, 0.6");
            }
        }
        timing() {
            timing_type :min_clock_tree_path;
            timing_sense :positive_unate;
            cell_rise(lu_template) {
                values ("0.2, 0.35, 0.40, 0.59");
            }
            cell_fall(lu_template) {
                values ("0.3, 0.45, 0.50, 0.69");
            }
            rise_transition(poly_template) {
                orders("2");
                coefs("0.2, 0.3, 0.4");
            }
            fall_transition(poly_template) {
                orders("2");
                coefs("0.5, 0.6, 0.7");
            }
        }
    }
  ...
}
```

Describing a Transparent Latch Clock Model

The `tlatch` group lets you specify a functional latch when the latch arcs are absent. You use the `tlatch` group at the data pin level to specify the relationship between the data pin and the enable pin on a latch.

Figure 4-21 shows a transparent latch timing model.

Figure 4-21 Transparent Latch Timing Model



Syntax

```
library (name_string) {cell (name_string) {...timing_model_type : value_enum ;
....  pin (data_pin_name_string) {tlatch (enable_pin_name_string){edge_type :
value_enum ;  tdisable : value_Boolean ;}}}}
```

The tlatch name specifies the enable pin that defines the latch clock pin. You define the tlatch group in a pin group, but it is is only effective if you also define the timing_model_type attribute in the cell that the pin belongs to. The timing_model_type attribute can have the following values: “abstracted,” extracted,” and “qtm.”

A tlatch group is optional. You can define one or more tlatch groups for a pin, but you must not define more than two tlatch groups between the same pair of data and enable pins, one rising and one falling. Also, the data pin and the enable pin must be different.

Pins in the tlatch group can be input or inout pins or internal pins. When a tlatch group is not present, a timing analysis tool infers the latch clock pin based on the presence of:

- A related_pin statement in a timing group with either a rising_edge or falling_edge timing_type value within a latch output pin
- A related_pin statement in a timing group with a setup, hold_rising, or falling timing_type value within a latch input pin

The edge_type attribute defines whether the latch is positive (high) transparent or negative (low) transparent. The rising and falling edge_type values specify the opening edge, and therefore the transparent window of the latch, and completely define the latch to be level-high transparent or level-low transparent.

When a tlatch group is not present, a timing analysis tool infers transparency on an output pin based on the timing arc attribute and the presence of a latch functional construct on that pin.

The rising and falling edge_type attribute values explicitly define the transparency windows

- When the rising_edge and falling_edge timing_type values are missing
- When the rising_edge and falling_edge timing_type values are different from the latch transparency

The tdisable attribute disables transparency in a latch. During path propagation, timing analysis tools disable and ignore all data pin output pin arcs that reference a tlatch group whose tdisable attribute is set to true on an edge triggered flip flop.

Example

```
pin (D) {
   tlatch (CP) {
      edge_type : rising ;
      tdisable : true ;
   }
}
pin (Q) {
   direction : output ;
   timing () {
      timing_)sense : positive_unate ;
```

```
      related_pin : D ;
      cell_rise ...
   }
   timing () {
      /* optional arc that can differ from edge_type */
      timing_type : falling_edge ;
      related_pin : CP ;
      cell_fall ...
   }
}
```

Checking Timing and Nonlinear Delay Data

You can use the NLDM library screener to check for timing data and nonlinear delay errors in your library models. For information about using the NLDM library screener and for descriptions of the nonlinear delay noise variables, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Driver Waveform Support

In cell characterization, the shape of the waveform driving the characterized circuit can have a significant impact on the final results. Typically, the waveform is generated by a simple piecewise linear (PWL) waveform or an active-driver cell (a buffer or inverter).

Liberty supports driver waveform syntax, which specifies the type of waveform that is applied to library cells during characterization. The driver waveform syntax helps facilitate the characterization process for existing libraries and correlation checking. The driver waveform requirements can vary. Possible usage models include:

- Using a common driver waveform for all cells.
- Using a different driver waveform for different categories of cells.
- Using a pin-specific driver waveform for complex cell pins.
- Using a different driver waveform for rise and fall timing arcs.

Syntax

The driver waveform syntax is as follows:

```
library(library_name) {
   …
   lu_table_template (waveform_template_name) {
     variable_1: input_net_transition;
     variable_2: normalized_voltage;
     index_1 ("float…, float");
```

```
    index_2 ("float…, float");
  }
  normalized_driver_waveform(waveform_template_name) {
    driver_waveform_name : string; /* Specifies the name of the driver
          waveform table */
    index_1 ("float…, float"); /* Specifies input net transition */
    index_2 ("float…, float"); /* Specifies normalized voltage */
    values ( "float…, float", \ /* Specifies the time in library units */
          …, \
          "float…, float");
  }
  …
  cell (cell_name) {
    …
    driver_waveform : string;
    driver_waveform_rise : string;
    driver_waveform_fall : string;
    pin (pin_name) {
      driver_waveform : string;
      driver_waveform_rise : string;
      driver_waveform_fall : string;
      …
    }
  }
}/* end of library*/
```

In the driver waveform syntax, the first index value in the table specifies the input slew and the second index value specifies the voltage normalized to VDD. The values in the table specify the time in library units (not scaled) when the waveform crosses the corresponding voltages. The `driver_waveform_name` attribute specified for the driver waveform table differentiates the tables when multiple driver waveform tables are defined.

The cell-level `driver_waveform`, `driver_waveform_rise` and `driver_waveform_fall` attributes meet cell-specific and rise- and fall-specific requirements. The attributes refer to the driver waveform table name predefined at the library level.

Similar to the cell-level driver waveform attributes, the pin group includes the `driver_waveform`, `driver_waveform_rise` and `driver_waveform_fall` attributes to meet pin-specific predriver requirements for complex cell pins (such as macro cells). These attributes also refer to the predefined driver waveform table name.

Library-Level Tables, Attributes, and Variables

This section describes driver waveform tables, attributes, and variables that are specified at the library level.

normalized_voltage Variable

The `normalized_voltage` variable is specified under the `lu_table_template` table in order to describe a collection of waveforms under various input slew values. For a given input slew in `index_1` (for example, index_1[0] = 1.0 ns), the `index_2` values are a set of points that represent how the voltage rises from 0 to VDD in a rise arc, or from VDD to 0 in a fall arc.

Rise Arc Example

```
normalized_driver_waveform (waveform_template) {
   index_1 ("1.0"); /* Specifies the input net transition*/
   index_2 ("0, 0.1, 0.3, 0.5, 0.7, 0.9, 1.0"); /* Specifies
             the voltage normalized to VDD */
   values ("0, 0.2, 0.4, 0.6, 0.8, 0.9, 1.1"); /* Specifies        the
time when the voltage reaches the index_2 values*/
}
```

The `lu_table_template` table represents an input slew of 1.0 ns, when the voltage is 0%, 10%, 30%, 50%, 70%, 90% or 100% of VDD, and the time values are 0, 0.2, 0.4, 0.6, 0.8, 0.9, 1.1 (ns). Note that the time value can go beyond the corresponding input slew because a long tail might exist in the waveform before it reaches the final status.

normalized_driver_waveform Group

The library-level `normalized_driver_waveform` group represents a collection of driver waveforms under various input slew values. The `index_1` specifies the input slew and `index_2` specifies the normalized voltage.

Note that the slew index in the `normalized_driver_waveform` table is based on the slew derate and slew trip points of the library (global values). When applied on a pin or cell with different slew or slew derate, the new slew should be interpreted from the waveform.

driver_waveform_name Attribute

The `driver_waveform_name` string attribute differentiates the driver waveform table from other driver waveform tables when multiple tables are defined. Cell-specific and rise- and fall-specific driver waveform usage modeling depend on this attribute.

The `driver_waveform_name` attribute is optional. You can define a driver waveform table without the attribute, but there can be only one table in a library, and that table is regarded as the default driver waveform table for all cells in the library. If more than one table is defined without the attribute, the last table is used. The other tables are ignored and not stored in the .db file.

Cell-level Attributes

This section describes driver waveform attributes defined at the cell level.

driver_waveform Attribute

The `driver_waveform` attribute is an optional string attribute that allows you to define a cell-specific driver waveform. The value must be the `driver_waveform_name` predefined in the `normalized_driver_waveform` table. Otherwise, Library Compiler issues an error.

When the attribute is defined, the cell uses the specified driver waveform during characterization. When it is not specified, the common driver waveform (the `normalized_driver_waveform` table without the `driver_waveform_name` attribute) is used for the cell.

driver_waveform_rise and driver_waveform_fall Attributes

The `driver_waveform_rise` and `driver_waveform_fall` string attributes are similar to the `driver_waveform` attribute. These two attributes allow you to define rise-specific and fall-specific driver waveforms. The `driver_waveform` attribute can coexist with the `driver_waveform_rise` and `driver_waveform_fall` attributes, though the `driver_waveform` attribute becomes redundant.

You should specify a driver waveform for a cell by using the following priority:

1. Use the `driver_waveform_rise` for a rise arc and the `driver_waveform_fall` for a fall arc during characterization. If they are not defined, specify the second and third priority driver waveforms.
2. Use the cell-specific driver waveform (defined by the `driver_waveform` attribute).
3. Use the library-level default driver waveform (defined by the `normalized_driver_waveform` table without the `driver_waveform_name` attribute).

The `driver_waveform_rise` attribute can refer to a `normalized_driver_waveform` that is either rising or falling. You can invert the waveform automatically during runtime if necessary.

Pin-Level Attributes

This section describes driver waveform attributes defined at the pin level.

driver_waveform Attribute

The `driver_waveform` attribute is the same as the `driver_waveform` attribute specified at the cell level. For more information, see “driver_waveform Attribute” on page 4-116.

driver_waveform_rise and driver_waveform_fall Attributes

The `driver_waveform_rise` and `driver_waveform_fall` attributes are the same as the `driver_waveform_rise` and `driver_waveform_fall` attributes specified at the cell level. For more information, see "driver_waveform_rise and driver_waveform_fall Attributes" on page 4-116.

Checking Driver Waveform Data

Library Compiler checks the `normalized_driver_waveform` table data at the library level and cell level. For information about the types of checks Library Compiler performs, see "Library Screener Checks" in the "Reading and Compiling Libraries" chapter in the *Library Quality Assurance System User Guide*.

Driver Waveform Example

```
library(test_library) {

  lu_table_template(waveform_template) {
     variable_1 : input_net_transition;
     variable_2 : normalized_voltage;
     index_1 ("0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7");
     index_2 ("0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9");
  }

  /* Specifies the default library-level driver waveform table (the
default
          driver waveform without the driver_waveform attribute) */
  normalized_driver_waveform (waveform_template) {
     values ("0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09", \
              "0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19", \
              … … …
              "0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9");
  }
  /* Specifies the driver waveform for the clock pin */
  normalized_driver_waveform (waveform_template) {
     driver_waveform_name : clock_driver;
     index_1 ("0.15, 0.25, 0.35, 0.45, 0.55, 0.65, 0.75");
     index_2 ("0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9");
     values ("0.012, 0.03, 0.045, 0.06, 0.075, 0.090, 0.105, 0.13,
0.145", \
              … … …
              "0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9");
  }
  /* Specifies the driver waveform for the bus */
  normalized_driver_waveform (waveform_template) {
     driver_waveform_name : bus_driver;
     index_1 ("0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7");
     index_2 ("0.15, 0.25, 0.35, 0.45, 0.55, 0.65, 0.75, 0.85, 0.95");
```

```
      values ("0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09", \
              "0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19", \
              … … …
              "0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9");
   }
   /* Specifies the driver waveform for the rise */
   normalized_driver_waveform (waveform_template) {
       driver_waveform_name : rise_driver;
       index_1 ("0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7");
       index_2 ("0.15, 0.25, 0.35, 0.45, 0.55, 0.65, 0.75, 0.85, 0.95");
       values ("0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09", \
               "0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19", \
               … … …
               "0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9");
   }
   /* Specifies the driver waveform for the fall */
   normalized_driver_waveform (waveform_template) {
      driver_waveform_name : fall_driver;
      index_1 ("0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7");
      index_2 ("0.15, 0.25, 0.35, 0.45, 0.55, 0.65, 0.75, 0.85, 0.95”);
      values ("0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09", \
              "0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19", \
              … … …
              "0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9");
   }
   …
   cell (my_cell1) {
      driver_waveform : clock_driver;
      …
   }
   cell (my_cell2) {
      driver_waveform : bus_driver;
      …
   }
   cell (my_cell3) {
      driver_waveform_rise : rise_driver;
      driver_waveform_fall : fall_driver;
      …
   }
   cell (my_cell4) {
   /* No driver_waveform attribute is specified. Use the default driver
waveform */
      …
   }
} /* End library */
```

Sensitization Support

Timing information specified in libraries results from circuit simulator (library characterization) tools. The models themselves often represent only a partial record of how a particular arc was sensitized during characterization. To fully reproduce the conditions that allowed the model to be generated, the states of all the input pins on the cell must be known.

In composite current source (CCS) models, the accuracy requirements are very high (the expectation is as high as 2 percent compared to SPICE). In order to achieve this level of accuracy, correlation with SPICE requires that the cell conditions be represented exactly as during characterization. For more information about CCS models, see Chapter 5, "Composite Current Source Modeling."

The following sections describe the pin sensitization condition information details used to characterize the timing data. The syntax pre-declares state vectors as reusable sensitization patterns in the library. The patterns are referenced and instantiated as stimuli waveforms specific to timing arcs. The same sensitization pattern can be referenced by multiple cells or multiple timing arcs. One cell can also reference multiple sensitization patterns, which saves storage resources. The Liberty attributes and groups highlighted in the following sections help to specify the sensitization information of timing arcs during simulation and characterization.

sensitization Group

The `sensitization` group defined at the library level describes the complete state patterns for a specific list of pins (defined by the `pin_names` attribute) that will be referenced and instantiated as stimuli in the timing arc.

Vector attributes in the group define all possible pin states used as stimuli. Actual stimulus waveforms can be described by a combination of these vectors. Multiple `sensitization` groups are allowed in a library. Each `sensitization` group can be referenced by multiple cells, and each cell can make reference to multiple `sensitization` groups.

The following attributes are library-level attributes under the `sensitization` group.

pin_names Attribute

The `pin_names` complex attribute defines a default list of pin names. All vectors in this `sensitization` group are the exhaustive list of all possible transitions of the input pins and their subsequent output response.

The `pin_names` attribute is required, and it must be declared in the `sensitization` group before all `vector` declarations.

vector Attribute

Similar to the `pin_names` attribute, the `vector` attribute describes a transition pattern for the specified pins. The stimulus is described by an ordered list of vectors.

The two arguments for the `vector` attribute are as follows:

`vector id`

The `vector id` argument is an identifier to the vector string. The `vector id` value must be an integer greater than or equal to zero and unique among all vectors in the current `sensitization` group.

`vector string`

The `vector string` argument represents a pin transition state. The string consists of the following transition status values: 0, 1, X, and Z where each character is separated by a space. The number of elements in the vector string must equal the number of arguments in `pin_names`.

The `vector` attribute can also be declared as:

```
vector (positive_integer, "[0|1|X|Z] [0|1|X|Z]…");
```

Example

```
sensitization(sensitization_nand2) {
      pin_names ( IN1, IN2, OUT1 );
      vector ( 1, "0 0 1" );
      vector ( 2, "0 1 1" );
      vector ( 3, "1 0 1" );
      vector ( 4, "1 1 0" );
```

Cell-Level Attributes

Generally, one cell references one `sensitization` group for cells. A cell-level attribute that can link the cell with a specific `sensitization` group helps to simplify sensitization usage. The following cell-level attributes ensure that `sensitization` groups can be referenced by cells with similar functionality but can have different pin names.

sensitization_master Attribute

The `sensitization_master` attribute defines the sensitization group referenced by the cell to generate stimuli for characterization. The attribute is required if the cell contains sensitization information. Its string value should be any `sensitization` group name predefined in the current library.

pin_name_map Attribute

The pin_name_map attribute defines the pin names that are used to generate stimuli from the sensitization group for all timing arcs in the cell. The pin_name_map attribute is optional when the pin names in the cell are the same as the pin names in the sensitization master, but it is required when they are different.

If the pin_name_map attribute is set, the number of pins must be the same as that in the sensitization master, and all pin names should be legal pin names in the cell.

timing Group Attributes

This section describes the sensitization_master and pin_name_map timing-arc attributes. These attributes enable a complex cell (a macro in most cases) to refer to multiple sensitization groups. You can also specify a sampling vector and user-defined time intervals between vectors. The wave_rise and wave_fall attributes, which describe characterization stimuli (instantiated in pin timing arcs), are also discussed in this section.

sensitization_master Attribute

The sensitization_master simple attribute defines the sensitization group specific to the current timing group to generate stimulus for characterization. The attribute is optional when the sensitization master used for the timing arc is the same as that defined in the current cell, and it is required when they are different. Any sensitization group name predefined in the current library is a valid attribute value.

pin_name_map Attribute

Similar to the pin_name_map attribute defined in the cell level, the timing-arc pin_name_map attribute defines pin names used to generate stimulus for the current timing arc. The attribute is optional when pin_name_map pin names are the same as the following (listed in order of priority):

1. Pin names in the sensitization_master of the current timing arc.
2. Pin names in the pin_name_map attribute of the current cell group.
3. Pin names in the sensitization_master of the current cell group.

The pin_name_map attribute is required when pin_name_map pin names are different from all of the pin names in the previous list.

wave_rise and wave_fall Attributes

The `wave_rise` and `wave_fall` attributes represent the two stimuli used in characterization. The value for both attributes is a list of integer values, and each value is a vector ID predefined in the library sensitization group. The following example describes the `wave_rise` and `wave_fall` attributes:

```
wave_rise (vector_id[m]…, vector_id[n]);
wave_fall (vector_id[j]…, vector_id[k]);
```

Example

```
library(my_library) {
…
sensitization(sensi_2in_1out) {
   pin_names (IN1, IN2, OUT);
   vector (0, "0 0 0");
   vector (1, "0 0 1");
   vector (2, "0 1 0");
   vector (3, "0 1 1");
   vector (4, "1 0 0");
   vector (5, "1 0 1");
   vector (6, "1 1 0");
   vector (7, "1 1 1");
}
cell (my_nand2) {
   sensitization_master : sensi_2in_1out;
   pin_name_map (A, B, Z); /* these are pin names for the sensitization
in this
           cell. */
   …
   pin(A) {
      …
   }
   Pin(B) {
      …
   }
   pin(Z) {
      …
      timing() {
         related_pin : "A";
         wave_rise (6, 3); /* 6, 3 - vector id in sensi_2in_1out
sensitization group. Wave form interpretation of the wave_rise is (for
"A, B, Z" pins): 10 1 01 */
         wave_fall (3, 6);
         …
      }
      timing() {
         related_pin : "B";
         wave_rise (7, 4); /* 7, 4 - vector id in sensi_2in_1out
sensitization
                 group. */
         wave_fall (4, 7);
```

```
      …
    }
    } /* end pin(Z)*/
  } /* end cell(my_nand2) */
  …
  } /* end library */
```

wave_rise_sampling_index and wave_fall_sampling_index Attributes

The `wave_rise_sampling_index` and `wave_fall_sampling_index` simple attributes override the default behavior of the `wave_rise` and `wave_fall` attributes.
(The `wave_rise` and `wave_fall` attributes select the first and the last vectors to define the sensitization patterns of the input to the output pin transition that are predefined inside the sensitization template specified at the library level).

By default, Library Compiler assumes that the delay is measured from the last transition of the sensitization description to the transition of the output pin.

Example

```
wave_rise (2, 5, 7, 6); /* wave_rise ( wave_rise[0],
wave_rise[1], wave_rise[2], wave_rise[3] );*/
```

In the previous example, the wave rise vector delay is measured from the last transition (vector 7 changing to vector 6) to the output transition. The default `wave_rise_sampling_index` value is the last entry in the vector, which is 3 in this case (because the numbering begins at 0).

To override this default, set the `wave_rise_sampling_index` attribute, as shown:

```
wave_rise_sampling_index : 2 ;
```

When you specify this attribute, the delay is measured from the second last transition of the sensitization vector to the final output transition, in other words from the transition of vector 5 to vector 7.

Note:
You cannot specify a value of 0 for the `wave_rise_sampling_index` attribute.

wave_rise_time_interval and wave_fall_time_interval Attributes

The `wave_rise_time_interval` and `wave_fall_time_interval` attributes control the time interval between transitions. By default, the stimuli (specified in `wave_rise` and `wave_fall`) are widely spaced apart during characterization (for example, 10 ns from one vector to the next) to allow all output transitions to stabilize. The attributes allow you to specify a short duration between one vector to the next in order to characterize special purpose cells and pessimistic timing characterization.

The wave_rise_time_interval and wave_fall_time_interval attributes are optional when the default time interval is used for all transitions, and they are required when you need to define special time intervals between transitions. Usually, the special time interval is smaller than the default time interval.

The wave_rise_time_interval and wave_fall_time_interval attributes can have an argument count from 1 to *n*-1, where *n* is the number of arguments in corresponding wave_rise or wave_fall. Use 0 to imply the default time interval used between vectors.

Example

```
wave_rise (2, 5, 7, 6); /* wave_rise ( wave_rise[0],
wave_rise[1], wave_rise[2], wave_rise[3] );*/
wave_rise_time_interval (0.0, 0.3);
```

The previous example suggests the following:

- Use the default time interval between wave_rise[0] and wave_rise[1] (in other words, vector 2 and vector 5).
- Use 0.3 between wave_rise[1] and wave_rise[2] (in other words, vector 5 and vector 7).
- Use the default time interval between wave_rise[2] and wave_rise[3] in other words, vector 7 and vector 6).

timing Group Syntax

```
library(<library_name>) {
   …
   sensitization (<sensitization_group_name>) {
    pin_names (string…, string);
    vector (integer, string);
    …
    vector (integer, string);
   }

   …

   cell(<cell_name>) {
      sensitization_master : <sensitization_group_name>;
      pin_name_map (string…, string);
      …
      pin(<pin_name>) {
        …
        timing() {
          related_pin : string;
          sensitization_master : <sensitization_group_name>;
          pin_name_map (string…, string);
          wave_rise (integer…, integer);
```

```
        wave_fall (integer…, integer);
        wave_rise_sampling_index : integer;
        wave_fall_sampling_index : integer;
        wave_rise_timing_interval (float…, float);
        wave_fall_timing_interval (float…, float);
        …
      }/*end of timing */
    }/*end of pin */
     }/*end of cell */
     …
  }/* end of library*/
```

Checking Sensitization Data

Library Compiler checks sensitization information and reports warnings and errors if it encounters issues. For information about the types of sensitization checks Library Compiler performs, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

NAND Cell Example

The following is an example of a NAND cell with sensitization information.

```
library(cell1) {
    sensitization(sensitization_nand2) {
     pin_names ( IN1, IN2, OUT1 );
     vector ( 1, "0 0 1" );
     vector ( 2, "0 1 1" );
     vector ( 3, "1 0 1" );
     vector ( 4, "1 1 0" );
    }

    cell (nand2) {
     sensitization_master : sensitization_nand2;
     pin_name_map (A, B, Y);
     pin (A) {
         direction : input ;
         …
     }
     pin (B) {
         direction : input ;
         …
     }
     pin (Y) {
         direction : output;
         ...
         timing() {
          related_pin : "A";
          timing_sense : negative_unate;
```

```
            wave_rise ( 4, 2 ); /* 10 1 01 */
            wave_fall ( 2, 4 ); /* 01 1 10 */
            ...
          }
          timing() {
            related_pin : "B";
            timing_sense : negative_unate;
            wave_rise ( 4, 3 );
            wave_fall ( 3, 4 );
            ...
          }
      } /* end pin(Y) */
    } /* end cell(nand2) */
} /* end library */
```

Complex Macro Cell Example

The following is an example of a complex macro cell highlighting the usage of the `sensitization_master` group inside the `timing` group.

```
library(cell1) {
    sensitization(sensitization_2in_1out) {
      pin_names ( IN1, IN2, OUT );
      vector ( 1, "0 0 1" );
      vector ( 2, "0 1 1" );
      vector ( 3, "1 0 1" );
      vector ( 4, "1 1 0" );
}
    sensitization(sensitization_3in_1out) {
      pin_names ( IN1, IN2, IN3, OUT );
      vector ( 0, "0 0 0 0" );
      vector ( 1, "0 0 0 1" );
      vector ( 2, "0 0 1 0" );
      vector ( 3, "0 0 1 1" );
      vector ( 4, "0 1 0 0" );
      vector ( 5, "0 1 0 1" );
      vector ( 6, "0 1 1 0" );
      vector ( 7, "0 1 1 1" );
      vector ( 8, "1 0 0 0" );
      vector ( 9, "1 0 0 1" );
      vector ( 10, "1 0 1 0" );
      vector ( 11, "1 0 1 1" );
      vector ( 12, "1 1 0 0" );
      vector ( 13, "1 1 0 1" );
      vector ( 14, "1 1 1 0" );
      vector ( 15, "1 1 1 1" );
}

    cell (nand2) {
      sensitization_master : sensitization_2in_1out;
      pin_name_map (A, B, Y);
```

```
      pin (A) {
          direction : input ;
          …
      }
      pin (B) {
          direction : input ;
          …
      }
      pin (CIN0) {
          direction : input ;
          …
      }
      pin (CIN1) {
          direction : input ;
          …
      }
      pin (CK) {
          direction : input;
          …
      }
      pin (Y) {
          direction : output;
          ...
          timing() {
            related_pin : "A";
      /* inherit sensitization_master & pin_name_map from cell level */
            timing_sense : negative_unate;
            wave_rise ( 4, 2 );
            wave_fall ( 2, 4 );
            ...
          }
          timing() {
            related_pin : "B";
            timing_sense : negative_unate;
            wave_rise ( 4, 3 );
            wave_fall ( 3, 4 );
            ...
          }
      } /* end pin(Y) */
      pin (Z) {
          direction : output;
          …
          timing () {
            related_pin : "CK";
            sensitization_master :  sensitization_3in_1out; /* timing arc
 specific sensitization master, overwrite the cell level attribute. */
            pin_name_map (CIN0, CIN1, CK, Z); /* timing arc specific
 pin_name_map, overwrite the cell level attribute. */
            wave_rise (14, 4, 0, 3, 10, 5); /* the waveform describe here
has
 no real meaning, just select random vector id in sensitization
 sensitization_3in_1out group. */
            wave_fall (15, 9, 3, 1, 6, 7);
```

```
          wave_rise_sampling_index : 3; /* sampling index, specific for
this
 timing arc. */
          wave_fall_sampling_index : 3;
          wave_rise_timing_interval(0, 0.3, 0.3); /* special timing
 interval, specific for this timing arc. */

          wave_fall_timing_interval(0, 0.3, 0.3);
         }
     } /* end pin (Z) */
    } /* end cell(nand2) */
} /* end library */
```

Using the Library Quality Assurance System

The Library Quality Assurance System checks a technology library in Liberty (.lib) format for the completeness, consistency, and accuracy of its cell models, especially composite current source (CCS) models. The Library Quality Assurance System is typically used to verify the integrity of new or changed library cell models that have been created or modified by a characterization tool.

For more information about the Library Quality Assurance System, see the *Library Quality Assurance System User Guide*.

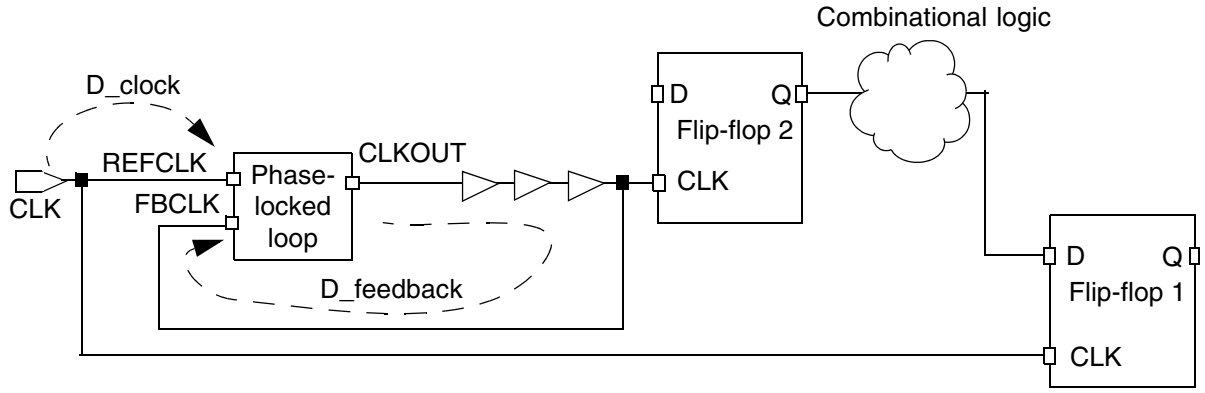
Phase-Locked Loop Support

A phase-locked loop (PLL) is a feedback control system that automatically adjusts the phase of a locally-generated signal to match the phase of an input signal. Phase-locked loops contain the following pins:

- The reference clock pin where the reference clock is connected
- The phase-locked loop output clock pin where the phase-locked loop generates a phase-shifted version of the reference clock
- The feedback pin where the feedback path from the output of the clock ends

Figure 4-22 shows a circuit with a phase-locked loop. Without the phase-locked loop block, there is a large clock skew in the clocks arriving at the launch point and at the flip-flops. This large skew results in an extremely tight delay constraint for the combinational logic. A phase-locked loop reduces the large skew between the launch point and the flip-flops.

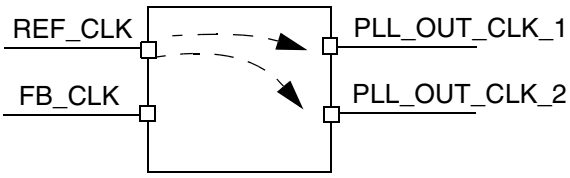
Figure 4-22 Phase-Locked Loop Circuit



The phase-locked loop generates a phase-shifted version of the reference clock at the output pin so that the phase of the feedback clock matches the phase of the reference clock. If the clock arrives at the reference pin of the phase-locked loop at the time, D_clock, and the delay of the feedback path is D_feedback, the source latency of the clock generated at the output of the phase-locked loop is D_clock - D_feedback. The clock arrives at the feedback pin at time, D_clock, which is the same as the time the clock arrives at the reference pin. Therefore, the phase-locked loop eliminates the latency on the launch path and relaxes the delay constraint on the combinational logic.

Figure 4-23 shows a simple phase-locked loop model. The phase-locked loop information that a library should contain are pins and timing arcs inside the phase-locked loop. The forward arcs from REF_CLK to PLL_OUT_CLK_* in the figure simulate the phase shift behavior of the phase-locked loop. There are two half-unate arcs from the reference clock to each output of the phase-locked loop.

Figure 4-23 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 Attributes

This section describes cell-level attributes.

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.

Checking Phase-Locked Loop Libraries

Library Compiler automatically checks phase-locked loop libraries and reports any problems it encounters. For information about the types of checks Library Compiler performs for phase-locked loop libraries, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Phase-Locked Loop Example

```
cell(my_pll) {
  is_pll_cell : true;

  pin( REFCLK ) {
direction : input;
is_pll_reference_pin : true;
  }

  pin( FBKCLK ) {
direction : input;
is_pll_feedback_pin : true;
  }

  pin (OUTCLK_1x) {
direction : output;
is_pll_output_pin : true;
    timing() {/* Timing Arc */
      related_pin: "REFCLK";
      timing_type: combinational_rise;
      timing_sense: positive_unate;
       . . .

    }
    timing() {/* Timing Arc */
      related_pin: "REFCLK";
      timing_type: combinational_fall;
      timing_sense: positive_unate;
      . . .
    }
  }

  pin (OUTCLK_2x) {
direction : output;
is_pll_output_pin : true;
    timing() {/* Timing Arc */
      related_pin: "REFCLK";
      timing_type: combinational_rise;
      timing_sense: positive_unate;
      . . .
    }
    timing() {/* Timing Arc */
```

```
        related_pin: "REFCLK";
        timing_type: combinational_fall;
        timing_sense: positive_unate;
        . . .
      }
    } /* End pin group */
  } /* End cell group */
```

5

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

Existing driver models can deliver acceptable accuracy when output waveforms are mostly linear and interconnect resistance is low. However, as integrated circuit technology advances to nanometer geometries, waveforms can become highly nonlinear and interconnect delay can become a concern. At the same time, faster circuit speeds require more accurate delay calculation.

Composite current source modeling supports additional driver model complexity by using a time- and voltage- dependent current source with essentially an infinite drive resistance. The new driver model achieves high accuracy by not modeling the transistor behavior. Instead, it maps the arbitrary transistor behavior for lumped loads to that for an arbitrary detailed parasitic network.

The composite current source model improves the receiver model accuracy because the input capacitance of a receiver is dynamically adjusted during the transition by using two capacitance values. The driver model can be used with or without the receiver model.

Representing Composite Current Source Driver Information

In the Liberty syntax, using the composite current source model, you can represent nonlinear delay information at the pin level by specifying a current lookup table at the timing group level that is dependent upon input slew and output load.

Composite Current Source Lookup Tables

You can represent composite current source driver models in your libraries by using lookup tables. To define your lookup tables, use the following groups and attributes:

- `output_current_template` group in the `library` group level
- `output_current_rise` and `output_current_fall` groups in the `timing` group level

Defining the output_current_template Group

Use this library-level group to create templates of common information that multiple current vectors can use. A table template specifies the composite current source driver model and the breakpoints for the axis. Specifying `index_1`, `index_2`, and `index_3` values at the library level is optional.

output_current_template Syntax

```
   library(name_id) {...output_current_template(template_name_id) {
variable_1: input_net_transition; variable_2:
total_output_net_capacitance;variable_3: time;...}...

   }
```

Template Variables for CCS Driver Models

The table template specifying composite current source driver models can have three variables: `variable_1`, `variable_2`, and `variable_3`. The valid values for `variable_1` and `variable_2` are `input_net_transition` and `total_output_net_capacitance`. The only valid value for `variable_3` is `time`.

output_current_template Example

```
library (new_lib) {
...
   output_current_template (CCT) {
      variable_1: input_net_transition;
      variable_2: total_output_net_capacitance;
      variable_3: time;
      ...
   }
   ...
}
```

Defining the Lookup Table Output Current Groups

To specify the output current for the nonlinear table model, use the `output_current_rise` and `output_current_fall` groups within the `timing` group.

output_current_rise Syntax

```
timing() {output_current_rise () {vector (template_name_id){reference_time
: float; index_1 (float);index_2 (float);index_3 ("float,..., float");
values("float,..., float");}}}
```

---

vector Group

Define the `vector` group in the `output_current_rise` or `output_current_fall` group. This group stores current information for a particular input slew and output load.

reference_time Simple Attribute

Define the `reference_time` simple attribute in the `vector` group. The `reference_time` attribute represents the time at which the input waveform crosses the rising or falling input delay threshold.

Template Variables for CCS Driver Models

The table template specifying composite current source driver models can have three variables: `variable_1`, `variable_2`, and `variable_3`. The valid values for `variable_1` and `variable_2` are `input_net_transition` and `total_output_net_capacitance`. The only valid value for `variable_3` is `time`.

The index value for `input_net_transition` or `total_output_net_capacitance` is a single floating-point number. The index values for `time` are a list of floating-point numbers.

The `values` attribute defines a list of floating-point numbers that represent the current values of the driver model.

vector Group Example

```
library (new_lib) {
...
   output_current_template (CCT) {
      variable_1: input_net_transition;
      variable_2: total_output_net_capacitance;
      variable_3: time;
   }
   ...
   timing() {
      output_current_rise() {
         vector(CCT) {
            reference_time : 0.05;
            index_1(0.1);
            index_2(2.1);
            index_3("1.0, 1.5, 2.0, 2.5, 3.0");
            values("1.1, 1.2, 1.4, 1.3, 1.5");
         ...
         }
      }
      output_current_fall() {
         vector(CCT) {
            reference_time : 0.05;
            index_1(0.1);
            index_2(2.1);
            index_3("1.0, 1.5, 2.0, 2.5, 3.0");
            values("1.1, 1.2, 1.4, 1.3, 1.5");
         ...
         }
      }
```

```
    }
  }
```

Checking CCS Driver Models

Library Compiler automatically checks CCS driver models and reports any problems it encounters. For information about the types of checks Library Compiler performs for CCS driver models, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Mode and Conditional Timing Support for Pin-Level CCS Receiver Models

Library Compiler supports conditional data modeling in pin-based CCS timing receiver models. The `mode` and `when` attributes are provided in the CCS timing receiver model groups to support this feature.

Library Compiler provides the following support:

- The `mode` and `when` attributes in timing arcs to allow conditional timing arcs and constraints.
- The `mode` and `when` attributes in pin-based expanded CCS timing models and receiver models.

Conditional Timing Support Syntax

Library Compiler provides the following syntax to support conditional data modeling for pin-based CCS timing receiver models.

Syntax

```
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 (lu_template_name) { … }
         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) { … }
              receiver_capacitance1_fall (lu_template_name) { … }
              receiver_capacitance2_rise (lu_template_name) { … }
              receiver_capacitance2_fall (lu_template_name) { … }
          }
    }…
  }
}
```

when Attribute

The `when` string attribute is provided in the pin-based `receiver_capacitance` group to support conditional data modeling.

mode Attribute

The complex `mode` attribute is provided in the pin-based `receiver_capacitance` 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.

Conditional Timing Support Example

```
library(new_lib) {
  ...
  output_current_template(CCT) {
    variable_1: input_net_transition;
    variable_2: total_output_net_capacitance;
    variable_3: time;
    index_1("0.1, 0.2");
    index_2("1, 2");
    index_3("1, 2, 3, 4, 5");
  }
  lu_table_template(LTT1) {
    variable_1: input_net_transition;
    index_1("0.1, 0.2, 0.3, 0.4");
  }
  lu_table_template(LTT2) {
    variable_1: input_net_transition;
    variable_2: total_output_net_capacitance;
    index_1("0.1, 0.2");
    index_2("1, 2");
  }
  ...
  cell(my_cell) {
    ...
```

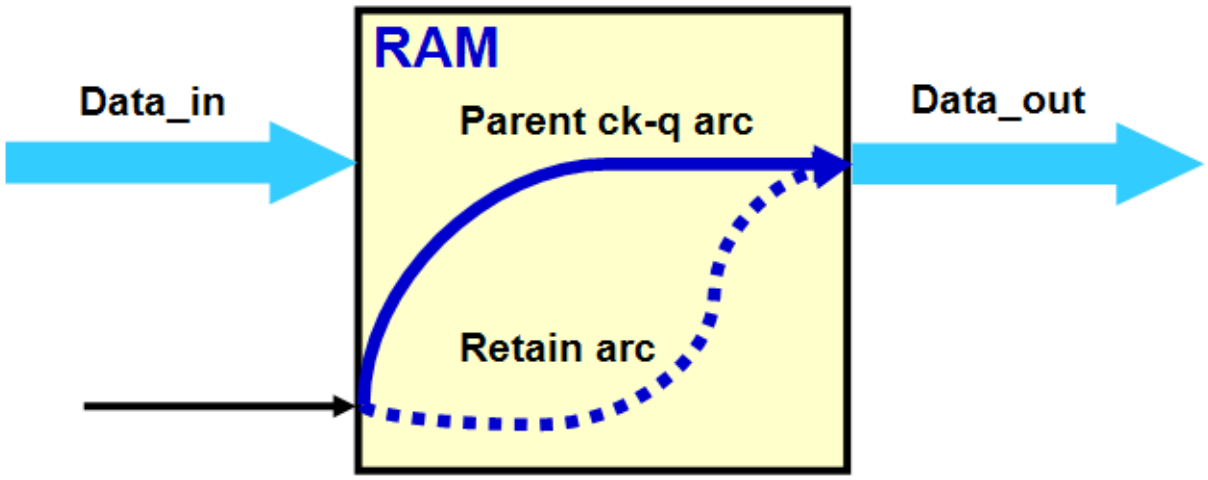
```
    mode_definition(rw) {
      mode_value(read) {
        when : "I";
        sdf_cond : "I == 1";
      }
      mode_value(write) {
        when : "!I";
        sdf_cond : "I == 0";
      }
    }
    pin(I) { /* pin-based receiver model defined for pin 'A' */
      direction : input;
      /* receiver capacitance for condition 1 */
      receiver_capacitance() {
        when : "I"; /* or using mode as next commented line */
        /* mode (rw, read); */
        receiver_capacitance1_rise(LTT1) {
        values("1, 2, 3, 4");
        }
        receiver_capacitance1_fall(LTT1) {
        values("1, 2, 3, 4");
        }
        receiver_capacitance2_rise(LTT1) {
        values("1, 2, 3, 4");
        }
        receiver_capacitance2_fall(LTT1) {
        values("1, 2, 3, 4");
        }
      }
      /* receiver capacitance for condition 2 */
      receiver_capacitance() {
        when : "!I"; /* or using mode as next commented line */
        /* mode (rw, write); */
        receiver_capacitance1_rise(LTT1) {
        values("1, 2, 3, 4");
        }
        receiver_capacitance1_fall(LTT1) {
        values("1, 2, 3, 4");
        }
        receiver_capacitance2_rise(LTT1) {
        values("1, 2, 3, 4");
        }
        receiver_capacitance2_fall(LTT1) {
        values("1, 2, 3, 4");
        }
      }
    }
    pin (ZN) {
      direction : input;
      capacitance : 1.2;
      …
      timing() {
        …
      }
    }
    …
  } /* end cell */
  …
```

```
} /* end library */
```

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 5-1, ensures that the output does not change during this time interval.

Figure 5-1 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 all CCS timing models, including expanded CCS timing models, compact CCS timing models, and variation-aware compact CCS timing models. Because retain arcs have no relation to receiver models, only syntax for CCS driver models is described below.

Syntax

```
library (library_namestring) {
   delay_model : table_lookup;
...
   output_current_template(template_namestring) {
     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(namestring) {
      pin (namestring) {
         timing() {
            ccs_retain_rise() {
            vector(template_namestring) {
              reference_time : float;
               index_1("float");
               index_2("float");
               index_3("float, ..., float");
              values("float, ..., float");
            }
            vector(template_namestring) { . . . } …
            }
            ccs_retain_fall() {
            vector(template_namestring) {
            reference_time : float;
            index_1("float");
            index_2("float");
            index_3("float, ..., float");
            values("float, ..., float");
            }
            vector(template_namestring) { . . . } …
            }
            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. Library Compiler provides the following retain arc syntax to support 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.

Representing Composite Current Source Receiver Information

Composite current source receiver modeling must be used in conjunction with composite current source driver modeling. This model improves the receiver model accuracy.

With source driver modeling, the capacitance is adjusted at the delay threshold. The capacitances used to model the receiver are dependent on input slew and output load.

Composite Current Source Lookup Table Model

Library information for composite current source receiver modeling can be defined

- At the pin level inside the `receiver_capacitance` group
- At the timing level by using the following groups: `receiver_capacitance1_rise receiver_capacitance1_fall receiver_capacitance2_rise receiver_capacitance2_fall`

Values for rise and fall can be defined at the pin or timing level. The pin-level definition does not depend on output capacitance and is useful when there are no forward timing arcs.

Defining the receiver_capacitance Group at the Pin Level

You can define a `receiver_capacitance` group at the pin level. Use the `receiver_capacitance1_rise`, `receiver_capacitance1_fall`, `receiver_capacitance2_rise`, and `receiver_capacitance2_fall` groups.

when Attribute

The `when` string attribute is provided in the pin-based `receiver_capacitance` group to support conditional data modeling.

mode Attribute

The complex `mode` attribute is provided in the pin-based `receiver_capacitance` 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.

receiver_capacitance Group Example

```
cell(my_cell) {
  ...
  mode_definition(rw) {
    mode_value(read) {
      when : "I";
      sdf_cond : "I == 1";
    }
    mode_value(write) {
      when : "!I";
      sdf_cond : "I == 0";
    }
  }
  pin(I) { /* pin-based receiver model defined for pin 'A' */
    direction : input;
    /* receiver capacitance for condition 1 */
    receiver_capacitance() {
      when : "I"; /* or using mode as next commented line */
      /* mode (rw, read); */
      receiver_capacitance1_rise(LTT1) {
      values("1, 2, 3, 4");
      }
      receiver_capacitance1_fall(LTT1) {
      values("1, 2, 3, 4");
      }
      receiver_capacitance2_rise(LTT1) {
      values("1, 2, 3, 4");
      }
      receiver_capacitance2_fall(LTT1) {
      values("1, 2, 3, 4");
      }
    }
    /* receiver capacitance for condition 2 */
    receiver_capacitance() {
      when : "!I"; /* or using mode as next commented line */
```

```
        /* mode (rw, write); */
        receiver_capacitance1_rise(LTT1) {
        values("1, 2, 3, 4");
        }
        receiver_capacitance1_fall(LTT1) {
        values("1, 2, 3, 4");
        }
        receiver_capacitance2_rise(LTT1) {
        values("1, 2, 3, 4");
        }
        receiver_capacitance2_fall(LTT1) {
        values("1, 2, 3, 4");
        }
      }
    }
    pin (ZN) {
      direction : input;
      capacitance : 1.2;
      …
      timing() {
        …
      }
    }
    …
  } /* end cell */
  …
} /* end library */
```

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

```
library(name_id) {...lu_table_template(template_name_id) {variable_1:
input_net_transition; index_1 ("float,..., float");...}...

}
```

Pin-Level Template Variables

In the `pin` group, the table template specifying composite current source receiver models can have one variable: `variable_1`. The only valid value is `input_net_transition`.

The index values in the `index_1` attribute are a list of ascending floating-point numbers.

Pin-Level lu_table_template Example

```
...
   lu_table_template(LTT1) {
      variable_1: input_net_transition;
      index_1("0.1, 0.2, 0.3, 0.4");
   }
```

Defining the Lookup Table receiver_capacitance Group

To specify the receiver capacitance for the nonlinear table model, use the `receiver_capacitance` group within the `pin` group.

Pin-Level receiver_capacitance Group Syntax

```
pin(name_id) {direction: input; /* or "inout" */receiver_capacitance() {
receiver_capacitance1_rise(template_name_id) {index_1("float,..., float");
/* optional */values("float,..., float");

      }     receiver_capacitance1_fall(template_name_id) {
index_1("float,..., float"); /* optional */values("float,..., float");}

      receiver_capacitance2_rise(template_name_id)index_1("float,...,
float"); /* optional */values("float,..., float");}

      receiver_capacitance2_fall(template_name_id) {index_1("float,...,
float"); /* optional */values("float,..., float"); } }}
```

Pin-Level Template Variables

In the `pin` group, the table template specifying receiver characteristics can have one variable: `variable_1`. The only valid value is `input_net_transition`.

The index values in the `index_1` attribute are a list of ascending floating-point numbers.

The `values` attribute defines a list of floating-point numbers that represent the capacitance of the receiver model.

Pin-Level receiver_capacitance Example

```
pin(A) { /* pin-based receiver model*/

   direction : input;
   receiver_capacitance() {
      receiver_capacitance1_rise(LTT1) {
         values("1.0, 4.1, 2.1, 3.0");
      }
      receiver_capacitance1_fall(LTT1) {
         values("1.0, 3.2, 2.1, 4.0");
```

```
      }
      receiver_capacitance2_rise(LTT1) {
         values("1.0, 4.1, 2.1, 3.0");
      }
      receiver_capacitance2_fall(LTT1) {
         values("1.0, 3.2, 2.1, 4.0");
      }
   }
}
```

Checking the receiver_capacitance Group

Library Compiler automatically checks the `receiver_capacitance` group and reports any problems it encounters. For information about the types of checks Library Compiler performs, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Defining the Receiver Capacitance Groups at the Timing Level

At the timing level, you do not need to define the `receiver_capacitance` group. Define the receiver capacitance for the timing arcs by using only the `receiver_capacitance1_rise`, `receiver_capacitance1_fall`, `receiver_capacitance2_rise, and receiver_capacitance2_fall` groups.

Defining the lu_table_template Group

Use this library-level group to create templates of common information that multiple lookup tables can use. A table template specifies the table parameters and the breakpoints for each axis. Assign each template a name so that lookup tables can refer to it.

lu_table_template Group 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.

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");
   }
```

Defining the Lookup Table receiver_capacitance Groups

To specify the receiver capacitance for the nonlinear table model, use the `receiver_capacitance1_rise, receiver_capacitance1_fall, receiver_capacitance2_rise receiver_capacitance2_fall` groups within the `timing` group.

Timing-Level receiver_capacitance Group Syntax

```
direction: output; /* or "inout" */timing () {...
   receiver_capacitance1_rise(template_name_id) {index_1("float,...,
float"); /* optional */index_2("float,..., float"); /* optional */
values("float,..., float");}

   receiver_capacitance1_fall(template_name_id) {index_1("float,...,
float"); /* optional */index_2("float,..., float"); /* optional */
values("float,..., float");}

   receiver_capacitance2_rise(template_name_id) {

      index_1("float,..., float"); /* optional */index_2("float,...,
float"); /* optional */values("float,..., float");}

   receiver_capacitance2_fall(template_name_id) {index_1("float,...,
float"); /* optional */index_2("float,..., float"); /* optional */
values("float,..., float");}...}
```

Template Variables for CCS Receiver Models

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.

Timing-Level receiver_capacitance Example

```
timing() { /* timing arc-based receiver model*/
   ...
   related_pin : "B"
   receiver_capacitance1_rise(LTT2) {
      values("1.1, 4., 2.0, 3.2");
   }
   receiver_capacitance1_fall(LTT2) {
      values("1.0, 3.2, 4.0, 2.1");
   }
   receiver_capacitance2_rise(LTT2) {
      values("1.1, 4., 2.0, 3.2");
   }
   receiver_capacitance2_fall(LTT2) {
      values("1.0, 3.2, 4.0, 2.1");
   }
   ...
}
```

Checking the Timing-Level receiver_capacitance Group

Library Compiler automatically checks the `receiver_capacitance` group at the timing level and reports any problems it encounters. For information about the types of checks Library Compiler performs, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Composite Current Source Driver and Receiver Model Example

Example 5-1 is an example of composite current source driver and receiver model syntax.

Example 5-1 Composite Current Source Driver and Receiver Model

```
library(new_lib) {
   . . .
   output_current_template(CCT) {
      variable_1: input_net_transition;
      variable_2: total_output_net_capacitance;
      variable_3: time;
   }
   lu_table_template(LTT1) {
      variable_1: input_net_transition;
      index_1("0.1, 0.2, 0.3, 0.4");
   }
   lu_table_template(LTT2) {
      variable_1: input_net_transition;
      variable_2: total_output_net_capacitance;
```

```
      index_1("1.1, 2.2");
      index_2("1.0, 2.0");
   }
   . . .
   cell(DFF) {
      pin(D) { /* pin-based receiver model*/
         direction : input;
         receiver_capacitance() {
            receiver_capacitance1_rise(LTT1) {
               values("1.1, 0.2, 1.3, 0.4");
            }
            receiver_capacitance1_fall(LTT1) {
               values("1.0, 2.1, 1.3, 1.2");
            }
            receiver_capacitance2_rise(LTT1) {
               values("0.1, 1.2, 0.4, 1.3");
            }
            receiver_capacitance2_fall(LTT1) {
               values("1.4, 2.3, 1.2, 1.1");
         }
      }/*end of pin (D)*/
   } /*end of cell (DFF)*/
   ...
   cell() {
      ...
      pin (Y) {
         direction : output;
         capacitance : 1.2;

         timing() { /* CCS and arc-based receiver model*/

            . . .
            related_pin : "B";
            receiver_capacitance1_rise(LTT2) {
               values("0.1, 1.2”);
               values(“3.0, 2.3");
            }
            receiver_capacitance1_fall(LTT2) {
               values("1.1, 2.3”);
               values(“1.3, 0.4");
            }
            receiver_capacitance2_rise(LTT2) {
               values("1.3, 0.2”);
               values(“1.3, 0.4");
            }
            receiver_capacitance2_fall(LTT2) {
               values("1.3, 2.1”);
               values(“0.4, 1.3");
            }
            output_current_rise() {
               vector(CCT) {
                  reference_time : 0.05;
                  index_1(0.1);
```

```
            index_2(1.0);
            index_3("1.0, 1.5, 2.0, 2.5, 3.0");
            values("1.1, 1.2, 1.5, 1.3, 0.5");
         }
         vector(CCT) {
            reference_time : 0.05;
            index_1(0.1);
            index_2(2.0);
            index_3("1.2, 2.2, 3.2, 4.2, 5.2");
            values("1.11, 1.31, 1.51, 1.41, 0.51");
         }
         vector(CCT) {
            reference_time : 0.06;
            index_1(0.2);
            index_2(1.0);
            index_3("1.2, 2.1, 3.2, 4.2, 5.2");
            values("1.0, 1.5, 2.0, 1.2, 0.4");
         }
         vector(CCT) {
            reference_time : 0.06;
            index_1(0.2);
            index_2(2.0);
            index_3("1.2, 2.2, 3.2, 4.2, 5.2");
            values("1.11, 1.21, 1.51, 1.41, 0.31");
         }
      }
      output_current_fall() {
         vector(CCT) {
            reference_time : 0.05;
            index_1(0.1);
            index_2(1.0);
            index_3("0.1, 2.3, 3.3, 4.4, 5.0");
            values("-1.1, -1.3, -1.6, -1.4, -0.5");
      }
         vector(CCT) {
            reference_time : 0.05;
            index_1(0.1);
            index_2(2.0);
            index_3("1.2, 2.2, 3.2, 4.2, 5.2");
            values("1.11, -1.21, -1.41, -1.31, -0.51");
      }
         vector(CCT) {
            reference_time : 0.13;
            index_1(0.2);
            index_2(1.0);
            index_3("0.1, 1.3, 2.3, 3.4, 5.0");
            values("-1.1, -1.3, -1.8, -1.4, -0.5");
      }
         vector(CCT) {
            reference_time : 0.13;
            index_1(0.2);
            index_2(2.0);
            index_3("1.2, 2.2, 3.2, 4.2, 5.2");
```

```
                values("-1.11, -1.31, -1.81, -1.51, -0.41");
            }
          }/*end of timing*/
        . . .
        } /* end of pin (Y) */
        . . .
    } /* end of cell */
    . . .
} /* end of library */
```

Checking CCS Library Models

Library Compiler automatically checks CCS models, including the polarity of the current and the voltage swing, and reports any problems it encounters. For information about the types of checks Library Compiler performs for CCS models, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

6

Advanced Composite Current Source Modeling

This chapter provides an overview of advanced composite current source (CCS) modeling to support nanometer and very deep submicron IC development. The following composite current source modeling topics are covered:

- Modeling Cells With Advanced Composite Current Source Information
- Compact CCS Timing Model Support
- Variation-Aware Timing Modeling Support

Modeling Cells With Advanced Composite Current Source Information

Composite current source modeling supports additional driver model complexity by using a time- and voltage- dependent current source with essentially an infinite drive resistance. The new driver model achieves high accuracy by not modeling the transistor behavior. Instead, it maps the arbitrary transistor behavior for lumped loads to that for an arbitrary detailed parasitic network.

The composite current source model improves the receiver model accuracy because the input capacitance of a receiver is dynamically adjusted during the transition by using two capacitance values. The driver model can be used with or without the receiver model.

Compact CCS Timing Model Support

Existing CCS timing driver modeling syntax requires that you describe each CCS driver switching current waveform by adaptively sampling data points. Often, a large amount of data is required to represent the library to model these switching curves. As the number of timing arcs in a standard cell library grows, the CCS timing library size can become very large.

This section describes the syntax of a compact modeling format that uses indirectly shared base curves to model the shape of switching curves. By allowing each base curve to model multiple switching curves with similar shapes, the modeling efficiency is improved and the CCS timing library is efficiently compressed.

The topics in the following sections include:

- Using the `format_lib` command to convert conventional CCS modeling information into compact CCS data format.
- Describing CCS timing base curves.
- Describing the syntax of base curves and the compact CCS driver modeling format.
- Describing Library Compiler screener rules for reading compact CCS timing data into Library Compiler.

Converting CCS Data to Compact CCS Format

The `format_lib` command allows you to convert conventional CCS modeling information into compact CCS data format. The compact CCS format offers the same high accuracy as conventional CCS data but uses much less space in library data files.

To convert conventional CCS data to compact CCS data, set the `compact_ccs` option, the input library containing the original (not compacted) CCS data, and the new library to contain the compact CCS data in a command file. Both library files are in Liberty (.lib) format. For example,

myfile contents:

```
set compact_ccs true
set input_library xlib82.lib
set output_library xlib82c.lib
```

Next, run the `format_lib` command at the Library Compiler prompt:

```
lc_shell-xg-t> format_lib -f myfile
```

Important:

In order to run the `format_lib` command in lc_shell, you must set the path for the SYNOPSYS_NCX_ROOT variable in your .cshrc file and point it to the NCX installation directory location. Otherwise, Library Compiler issues an error when you run `format_lib`. For example,

```
setenv SYNOPSYS_NCX_ROOT path_to_ncx_installation_directory
```

To view a list of the formatting options from Library Compiler, use the `-help` option:

```
lc_shell-xg-t> format_lib -help
```

The `-expand` option performs the opposite function; it expands a library containing compact CCS modeling information into a library containing the original (not compacted) CCS modeling information.

For more information about the various types of library formatting operations Library Compiler can perform, see the “Model Adaptation System” chapter in the *Library Compiler Methodology and Modeling Functionality in Technology Libraries User Guide*.

Modeling With CCS Timing Base Curves

CCS driver switching curves in the I-V domain are smoother than those in the I(t) and V(t) domains. The I-V switching curves are usually convex, and they have no inflection point in the middle, a feature that facilitates compact modeling.

Figure 6-1 illustrates modeling an inverter cell rise transition with the existing CCS format. The figure shows the I(t) curve, corresponding V(t) curve, and I-V curves.

The CCS segmentation process adaptively samples nine data points from the I(t) curve based on the given tolerance. There are 18 floating-point numbers (nine time points + nine current points) that are stored in the CCS library.

Figure 6-1 Inverter Cell Rise Transition With Existing CCS Format

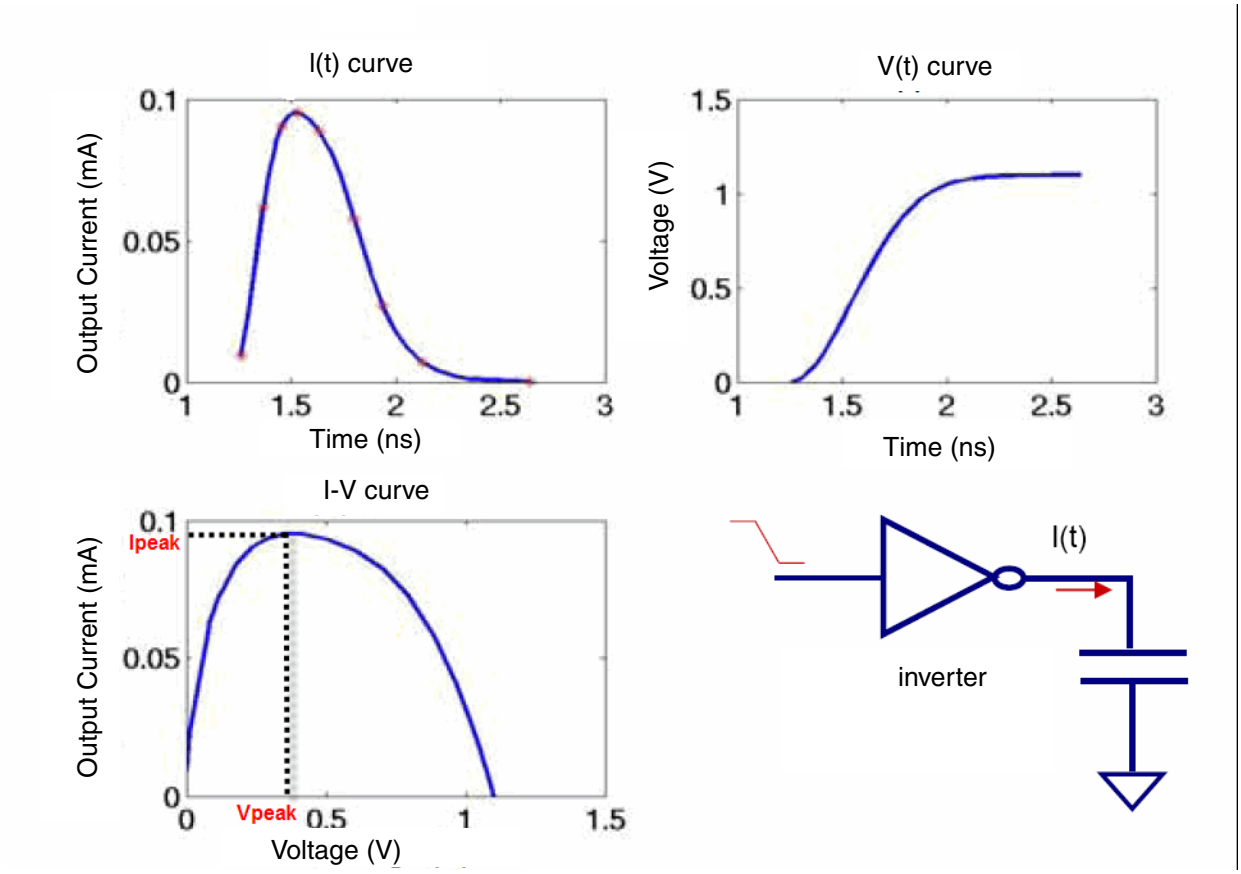


Figure 6-2 shows the mapping of I-V curves to base curves. The I-V curve is split into two halves at the peak, and an eleven point normalized base curve in the base curve database is selected to exactly match each half curve.

Only six parameters are required to model this inverter switching curve, reducing the storage cost by three times. The six parameters are as follows:

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.

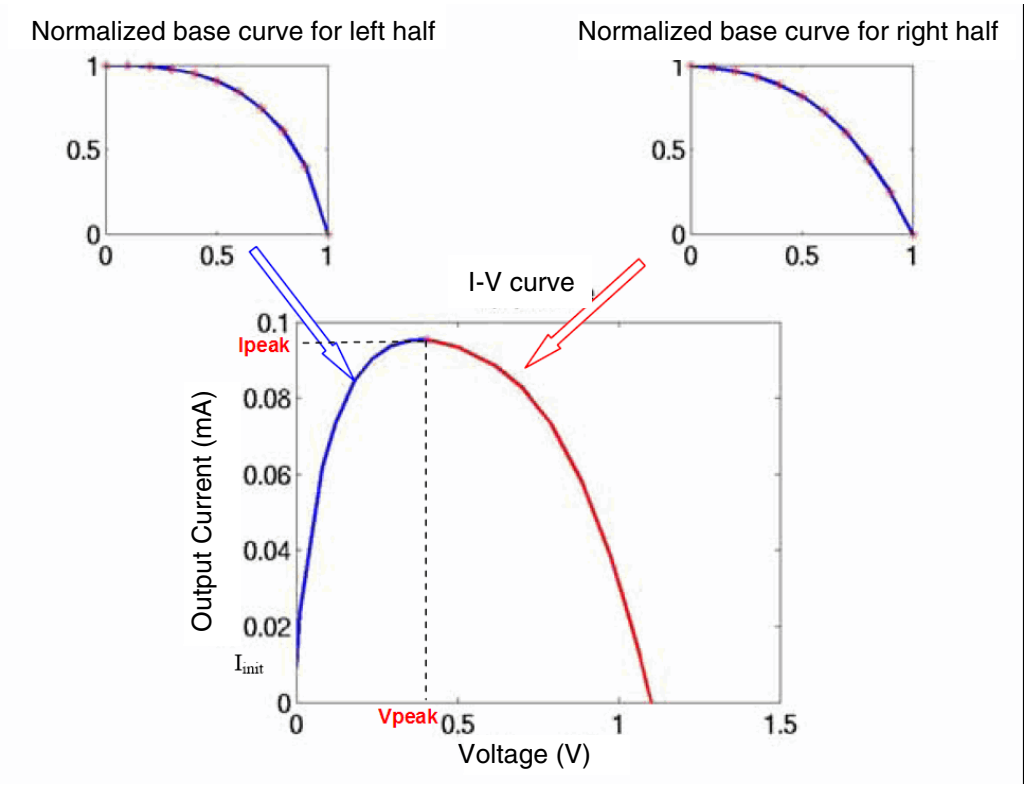
Left id

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 6-2 Using Base Curves to Simplify I-V Curve Modeling



Compact CCS Timing Model Syntax

In Figure 6-2, each switching I-V curve can be modeled by normalized base curves using the following parameters: `init_current`, `peak_current`, `peak_voltage`, `peak_time`, `left_id`, and `right_id`.

The `init_current`, `peak_current`, `peak_voltage`, and `peak_time` parameters are the critical characteristics of the curve. The `left_id` and `right_id` describe the two base curves that represent the two halves of the I-V curve.

The syntax for compact CCS timing model is as follows:

```
library(my_compact_ccs_lib) {
…
base_curves (base_curves_name) {
```

```
  base_curve_type: enum (ccs_timing_half_curve);
  curve_x ("float…, float");
  curve_y (curve_id, "float…, float");
  curve_y (curve_id, "float…, float");
  …
  curve_y (curve_id, "float…, float");
}

compact_lut_template(template_name) {
  base_curves_group: base_curves_name;
  variable_1 : input_net_transition |
total_output_net_capacitance;
  variable_2 : input_net_transition |
total_output_net_capacitance;
  variable_3 : curve_parameters;
  index_1 ("float…, float");
  index_2 ("float…, float");
  index_3 ("string…, string");
}
…

cell(cell_name) {
…
pin(pin_name) {
direction : string;
capacitance : float;
timing() {
/*Compact CCS arc-based driver model*/
compact_ccs_rise (template_name) {
  base_curves_group : "base_curves_name";
  values ("…"…, "…" \
  "…"…,  "…" \
  "…"…,  "…" \
  "…"…, "…")
}/*end of compact_ccs_rise() */
compact_ccs_fall(template_name) {
  …
}/*end of compact_ccs_fall() */
…
} /*end of timing */
} /*end of pin */
} /*end of cell */
…
} /* end of library*/
```

The groups described in the following sections support compact CCS timing models.

base_curves Group

The `base_curves` library-level group contains the detailed description of normalized base curves. The `base_curves` group has the following attributes:

base_curve_type Complex Attribute

The `base_curve_type` attribute specifies the type of base curve. The only valid value for this attribute is `ccs_timing_half_curve`.

curve_x complex Attribute

The data array contains the X-axis values of the normalized base curve. Only one `curve_x` is allowed for each `base_curves` group. See Figure 6-2 for more details.

For a `ccs_timing_half_curve` type base curve, the `curve_x` value must be between 0 and 1 and increase monotonically.

curve_y complex Attribute

Each base curve (as illustrated in Figure 6-2) is composed of one `curve_x` and one `curve_y`. You should define the `curve_x` base curve before `curve_y` for better readability and easier implementation.

There are two data sections in 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 valid string value for this variable.

index_1 and index_2 Attributes

These are required attributes. The define sample `input_net_transition` or `total_output_net_capacitance` values using the float notation.

index_3 Attribute

String values in `index_3` are determined by the `base_curve_type` in `base_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, Library Compiler issues a compilation error.

Library Compiler allows more than six parameters for cases that require more parameters to describe the original data (for example, Vddcell and Vsscell, or the final voltage of load capacitor in cell rise or fall transition).

compact_ccs_{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. Library Compiler checks the data specified in values according to the string order of `index_3`.

For compact CCS timing, base curves data `left_id` and `right_id` values can only be integers. If more than, six parameters are specified in `index_3`, Library Compiler does not check for other data in the values group. These six parameters are stored in the database as is.

Checking the compact_lut_template Group

Library Compiler automatically checks the `compact_lut_template` group and issues a compilation error message if it encounters problems. For information about the types of checks Library Compiler performs for the `compact_lut_template` group, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

CCS Timing Library Example

The following example shows the CCS timing library with the compact syntax:

```
library(my_lib) {
…
/* normal lu table template for timing arcs */
lu_table_template (LTT2) {
  variable_1 : input_net_transition;
  variable_2 : total_output_net_capacitance;
  index_1 ("0.1, 0.2");
  index_2 ("1.0, 2.0");
}
base_curves (ctbct1){
  base_curve_type : ccs_timing_half_curve;
  curve_x("0.2, 0.5, 0.8");
  curve_y(1, "0.8, 0.5, 0.2");
  curve_y(2, "0.75, 0.5, 0.35");
  …
  curve_y(100, "0.85, 0.5, 0.15");
}
…
/* New lu table template for compact CCS timing model*/
compact_lut_template(LTT3) {
  variable_1 : input_net_transition;
  variable_2 : total_output_net_capacitance;
  variable_3 : curve_parameters;
  index_1 ("0.1, 0.2");
  index_2 ("1.0, 2.0");
  index_3 ("init_current, peak_current, peak_voltage, peak_time, left_id,
right_id");
  base_curves_group: "ctbct1";
}
…

cell(cell_name) {
…
pin(Y) {
direction : output;
capacitance : 1.2;
timing() {
/*compact CCS and arc-based receiver model*/
compact_ccs_rise(LTT3) {
    base_curves_group : "ctbct1"; /* optional*/
    values("0.1, 0.5, 0.6, 0.8, 1, 3", \
   "0.15, 0.55, 0.65, 0.85, 2, 4", \
   "0.2, 0.6, 0.7, 0.9, 3, 2", \
   "0.25, 0.65, 0.75, 0.95, 4, 1")
}/*end of compact_ccs_rise() */
compact_ccs_fall(LTT3) {
    . . .
}/*end of compact_ccs_fall() */
. . .
```

```
}/*end of timing */
}/*end of pin(Y) */
}/*end of cell */
…
}/* end of library*/
```

Variation-Aware Timing Modeling Support

As process technologies scale to nanometer geometries, it is crucial to build variation-based cell models to solve uncertainties attributed to the variability in the device and interconnect. The CCS timing approach addresses the effects of nanometer processes by enabling advanced driver and receiver modeling.

This modeling capability supports variation parameters and is an extension of compact CCS timing driver modeling. You can even apply variation parameter models to CCS timing models. For more information about compact CCS timing driver modeling, see “Compact CCS Timing Model Support” on page 6-2.

These timing models employ a single current-based behavior that enables the concurrent analysis and optimization of timing issues. The result is a complete open-source current based modeling solution that reduces design margins and speeds design closure.

Process variation is modeled in static timing analysis tools to improve parametric yield and to control the design for corner-based analysis. This section describes the extension for variation-aware timing modeling using the existing CCS syntax.

The following 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 va_parameters is specified at the library level, all cells under the library default to the same variation parameters.
- If va_parameters is defined in the variation group, all va_values and nominal_va_values under the same variation group shall refer to this va_parameters.

The attribute values can be user-defined or predefined parameters. For more information, see Example 6-1 on page 6-28.

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” on page 2-7.

You can use the following predefined parameters:

- For the parameters defined in operating_conditions, if voltage_map is defined, and you specify these attributes as values of va_parameters, Library Compiler takes these attributes as user-defined variables.
- For the parameters defined in operating_conditions, if there is no voltage_map attribute at library level, and you specify these attributes as values of va_parameters, Library Compiler takes these attributes as predefined parameters.

nominal_va_values Attribute

This attribute characterizes nominal values of all variation parameters.

- It is required for every timing_based_variation group.
- The value of this attribute has a 1-to-1 mapping to the corresponding va_parameters.
- If a nominal compact CCS driver model group and a variation-aware compact CCS driver model group are defined under the same timing group, the nominal values are applied to the nominal compact CCS driver model group.

va_compact_ccs_rise and va_compact_ccs_fall Groups

The `va_compact_ccs_rise` and `va_compact_ccs_fall` groups specify characterization corners with variation value parameters. All screener checks applicable to the compact CCS format apply to attributes under this group.

- These groups can be specified under different `timing_based_variation` groups if they cannot share the same `va_parameters`.
- You should characterize two corners at each side of the nominal value of all variation parameters as specified in `va_parameters`. When corners are characterized for one of the parameters, all other variations are assumed to be nominal values. Therefore, a `timing_based_variation` group with *N* variation parameters requires exactly 2*N* characterization corners. For an example, see Example 6-2 on page 6-29.
- All variation-aware compact CCS driver model groups inside the `timing_based_variation` share the same `va_parameters` attribute.

va_values Attribute

The `va_values` attribute specifies values of each variation parameter for all corners characterized in the variation-aware compact CCS driver model groups.

- Required for the variation-aware compact CCS driver model groups.
- The value of this attribute has a 1-to-1 mapping to the corresponding `va_parameters`.

For an example that shows how to specify `va_values` with three variation parameters, see Example 6-3 on page 6-29. 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.

Checking Variation-Aware Compact CCS Timing Driver Models

Library Compiler automatically checks variation-aware compact CCS timing driver models and reports any problems it encounters. For information about the types of checks Library Compiler performs for variation-aware compact CCS timing driver models, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

timing_based_variation and pin_based_variation Groups

These groups represent variation-aware receiver capacitance information under the timing and pin groups.

- If `receiver_capacitance` group exists in pin group, variation-aware CCS receiver model groups are required in `pin_based_variation`.
- If nominal CCS receiver model groups exist in the timing group, variation-aware CCS receiver model groups are required in `timing_based_variation`.

va_parameters Attribute

The `va_parameters` attribute specifies a list of variation parameters within a `timing_based_variation` or `pin_based_variation` group. See “va_parameters Attribute” on page 6-11 for details.

nominal_va_values Attribute

The `nominal_va_values` attribute characterizes nominal values for all variation parameters. The following list describes the `nominal_va_values` attribute.

- The attribute is required in the `timing_based_variation` and `pin_based_variation` groups.
- The value of this attribute has one-to-one mapping to the corresponding `va_parameters` attribute.
- In pin-based models, if nominal CCS receiver model and variation-aware CCS receiver model groups are defined under the same pin, the nominal values are applied to the nominal CCS receiver model groups. For an example, see Example 6-5 on page 6-31.
- In timing-based models, if the nominal compact CCS driver model group and variation-aware CCS receiver model groups are defined under the same timing group, the nominal values are applied to the nominal compact CCS driver model groups.

va_receiver_capacitance1_rise, va_receiver_capacitance1_fall, va_receiver_capacitance2_rise, va_receiver_capacitance2_fall Groups

These groups specify characterization corners with variation values in `timing_based_variation` and `pin_based_variation` groups. All attributes under these groups are checked with the same library screener as nominal CCS receiver models.

- You should characterize two corners at each side of the nominal value of all variation parameters specified in `va_parameters`.

  When corners are characterized for one of parameters, all other variations are assumed to be nominal value. Therefore, for a `timing_based_variation` group with *N* variation parameters, exactly 2*N* characterization corner groups are required. This same rule applies to `pin_based_variation`.

- All variation-aware CCS receiver model groups in `timing_based_variation` or `pin_based_variation` group share the same `va_parameters attribute`.

va_values Attribute

Specifies values of each variation parameter for all corners characterized in variation-aware CCS receiver model groups.

- The attribute is required for variation-aware CCS receiver model groups.
- The value of this attribute has 1-to-1 mapping to the corresponding `va_parameters` attribute.

Variation-Aware Syntax of the CCS Timing in Receiver Modeling Checks

Library Compiler checks the following variation aware syntax of the CCS timing in receiver modeling, and issues an error if any of the following are not satisfied:

- The `va_values` is required in variation-aware CCS receiver model groups.
- The variation-aware CCS receiver model groups can be specified in a `timing_based_variation` group only if `the nominal CCS receiver model` groups are defined within the same timing group where the `timing_based_variation` group is defined.
- The variation-aware CCS receiver model groups can be specified in a `pin_based_variation` group only if `receiver_capacitance` group is defined within the same pin group as where the `pin_based_variation` group is defined.
- The variation-aware CCS receiver model group must refer to a valid `lu_table_template`.

Variation-Aware CCS Timing Receiver Model

The variation-aware CCS receiver model is expected to be used together with variation-aware compact CCS driver model. The `timing_based_variation` and `pin_based_variation` groups specify timing and pin groups respectively. In both groups, the variation-aware CCS receiver model groups are used to represent variation-aware CCS receiver information. They are defined in the syntax below and in the following 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>) {
       variable_1 : input_net_transition; …}
    va_parameters(<string> , …);
     …
       pin(<pin_name>) {
         receiver_capacitance() {
         receiver_capacitance1_rise(<template_name>) { …}
         receiver_capacitance2_rise(<template_name>) { …}
         receiver_capacitance1_fall(<template_name>) { …}
         receiver_capacitance2_fall(<template_name>) { … }
       }
       pin_based_variation() {
         va_parameters(<string> , … );
         nominal_va_values(<float>,…);
         va_receiver_capacitance1_rise(<pin_based_template_name>) {
           va_values(<float>, …);
           values("<float>, …", …);
         …}
         va_receiver_capacitance2_rise(<pin_based_template_name>) {…}
         va_receiver_capacitance1_fall(<pin_based_template_name>) {...}
         va_receiver_capacitance2_fall(<pin_based_template_name>) { …}
       …} /* end of pin_based_variation */
     …} /* end of pin */
  …
       pin(<pin_name>) {
       …
           timing() {
           receiver_capacitance1_rise(<template_name>) {…}
           receiver_capacitance2_rise(<template_name>) {…}
           receiver_capacitance1_fall(<template_name>) {…}
           receiver_capacitance2_fall(<template_name>) {…}
           timing_based_variation() {
             va_parameters(<string> , … );
             nominal_va_values(<float>,…);
             va_receiver_capacitance1_rise(<timing_based_template_name>)
{
               va_values(<float>, …);
               values("<float>, …", …);
```

```
                …}
                va_receiver_capacitance2_rise(<timing_based_template_name>)
{…}
                va_receiver_capacitance1_fall(<timing_based_template_name>)
{…}
                va_receiver_capacitance2_fall(<timing_based_template_name>)
{…}
          …} /* end of timing_based_variation */
       …} /*end of  timing */
    …} /* end of pin */
  …
```

timing_based_variation and pin_based_variation Groups

These groups represent variation-aware receiver capacitance information under the pin or timing group level.

- If the `receiver_capacitance` group exists in the pin group, variation-aware CCS receiver model groups are required in `pin_based_variation`.
- If nominal CCS receiver model groups exist in the timing group, variation-aware CCS receiver model groups are required in `timing_based_variation`.

va_parameters Complex Attribute

This attribute specifies a list of variation parameters within `timing_based_variation` or `pin_based_variation`. See “va_parameters Attribute” on page 6-11 for more details.

nominal_va_values Complex Attribute

This complex attribute specifies the nominal values of all variation parameters.

- The attribute is required for `timing_based_variation` and `pin_based_variation` groups.
- The value of this attribute has one-to-one mapping to the corresponding `va_parameters`.
- In a pin-based model, if nominal CCS receiver model and variation-aware CCS receiver model groups are defined under the same pin, the nominal values are applied to nominal CCS receiver model groups. For an example, see Example 6-5 on page 6-31.
- In a timing-based model, if nominal CCS receiver model and variation-aware CCS receiver model groups are defined under the same timing group, the nominal values are applied to nominal CCS receiver model groups.

va_receiver_capacitance1_rise, va_receiver_capacitance1_fall, va_receiver_capacitance2_rise, and va_receiver_capacitance2_fall Groups

These groups specify characterization corners with variation values in the `timing_based_variation` and `pin_based_variation` groups. The same library screening checks apply to all attributes under these groups as with the nominal CCS receiver model groups.

- You should characterize two corners at each side of the nominal value of all variation parameters specified in `va_parameters`.

  When corners are characterized for one of the parameters, all other variations are assuming to be nominal value. Therefore, for a `timing_based_variation` group with N variation parameters, exactly 2N characterization corner groups are required. This rule also applies to `pin_based_variation`.

- All variation-aware CCS receiver model groups in `timing_based_variation` or `pin_based_variation` group share the same `va_parameters`.

va_values Attribute

Specifies values of each variation parameter for all corners characterized in variation-aware CCS receiver model groups.

- Required for variation-aware CCS receiver model groups.
- The value of this attribute has 1-to-1 mapping to the corresponding `va_parameters`.

Checking Variation-Aware Receiver Models

Library Compiler automatically checks the syntax for variation-aware receiver models and reports any problems it encounters. For information about the types of checks Library Compiler performs for variation-aware receiver models, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

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 syntax below and in the following sections.

```
library() {
   …
```

```
lu_table_template(<template_name>) {
   variable_1 : <variables>;
   variable_2 : <variables>;
   variable_3 : <variables>; … }
va_parameters(<string> , …);
 …
  timing () {
   …
   interdependence_id :<integer>;
   rise_constraint(<template_name>) { … }
   fall_constraint(<template_name>) { … }
   timing_based_variation() {
      va_parameters(<string> , … );
      nominal_va_values(<float>,…);
      va_rise_constraint(<template_name>) {
         va_values(<float>,…);
         values("<float>,…");
       …}
      va_fall_constraint(<template_name>) { …}
    … }  /* end of timing_based_variation */
  … }  /* end of timing */
 …} /* end of pin */
…} /* end of library */
```

timing_based_variation Group

The `timing_based_variation` group specifies the rise and fall timing constraints for variation parameters within a timing group. The rise and fall timing constraints are specified in the `va_rise_constraint` and `va_fall_constraint` groups respectively.

- The `va_rise_constraint` group is required only if `rise_constraint` group exists within a timing group.
- The `va_fall_constraint` group is required only if `fall_constraint` group exists within a timing group.

va_parameters Complex Attribute

This complex attribute specifies a list of variation parameters within `timing_based_variation` or `pin_based_variation`. See "va_parameters Attribute" on page 6-11 for details.

nominal_va_values Complex Attribute

This complex attribute is used to specify nominal values of all variation parameters. See "nominal_va_values Attribute" on page 6-12 for more information.

va_rise_constraint and va_fall_constraint Groups

The `va_rise_constraint` and `va_fall_constraint` groups specify characterization corners with variation values in `timing_based_variation`. The screener checks all attributes under this group to normal timing constraint models.

- The template name refers to the `lu_table_template` group.
- Both groups can be specified under different `timing_based_variation` groups if they cannot share the same `va_parameters`.
- You are expected to characterize two corners at each side of the nominal value of all variation parameters as specified in `va_parameters`.

  When corners are characterized for one parameter, all other variations are assumed to be of nominal value. Therefore, for a `timing_based_variation` group with *N* variation parameters, exactly 2*N* characterization corners are required.
- All `va_rise_constraint` and `va_fall_constraint` groups in the `timing_based_variation` group share the same `va_parameters`.

va_values Attribute

Specifies values of each variation parameter for all corners characterized in `va_rise_constraint` and `va_fall_constraint` groups.

- Required for `va_rise_constraint` and `va_fall_constraint` groups.
- The value of this attribute has a one-to-one mapping to the corresponding `va_parameters`.

Checking Variation-Aware CCS Timing Constraint Models

Library Compiler automatically checks the syntax for variation-aware CCS timing constraint models and reports any problems it encounters. For information about the types of checks Library Compiler performs for variation-aware CCS timing constraint models, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Conditional Data Modeling for Variation-Aware Timing Receiver Models

Library Compiler 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");
        }
      }
      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);
      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");
    }
  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.

Library Compiler 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.

Checking Variation-Aware Timing Models

Library Compiler automatically checks the syntax for compact CCS timing driver, CCS timing receiver, and timing constraint models and reports any problems it encounters. For information about the types of checks Library Compiler performs for variation-aware receiver models, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Variation-Aware Syntax Examples

Example 6-1 va_parameters in Advanced CCS Modeling Usage

```
library (sample) {
  …
  operating_conditions (typical) {
    process : 1.5 ;
    temperature : 70 ;
    voltage : 2.75 ;
     …}
    default_operating_conditions: typical;
            …
    /* " temperature",  "voltage"  and "process" are predefined
parameters, and "Vthr" is an user-defined parameter. */

         va_parameters(temperature, voltage, process, Vthr, …);
       …}


library (sample) {
  …
```

```
  operating_conditions (typical) {
    process : 1.5 ;
    temperature : 70 ;
    voltage : 2.75 ;
    …}
    voltage_map(VDD1, 2.75);
    voltage_map(GND2, 0.2);
    default_operating_conditions: typical;
            …
    /* "VDD1" and "GND2" are predefined parameters, and LC takes
"voltage" as
         an user-defined  parameter. */

        va_parameters(VDD1, GND2, voltage,…);
```

For information about `va_parameters`, see "va_parameters Attribute" on page 6-11.

Example 6-2 va_compact_ccs_rise and va_compact_ccs_fall Groups

```
 …
timing() {
      …
    compact_ccs_rise(temp) { /* nominal I-V waveform */
                  …}
    timing_based_variation() {
    va_parameters(<string>, … ); /* N variation parameters */
      …
    va_compact_ccs_rise(temp) {  /* 1st corner */
                        …}
    …
    va_compact_ccs_rise(temp) {  /* last corner : total (2 * N) corners
*/ …}
   } /* end of timing_based_variation */ …} /* end of  timing */
…
```

For information about `va_compact_ccs_rise` and `va_compact_ccs_fall`, see "va_compact_ccs_rise and va_compact_ccs_fall Groups" on page 6-13.

Example 6-3 va_values With Three Variation Parameters

```
 …
timing_based_variation ( ) {
   va_parameters(var1, var2, var3); /* assumed that three variation
parameters
                                        are var1, var2 and var3 */
   nominal_va_values(0.5, 1.0, 2.0);
   va_compact_ccs_rise ( )  {
   va_values(0.50,  1.0,  1.8);  ...}
   va_compact_ccs_rise ( )  {
   va_values(0.50,  1.0,  2.2);  ...}
   va_compact_ccs_rise ( )  {
   va_values(0.50,  0.9,  2.0);  ...}
   va_compact_ccs_rise ( )  {
```

```
   va_values(0.50,  1.1,  2.0);  ...}
   va_compact_ccs_rise ( )  {
   va_values(0.45,  1.0,  2.0);  ...}
   va_compact_ccs_rise ( )  {
   va_values(0.55,  1.0,  2.0);  ...}
}
  …
```

For information about using `va_values` with the `va_compact_ccs_rise` and `va_compact_ccs_fall` groups, see "va_compact_ccs_rise and va_compact_ccs_fall Groups" on page 6-13.

Example 6-4 peak_voltage in Values Attribute

```
 …
library(va_ccs) {
  compact_lut_template(clt) {
    index_3 ("init_current, peak_current, peak_voltage, peak_time,
left_id,
right_id"); …}
  voltage_map(VDD1, 3.0);
  voltage_map(VDD2, 3.5);
  voltage_map(GND1, 0.5);
  voltage_map(GND2, 0.2);
   …
  cell(test) {
    pg_pin(v1) {
      voltage_name : VDD1;  …}
    pg_pin(v2) {
      voltage_name : VDD2;  …}
    pg_pin(g1) {
      voltage_name : GND1; … }
    pg_pin(g2) {
      voltage_name : GND2;  …}
       …
        timing() {
          timing_based_variation ( ) {
            va_parameters(Vthr);
            nominal_va_values(0.23);

            va_compact_ccs_rise (clt )  {
            /* error : There are two power pins (v1 and v2)
               and two ground pins (g1 and g2) in the cell "test".
               The peak_voltage cannot be greater than the largest power
               voltage, which is 3.5, and less than the smallest ground
               voltage,
               which is 0.2. The value 4.0 is greater than 3.5 and 0.1
               is less than 0.2.  Both of them are wrong. */

               va_values(0.25);
               values("0.21, 0.54, 4.0, 0.36, 1, 2",\
                      "0.15, 0.55, 0.1, 0.85, 2, 4",  … ); … }
             …
```

```
            timing_based_variation ( ) {
              va_parameters(Vthr, VDD2, GND2);
              nominal_va_values(0.23, 3.5, 0.2);
              va_compact_ccs_rise ( clt)  {
            /* In this group,  the variation value of VDD2 is 4.1,
               and GND2 is 0.0, so the largest power voltage is 4.1 and
               the smallest ground voltage is 0.0. The peak_voltage 4.0 is
               less than 4.1 and 0.1 is greater than 0.0, which is okay. */

                 va_values(0.18, 4.1, 0.0);
                 values("0.21, 0.54, 4.0, 0.36, 1, 2",\
                        "0.15, 0.55, 0.1, 0.85, 2, 4", … ); …}
                …
```

For information about `peak_voltage` in values attribute see "va_compact_ccs_rise and va_compact_ccs_fall Groups" on page 6-13.

Example 6-5 pin_based_variation Group

```
…
pin(pin_name) {
   receiver_capacitance() {
      receiver_capacitance1_rise(template_name) {
      /* nominal input capacitance table */ …}
   …}
   pin_based_variation() {
      nominal_va_values(2.0, 4.54, 0.23);
      /* These nominal values apply to nominal input capacitance tables.
*/
          va_receiver_capacitance1_rise(template_name) {
          /* variational input capacitance table */
   …}… }… } /* end of pin */
…
```

For information on `peak_voltage` in values attribute see "timing_based_variation and pin_based_variation Groups" on page 6-17.

Example 6-6 pin-based Model With nominal CCS receiver model and Variation-aware CCS Receiver Model Groups

```
/* Assume that there is no va_parameters defined */
   library(lib_name) {
     …
   timing() {
      timing_based_variation() {
         va_compact_ccs_rise(cltdf)  {
            base_curves_group : base_name;
            va_values(2.4)
            /* error : can't find a corresponding va_parameters */ …}
     …}  /* end of timing_based_variation */
… }  /* end of library */

/* Assume that va_parameters is defined at the end of
timing_based_variation
```

```
group and no default va_parameters is defined at library level */
…
      timing_based_variation() {
      nominal_va_values(2.0);
      /* error : can't find a corresponding va_parameters  */
      …
   va_parameters(Vthr);
   /* within a timing_based_variation, this shall be defined before
      all nominal_va_values and va_values attributes */
      }  /* end of timing_based_variation */
  ...

/* Assume that va_parameters is defined only at library level */
   …
library(lib_name) {
…
      timing_based_variation() {
         nominal_va_values(2.0);
         /* error : can't find a corresponding va_parameters  */  …}
  ...
         va_parameters(Vthr);
         /* within a library, this shall be defined before all
            cell groups (or all nominal_va_values and va_values
attributes) */
} /* end of library */
```

For information on `peak_voltage` in values attribute see "timing_based_variation Group" on page 6-11.

Example 6-7 nominal_va_values in Advanced CCS Modeling Usage

```
/* ASSUME that there is no voltage_map defined in library */

library (sample) {
   operating_conditions (typical) {
      process : 1.5 ;
      temperature : 70 ;
      voltage : 2.75 ; …}
   default_operating_conditions: typical;
    …
      timing_based_variation() {
         va_parameters(voltage, temperature, process);
         nominal_va_values(2.00, 70, 1.5);
         /* error : The nominal voltage defined in
   default_operating_conditions is 2.75. The value 2.00 is wrong. */


/* There is voltage_map defined at library level. */
library (sample) {
   operating_conditions (typical) {
      process : 1.5 ;
      temperature : 70 ;
      voltage : 2.75 ; …}
```

```
   voltage_map(VDD1, 2.75);
   default_operating_conditions: typical;
    …
      timing_based_variation() {
         va_parameters(voltage);
         nominal_va_values(2.00);
       /* This is okay because "voltage" is an user-defined parameter. */
   …
```

Example 6-8 Variational Values in Advanced CCS Modeling Usage

```
/* When var2 is in nominal value (1.0), var1 has three variational values
(0.45, 0.55 and 0.50), which is wrong because only two are allowed.
When var1 is in nominal value (0.50), var2 has two variational values
(0.8 and 1,0). 0.8 is less than nominal value (1.0), which is okay,
however, 1.0 is not greater than nominal value, which is wrong.   */
…
timing_based_variation ( ) {
   va_parameters(var1,  var2);
   nominal_va_values(0.50, 1.0);
   va_receiver_capacitance2_rise (temp_1)  {
      va_values(0.45,  1.0);
      ...
      }
      va_receiver_capacitance2_rise (temp_1)  {
         va_values(0.55,  1.0);
         ...
      }
      va_receiver_capacitance2_rise (temp_1)  {
         va_values(0.50,  0.8);
         ...
      }
      va_receiver_capacitance2_rise (temp_1)  {
         va_values(0.50,  1.0);
         ...
      }
   }
…
```

Example 6-9 Variation-Aware CCS Driver or Receiver with Timing Constraints

```
library(my_lib) {
    ...
   base_curves (ctbct1){
      base_curve_type : ccs_timing_half_curve;
      curve_x("0.2, 0.5, 0.8");
      curve_y(1, "0.8, 0.5, 0.2");
      curve_y(2, "0.75, 0.5, 0.35") ;
      curve_y(3,  "0.7, 0.5, 0.45") ;
      ...…
      curve_y(37, "0.23, 1.4, 6.23") ;
   }
```

```
   compact_lut_template(LUT4x4) {
      variable_1 : input_net_transition;
      variable_2 : total_output_net_capacitance;
      variable_3 : curve_parameters;
      index_1("0.1, 0.2, 0.3, 0.4");
      index_2("1.0, 2.0, 3.0, 4.0");
      index_3 ("init_current, peak_current, peak_voltage, peak_time,
left_id,
right_id");
      base_curves_group: "ctbct1";
   }
   lu_table_template(LUT3) {
      variable_1: input_net_transition;
      index_1("0.1, 0.3, 0.5");
   }
   lu_table_template(LUT3x3) {
      variable_1: input_net_transition;
      variable_2: total_output_net_capacitance;
      index_1("0.1, 0.3, 0.5");
      index_2("1.0, 3.0, 5.0");
   }
   lu_table_template(LUT5x5) {
      variable_1: constrained_pin_transition;
      variable_2: related_pin_transition;
      index_1("0.01, 0.05, 0.1, 0.5, 1");
      index_2("0.01, 0.05, 0.1, 0.5, 1");
   }
...
   cell(INV1) {
   ...
   pin (A) {
      direction: input;
      capacitance: 0.3;
      receiver_capacitance ( ) {
      ...
   }
   pin_based_variation ( ) {
      va_parameters(channel_length,  threshold_voltage);
      nominal_va_values(0.5, 0.5) ;
      va_receiver_capacitance1_rise (LUT3) {
         va_values(0.50, 0.45);
         values("0.29, 0.30, 0.31");
      }
      va_receiver_capacitance1_rise (LUT3) {
         va_values(0.50, 0.55);
         ...
      }
      va_receiver_capacitance1_rise (LUT3) {
         va_values(0.45, 0.50);
         ...
      }
      va_receiver_capacitance1_rise (LUT3) {
         va_values(0.55, 0.50);
```

```
            ...
         }
         va_receiver_capacitance2_rise (LUT3) {
            va_values(0.50, 0.45);
            values("0.19, 8.60, 5.41");
         }
         va_receiver_capacitance2_rise (LUT3) {
            va_values(0.50, 0.55);
            ...
         }
         va_receiver_capacitance2_rise (LUT3) {
            va_values(0.45, 0.50);
            ...
         }
         va_receiver_capacitance2_rise (LUT3) {
            va_values(0.55, 0.50);
            ...
         }
         va_receiver_capacitance1_fall (LUT3) {
            va_values(0.50, 0.45);
            values("0.53, 2.16, 9.18");
         }
         va_receiver_capacitance1_fall (LUT3) {
            va_values(0.50, 0.55);
            ...
         }
         va_receiver_capacitance1_fall (LUT3) {
            va_values(0.45, 0.50);
            ...
         }
         va_receiver_capacitance1_fall (LUT3) {
            va_values(0.55, 0.50);
            ...
         }
         va_receiver_capacitance2_fall (LUT3) {
            va_values(0.50, 0.45);
            values("0.39, 0.98, 5.15");
         }
         va_receiver_capacitance2_fall (LUT3) {
            va_values(0.50, 0.55);
            ...
         }
         va_receiver_capacitance2_fall (LUT3) {
            va_values(0.45, 0.50);
            ...
         }
         va_receiver_capacitance2_fall (LUT3) {
            va_values(0.55, 0.50);
            ...
         }
      }
   }     /* end of pin */
   pin (Y) {
```

```
direction: output;
   timing ( ) {
      related_pin: "A";
      compact_ccs_rise(LUT4x4) {
      ...
      }
      compact_ccs_fall(LUT4x4) {
      ...
      }
      timing_based_variation() {
         va_parameters(channel_length,  threshold_voltage);
         nominal_va_values(0.50, 0.50);
         va_compact_ccs_rise (LUT4x4 ) { /* without optional fields */
            va_values(0.50, 0.45);
            values("0.1, 0.5, 0.6, 0.8, 1, 3", \
                   "0.15, 0.55, 0.65, 0.85, 2, 4", \
                   "0.2, 0.6, 0.7, 0.9, 3, 2", \
                   "0.1, 0.2, 0.3, 0.4, 1,3", \
                   "0.2, 0.3, 0.4, 0.5, 4,5", \
                   "0.3, 0.4, 0.5, 0.6, 2,4", \
                   "0.4, 0.5, 0.6, 0.7, 7,8", \
                   "0.5, 0.6, 0.7, 0.8, 10,4", \
                   "0.5, 0.6, 0.8, 0.9, 11, 2", \
                   "0.25, 0.55, 1.65, 1.85, 3, 4", \
                   "1.2, 1.6, 1.7, 1.9, 5, 2", \
                   "1.1, 2.2, 2.3, 0.4, 1,30", \
                   "1.2, 2.3, 1.4, 0.5, 17,5", \
                   "1.3, 2.4, 1.5, 0.6, 22,24", \
                   "1.4, 2.5, 1.6, 1.7, 17,18", \
                   "1.5, 2.6, 0.7, 0.8, 10,33");
                              }
         va_compact_ccs_rise (LUT4x4 ) {
            va_values(0.50, 0.55);
            ...
         }
         va_compact_ccs_rise (LUT4x4 ) {
            va_values(0.45, 0.50);
            ...
         }
         va_compact_ccs_rise (LUT4x4 ) {
            va_values(0.55, 0.50);
            ...
         }
         va_compact_ccs_fall (LUT4x4 ) { /* without optional fields */
            ...
         }
         ...
      }  /* end of timing_based_variation */
         ...
   } /* end of timing */
      ...
   } /* end of pin */
   ...
```

```
} /* end of cell */
   ...

cell(INV4) {
   ...
   pin (Y) {
      direction: output;
         timing ( ) {
         related_pin: "A";
         receiver_capacitance1_rise (LUT3x3) {
         ...
         }
         receiver_capacitance2_rise (LUT3x3) {
         ...
         }
         receiver_capacitance1_fall (LUT3x3) {
         ...
         }
         receiver_capacitance2_fall (LUT3x3) {
      ...
         }
         rise_constraint(LUT5x5) {
      …
         }
         fall_constraint(LUT5x5) {
         …
         }
         timing_based_variation ( ) {
            va_parameters(channel_length,threshold_voltage);
            nominal_va_values(0.50, 0.50) ;
            va_receiver_capacitance1_rise (LUT3x3) {
               va_values(0.50, 0.45);
               values( "1.10, 1.20, 1.30", \
                       "1.11, 1.21, 1.31", \
                       "1.12, 1.22, 1.32");
         }
         va_receiver_capacitance2_rise (LUT3x3) {
            va_values(0.50, 0.45);
            values("1.20, 1.30, 1.40", \
                   "1.21, 1.31, 1.41", \
                   "1.22, 1.32, 1.42");
         }
         va_receiver_capacitance1_fall (LUT3x3) {
            va_values(0.50, 0.45);
            values("1.10, 1.20, 1.30", \
                   "1.11, 1.21, 1.31", \
                   "1.12, 1.22, 1.32");
         }
         va_receiver_capacitance2_fall (LUT3x3) {
            va_values(0.50, 0.45);
            values( 1.20, 1.30, 1.40", \
                   "1.21, 1.31, 1.41", \
                   "1.22, 1.32, 1.42");
```

```
}
va_receiver_capacitance1_rise (LUT3x3) {
   va_values(0.50, 0.55);
   ...
}
…
va_receiver_capacitance1_rise (LUT3x3) {
   va_values(0.45, 0.50);
   ...
}
                              …
va_receiver_capacitance1_rise (LUT3x3) {
   va_values(0.55, 0.50);
   ...
}
…
va_rise_constraint(LUT5x5) {
   va_values(0.50, 0.45);
   values( "-0.1452, -0.1452, -0.1452, -0.1452, 0.3329", \
           "-0.1452, -0.1452, -0.1452, -0.1452, 0.3952", \
           "-0.1245, -0.1452, -0.1452, -0.1358,  0.5142", \
           " 0.05829,  0.0216,  0.01068,  0.06927,  0.723", \
           " 1.263, 1.227, 1.223, 1.283, 1.963");
}
va_rise_constraint(LUT5x5) {
   va_values(0.50, 0.55);
   ...
}
va_rise_constraint(LUT5x5) {
   va_values(0.55, 0.50);
   ...
}
va_rise_constraint(LUT5x5) {
   va_values(0.45, 0.50);
   ...
}
va_fall_constraint(LUT5x5) {
   va_values(0.50, 0.55);
   ...
}
va_fall_constraint(LUT5x5) {
   va_values(0.55, 0.50);
   ...
}
va_fall_constraint(LUT5x5) {
   va_values(0.45, 0.50);
   ...
}
va_fall_constraint(LUT5x5) {
   va_values(0.50, 0.45);
   ...
}
} /* end of timing_based_variation */
```

```
            ...
        } /* end of timing */
...
    } /* end of pin */
      ...
  } /* end of cell */
   ...
} /* end of library */
```

7

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

A net can be either an aggressor or a victim:

- An aggressor net is a net that injects noise onto a victim net.
- A victim net is a net onto which noise is injected by one or more neighboring nets through the cross-coupling capacitors between the nets.

Noise effect can be categorized in two ways:

- Delay noise
- Functional noise

Delay noise occurs when victim and aggressor nets switch simultaneously. This activity alters the delay and slew of the victim net.

Functional noise occurs when a victim net is intended to be at a stable value and the noise injected onto this net causes it to glitch. The glitch might propagate to a state element, such as a latch, altering the circuit state and causing a functional failure.

For a gate-level static analysis tool to compute and detect any delay or functional noise failure, the following are calculated:

- Noise calculation
- Noise immunity
- Noise propagation

Noise Calculation

Coupled noise is the noise voltage induced at the output of nonswitching gates when coupled adjacent drivers to the output (aggressor drivers) are switching.

Noise Immunity

The main concept of noise immunity is that for most cells, a glitch on the input pin has to be greater than a certain fixed voltage to cause a failure. However, a glitch with a tall height might still not cause any failure if the glitch width is very small. This is mainly because noise failure is related to input noise glitch energy and this energy is proportional to the area under the glitch waveform.

For example, if a large voltage glitch in terms of height and width occurs on the clock pin of a flip-flop, the glitch can cause a change in the data and therefore the flip-flop output might change.

Noise Propagation

Propagated noise is the noise waveform created at the output of nonswitching gates due to the propagation of noise from the inputs of the same gate.

Modeling Cells for Noise

Library information for noise can be characterized in the following ways:

- I-V Characteristics and Drive Resistance
- Noise Immunity
- Noise Propagation

I-V Characteristics and Drive Resistance

To calculate the coupled noise glitch on a victim net, you need to know the effective steady-state drive resistance of the net. Figure 7-1 on page 7-4 shows the four different types of noise glitch:

- Vh+: Input is high, and the noise is over the high voltage rail.
- Vh-: Input is high, and the noise is below the high voltage rail.
- Vl+: Input is low, and the noise is over the low voltage rail.
- Vl-: Input is low, and the noise is below the low voltage rail.

Because the current is a nonlinear function of the voltage, you need to characterize the steady-state I-V characteristics curve, which provides a more accurate view of the behavior of a cell in its steady state. This information is specified for every timing arc of the cell that can propagate a transition. If an I-V curve cannot be obtained for a specific arc, the steady-state drive resistance single value can be used, but it is less accurate than the I-V curve.

Figure 7-1 Noise Glitch and Steady-State Drive Resistance

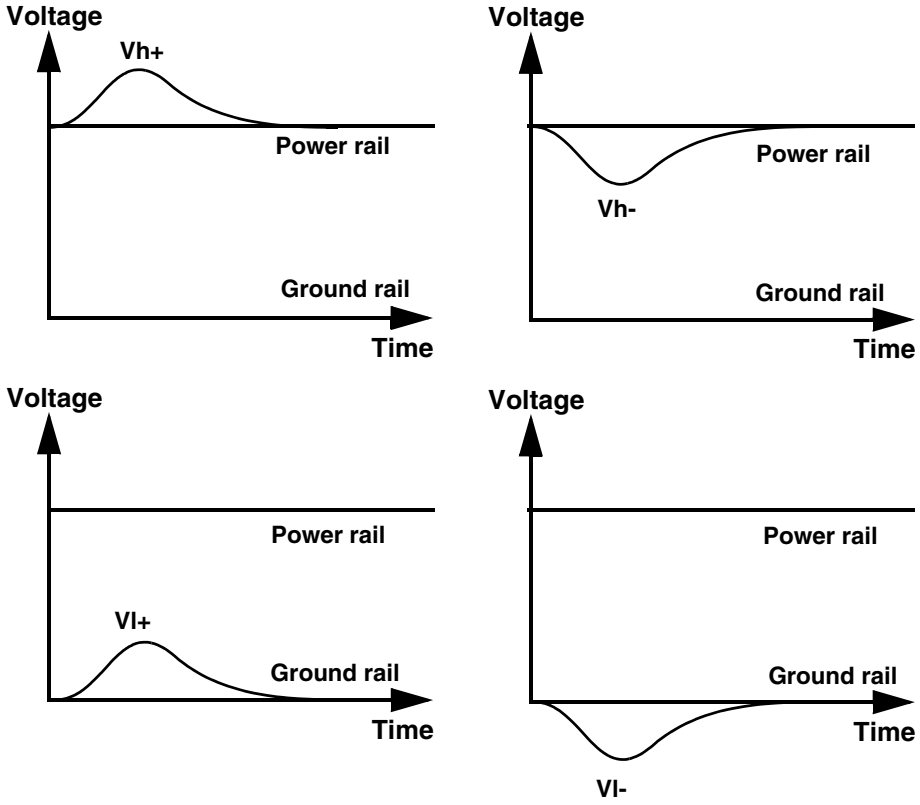
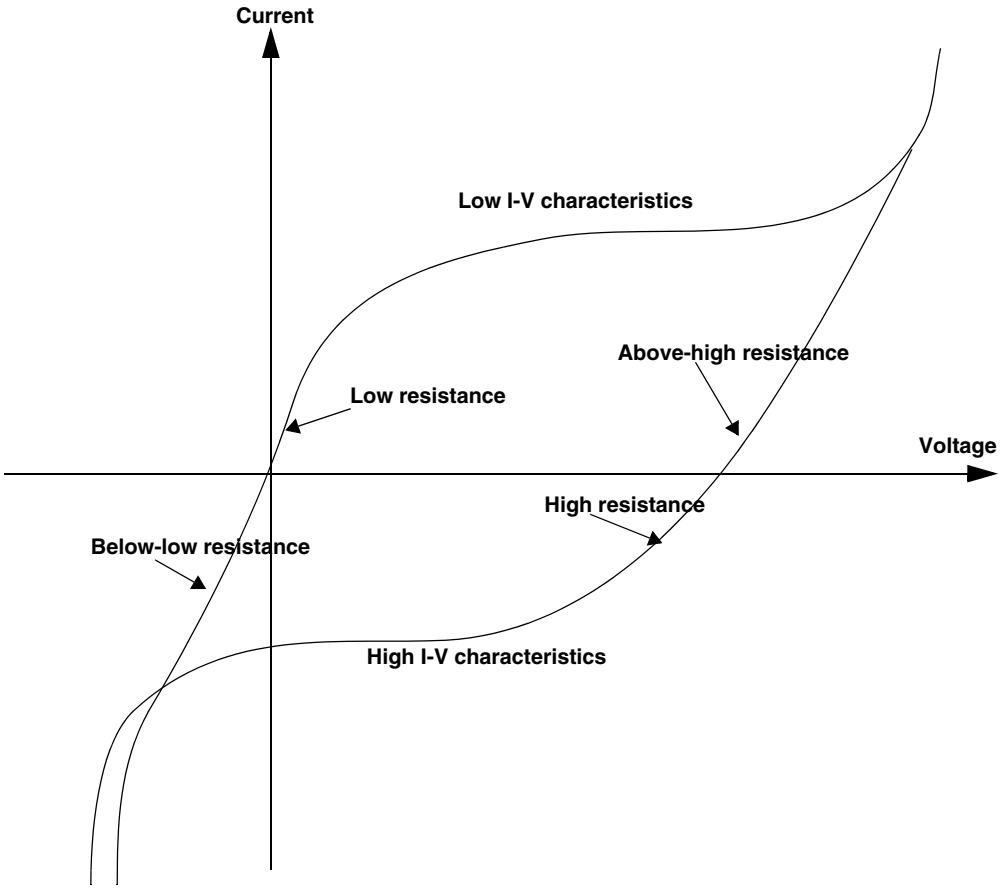


Figure 7-2 on page 7-5 is an example of two I-V curves and the steady-state resistance value.

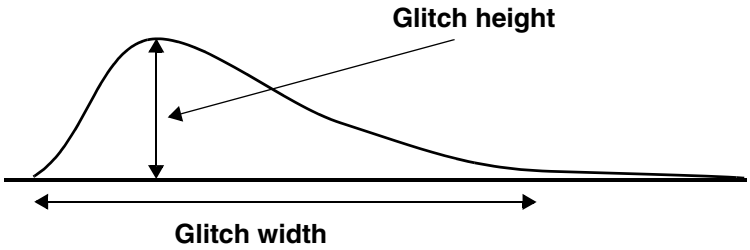
Figure 7-2 I-V Characteristics and Steady-State Drive Resistance



Noise Immunity

Circuits can tolerate large glitches at their inputs and still work correctly if the glitches deliver only a small energy. Given this concept, each cell input can be characterized by application of a wide range of coupling voltage waveform stimuli on it. Figure 7-3 shows a glitch noise model.

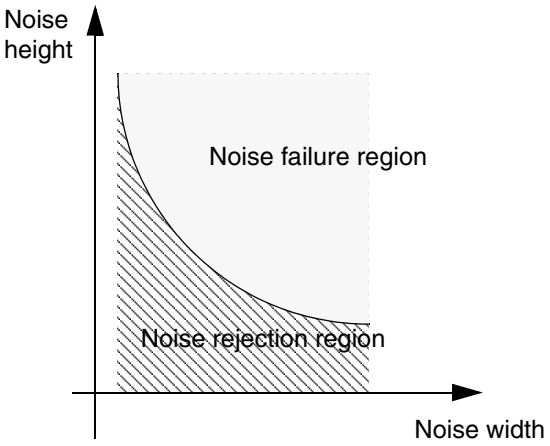
Figure 7-3 Glitch Noise Modeling



One method of modeling the noise immunity curve involves applying coupling voltage waveform stimuli with various heights (in library voltage units) and widths (in library time units) to the cell input, and then observing the output voltage waveform. The exact set of input stimuli (in terms of height and width) that produces an output noise voltage height equal to a predefined voltage is on the noise immunity curve. This predefined voltage is known as the *cell failure voltage*. Any input stimulus that has a height and width above the noise immunity curve causes a noise voltage higher than the cell failure voltage at the output and produces a functional failure in the cell.

Figure 7-4 shows an example of a noise immunity curve.

Figure 7-4 Noise Immunity Curve

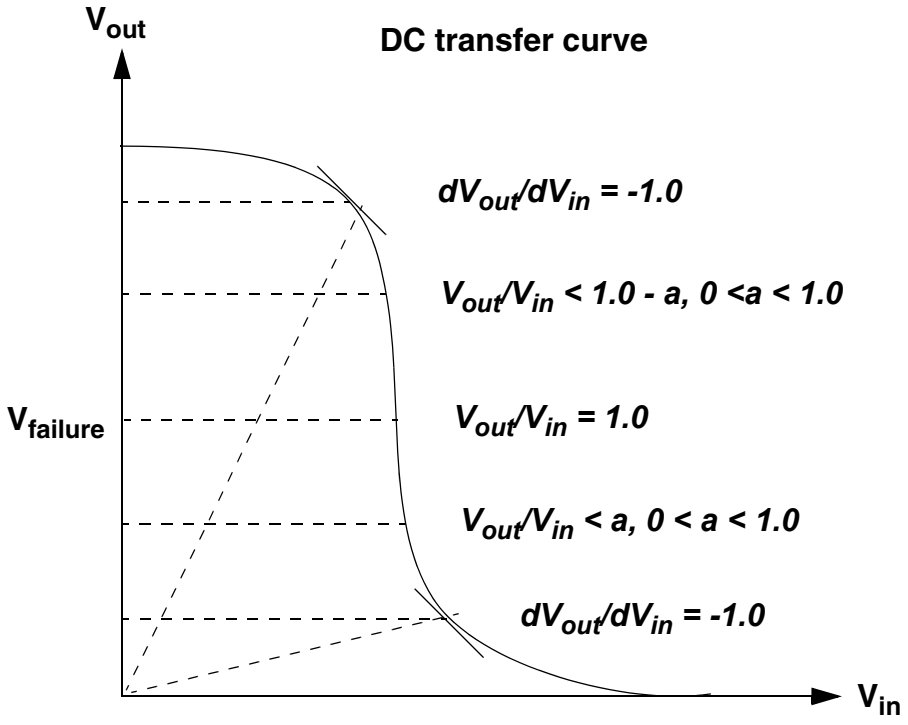


As shown in Figure 7-4, any noise width and height combination that falls above the noise rejection curve causes functional failure.

The selection of cell failure voltage is important for noise immunity curve characterization. There are many ways to select a failure voltage for a cell that produces usable noise immunity curves, including the following:

- $V_{failure}$ equal to the output DC noise margin
- $V_{failure}$ equal to the next cell's V_{IL} or ($V_{cc} – V_{IH}$)
- $V_{failure}$ corresponding to the point on the DC transfer curve where dV_{out}/dV_{in} is 1.0 or –1.0
- $V_{failure}$ corresponding to the point on the DC transfer curve where V_{out}/V_{in} is less than 1.0 or –1.0
- $V_{failure}$ corresponding to the point on the DC transfer curve where V_{out}/V_{in} is 1.0 or –1.0

Figure 7-5 Different Failure Voltage Criteria for Noise Immunity Curve



The noise immunity curve can also be a function of output loads, where cells with larger output loads can tolerate greater input noise.

The noise immunity curve is also state-dependent. For example, the noise on the A-to-Z arc of an XOR gate when B = 0 might be different from the B-to-Z arc when B = 1, because the arcs might go through different sets of transistors.

Using the Hyperbolic Model

The noise immunity curve resembles a hyperbola, because the area of different noise along the hyperbola is constant. Therefore noise immunity can be defined as a hyperbolic function with only three coefficients for every input on an I/O library pin. The formula for the height based on these three coefficients is as follows:

height = height_coefficient + area_coefficient / (width – width_coefficient);

Your tool (such as, PrimeTime *SI*) gets these coefficients from the library and applies the calculated height and width to determine whether the noise can cause functional failure. Any point above the hyperbolic curve signifies a functional failure.

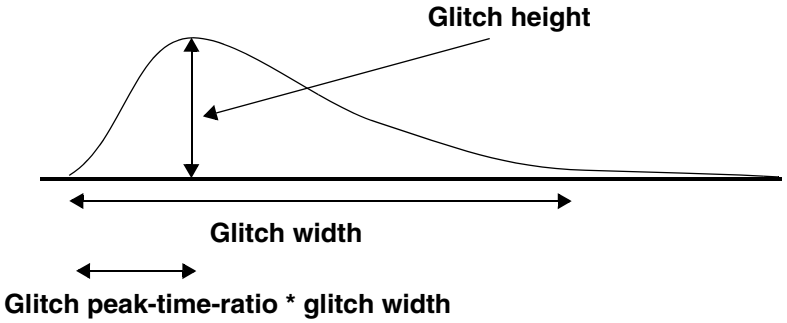
Noise Propagation

Propagated noise from the input to the output of a cell is modeled by

- Output glitch height
- Output glitch width
- Output glitch peak time ratio
- Output load

Figure 7-6 illustrates basic noise characteristics.

Figure 7-6 Basic Noise Characteristics



The output noise width, height, and peak-time ratio depend on the input noise width, height, and peak-time ratio as well as on the output load. However, in some cases, the dependency on peak-time ratio can be negligible; therefore, to reduce the amount of data, the lookup table does not have a peak-time-ratio dependency.

Table 7-1 shows a summary of the syntax used to model cases when the cell is not switching.

Table 7-1 Summary of Library Requirements for Noise Model

Category		Model type	Description
Noise detection	Voltage ranges (DC noise margin)	Lookup table and polynomial	input_voltage/output_voltage defined for all library pins
	Hyperbolic noise immunity curves	Lookup table and polynomial	Four hyperbolic curves; each has three coefficients, defined for input or bidirectional library pins
	Noise immunity tables	Lookup table	Four tables indexed by noise width and output load defined for timing arcs
	Noise immunity polynomials	Polynomial	Four polynomials as a function of noise width and output load defined for timing arcs
Noise calculation	Steady-state resistances	Lookup table and polynomial	Four floating-point values defined for timing arcs
	I-V characteristics tables	Lookup table	Two tables indexed by output steady-state voltage for non-three-state arcs and one table for three-state arcs
	I-V characteristics polynomials	Polynomial	Two polynomials as a function of output steady-state voltage for non-three-state arcs and one table for three-state arcs
Noise propagation	Noise propagation tables	Lookup table	Four pairs of noise width and height tables, each indexed by noise width, height, and load
	Noise propagation polynomials	Polynomial	Four sets of three polynomials (width, height, and peak-time ratio), each a function of width, height, peak-time ratio, and load

Representing Noise Calculation Information

In the Library Compiler syntax, you can represent coupled noise information with an I-V characteristics lookup table model or polynomial model at the timing level or four simple attributes defined at the timing level:

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

I-V Characteristics Lookup Table Model

You can describe I-V characteristics in your libraries by using lookup tables. To define your lookup tables, use the following groups and attributes:

- The `iv_lut_template` group in the `library` group
- The `steady_state_current_high`, `steady_state_current_low`, and `steady_state_current_tristate` groups in the `timing` group

iv_lut_template Group

Use this library-level group to create templates of common information that multiple lookup tables can use. A table template specifies the I-V output voltage and the breakpoints for the axis. Assign each template a name. Make the template name the group name of a `steady_state_current_low group`, `steady_state_current_high group`, or `steady_state_current_tristate group`.

Syntax

```
library(name_string) {
      ...
      iv_lut_template(template_name_string) {
            variable_1: iv_output_voltage ;
            index_1 ("float,..., float");
      }
      ...
}
```

Template Variables

To specify I-V characteristics, define the following variable and index:

`variable_1`

The only valid value is `iv_output_voltage`, which specifies the I-V voltage of the output pin specified in the `pin` group. The voltage is measured from the pin to the ground.

`index_1`

The index values are a list of floating-point numbers that can be negative or positive. The values in the list must be in increasing order. The number of floating-point numbers in the `index_1` variable determines the dimension.

Example

```
iv_lut_template(my_current_low) {
   variable_1: iv_output_voltage;
   index_1 ("-1, -0.1, 0, 0.1 0.8, 1.6, 2");
}
iv_lut_template(my_current_high) {
   variable_1 : iv_output_voltage;
   index_1("-1, 0, 0.3, 0.5, 0.8, 1.5, 1.6, 1.7, 2");
}
```

Defining the Lookup Table Steady-State Current Groups

To specify the I-V characteristics curve for the nonlinear table model, use the `steady_state_current_high`, `steady_state_current_low`, or `steady_state_current_tristate` groups within the `timing` group.

Syntax for Table Model

```
timing() { /* for non-three-state arcs */
      steady_state_current_high(template_name_string) {
            values("float,..., float");
      }
      steady_state_current_low(template_name_string) {
            values("float,..., float");
      }
      ...
}

timing() { /* for three-state arcs */
      steady_state_current_tristate(template_name_string) {
            values("float,..., float);
            }
      }
      ...
}
```

float

The values are floating-point numbers indicating values for current.

The following rules apply to lookup table groups:

- Each table must have an associated name for the `iv_lut_template` it uses. The name of the template must be identical to the name defined in a library `iv_lut_template` group.
- You can overwrite `index_1` in a lookup table, but the overwrite must come before the definition of values.
- The current values of the table are stored in a `values` attribute. The values can be negative.

Example

```
timing() {
   ...
   steady_state_current_low(my_current_low) {
      values("-0.1, -0.05, 0, 0.1, 0.25, 1, 1.8");
   }
   steady_state_current_high(my_current_high) {
      values("-2, -1.8, -1.7, -1.4, -1, -0.5, 0, 0.1, 0.8");
   }
}
```

I-V Characteristics Curve Polynomial Model

As with the lookup table model, you can describe an I-V characteristics curve in your libraries by using the polynomial representation. To define your polynomial, use the following groups and attributes:

- The `poly_template` group in the `library` group
- The `steady_state_current_high`, `steady_state_current_low`, and `steady_state_current_tristate` groups within the `timing` group

poly_template Group

You can define a `poly_template` group at the library level to specify the equation variables, the variable ranges, the voltages mapping, and the piecewise data. The valid values for the variables are extended to include `iv_output_voltage`, `voltage`, `voltage`*i*, and `temperature`.

Syntax

```
library(name_string) {
      ...
      poly_template(template_name_string) {
            variables(variable_1_enum,..., variable_n_enum);
            variable_i_range: (float, float);
            ...
            variable_n_range: (float, float);
            mapping(voltage_enum, power_rail_id);
            domain(domain_name_string) {
                  variable_i_range: (float, float);
                  ...
                  variable_n_range: (float, float);
            }
            ...
      }
...
}
```

The syntax of the `poly_template` group is the same as that used for the delay model, except that the variables used in the format are

- `iv_output_voltage` for the output voltage of the pin
- `voltage`, `voltage`*i*, `temperature`

The piecewise model through the `domain` group is also supported.

For the complete `poly_template` group syntax description, see the *Library Compiler Technology and Symbol Libraries Reference Manual*.

Example

```
poly_template ( my_current_low ) {
   variables ( iv_output_voltage, voltage,
voltage1,temperature );
      mapping(voltage1, VDD2);
      variable_1_range (-1, 2);
      variable_2_range (1.4, 1.8);
      variable_3_range (1.1, 1.5);
      variable_4_range (-40, 125);
   }
}
```

Defining Polynomial Steady-State Current Groups

To specify the I-V characteristics curve to define the polynomial, use the `steady_state_current_high`, `steady_state_current_low`, and `steady_state_current_tristate` groups within the `timing` group.

Syntax for Polynomial Model

```
timing { /* for non-three-state arcs */
      steady_state_current_high(template_name_string) {
            orders("integer,..., integer");
            coefs("float,..., float");
            domain(domain_name_string) {
                  orders("integer,..., integer");
                  coefs("float,..., float");
            }
            ...
      }
      steady_state_current_low(template_name_string) {
                              ...
      }

timing() { /* for three-state arcs */
      steady_state_current_tristate(template_name_string) {
            ...
      }
      ...
}
```

The `orders`, `coefs`, and `variable_range` attributes represent the polynomial for the current for `high`, `low`, and `three-state`. The syntax for `orders` and `coefs` is the same as that described in the *Library Compiler Technology and Symbol Libraries Reference Manual*.

The output voltage, temperature, and any power rail of the cell are allowed as variables for `steady_state_current` groups.

Example

```
timing() {
   steady_state_current_low(my_current_low) {
      orders ("3, 3, 0, 0");
      coefs ("8.4165, 0.3198, -0.0004, 0.0000, \
               1133.8274, 8.7287, -0.0054, 0.0000, \
               139.8645, -60.3898, 0.0589, -0.0000, \
               -167.4473, 95.7112, -0.1018, 0.0000");
   }
   steady_state_current_high(my_current_high) {
      orders ("3, 3, 0, 0");
      coefs ("10.9165, 0.2198, -0.0003, 0.0000, \
                  1433.8274, 8.7287, -0.0054, 0.0000, \
                  128.8645, -60.3898, 0.0589, -0.0000, \
                  -167.4473, 95.7112, -0.1018, 0.0000");
   }
   ...
}
```

Using Steady-State Resistance Simple Attributes

To represent steady-state drive resistance values, use the following attributes to define the four regions:

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

These attributes are defined within the `timing` group to represent the steady-state drive resistance. If one of these attributes is missing, Library Compiler will not flag it, but the model will be inaccurate.

Syntax

```
pin(name) {
      ...
      timing() {
            ...
            steady_state_resistance_above_high : float;
            steady_state_resistance_below_low : float;
            steady_state_resistance_high : float;
            steady_state_resistance_low : float;
            ...
      }
}
```

float

The value of steady-state resistance for the four different noise regions in the I-V curve.

Example

```
steady_state_resistance_above_high : 200.0;
steady_state_resistance_below_low : 100.0;
steady_state_resistance_high : 100.0;
steady_state_resistance_low : 1100.0
```

Using I-V Curves and Steady-State Resistance for tied_off Cells

In tied-off cells, the output pins are tied to either high or low and there is no need to define timing information for related pins. The tied-off cells have been enhanced to accept I-V curve and steady-state resistance in the `timing` group. PrimeTime *SI* uses these timing groups for noise data access, and no timing data is requested. However, to specify only the noise data (I-V curves and steady-state resistance) in the `timing` group, you must specify a new Boolean attribute, `tied_off,` and set it to true.

Defining tied_off Attribute Usage

You can specify the I-V characteristics and steady-state drive resistance values on tied-off cells by using the `tied_off` attribute in the `timing` group.

Syntax

```
pin(name) {
      ...
      timing() {
            ...
            tied_off : boolean;
      /* timing type is not defined */
      /* steady-state resistance */
```

The following rules apply to `tied_off` cells:

- Steady-state resistance and I-V curves can coexist in the same timing arc of a `tied_off` output pin.
- If the output pin is tied to low (function : "0") and its timing arc specifies the `steady_state_current_high` group, Library Compiler issues an LBDB-625 error message to alert you that the output pin “high” has a tied-off timing arc containing an invalid `steady_state_current_low` group. Similarly if the output pin is tied to high (function : "1") and its timing arc specifies the `steady_state_current_low` group, Library Compiler issues an LBDB-625 error message to alert you that the output pin “low” has a tied-off timing arc containing an invalid `steady_state_current_high` group.
- If noise immunity and noise propagation are specified in the timing arcs of a `tied_off` pin, Library Compiler issues an LBDB-626 error message to alert you that the tied-off timing arc contains an invalid `noise_immunity` or `noise_propagation` group.
- If the `related_pin` attribute is specified on a `tied_off` output pin, Library Compiler issues an LBDB-635 error message to alert you that the tied-off timing arc contains an illegal `related_pin` attribute.

Example

```
pin (high) {
   direction : output;
   capacitance : 0;
   function : "1";

   /* noise information */
   timing() {
      tied_off : true;
      steady_state_resistance_high : 1.22;
      steady_state_resistance_above_high : 1.00;
      steady_state_current_high(iv1x5){
            index_1("0.3,0.75,1.0,1.2,2");
            values("-513.2,-447.9,-359.3,-245.7,497.3");
```

```
        }
    }
}
```

Representing Noise Immunity Information

In the Library Compiler syntax, you can represent noise immunity information with a

- Lookup table or a polynomial model at the timing level
- Input noise width range at the pin level
- Hyperbolic model at the pin level

Noise Immunity Lookup Table Model

You can represent noise immunity in your libraries by using lookup tables. To define your lookup tables, use the following groups and attributes:

- `noise_lut_template` group in the `library` group
- `noise_immunity_above_high`, `noise_immunity_above_low`, `noise_immunity_below_low`, and `noise_immunity_high` groups in the `timing` group

noise_lut_template Group

Use this library-level group to create templates of common information that multiple noise immunity lookup tables can use.

A table template specifies the input noise width, the output load, and their corresponding breakpoints for the axis. Assign each template a name, and make the name the group name of a noise immunity group.

Syntax

```
library(name_string) {
      ...
      noise_lut_template(template_name_string) {
            variable_1: value;
            variable_2: value;
            index_1 ("float,..., float");
            index_2 ("float,..., float");
      }
      ...
}
```

Template Variables

The library-level table template specifying noise immunity can have two variables (`variable_1` and `variable_2`). The variables indicate the parameters used to index the lookup table along the first and second table axes. The parameters are `input_noise_width` and `total_output_net_capacitance`.

The index values in `index_1` and `index_2` are a list of positive floating-point numbers. The values in the list must be in increasing order.

The unit for the input noise width is the library time unit.

Example

```
noise_lut_template(my_noise_reject) {
  variable_1 : input_noise_width;
  variable_2 : total_output_net_capacitance;
  index_1("0, 0.1, 0.3, 1, 2");
  index_2("1, 2, 3, 4, 5");
}
```

Defining the Noise Immunity Table Groups

To represent noise immunity, use the noise_immunity_above_high, noise_immunity_below_low, noise_immunity_high, and noise_immunity_low groups within the `timing` group.

Syntax for Noise Immunity Table Model

```
timing() {
     noise_immunity_above_high(template_name_string) {
          index_1 ("float,..., float"):
          index_2 ("float,..., float"):
          values("float,...,float"..."float,...,float");
     }
     noise_immunity_below_low(template_name_string) {
          ...
     }
     noise_immunity_high(template_name_string) {
          ...
     }
     noise_immunity_low(template_name_string) {
          ...
     }
}
```

The following rules apply to the noise immunity groups:

- These tables are optional, and each of them can exist separately on the library timing arcs.

- Each noise immunity table has an associated name for the `noise_lut_template` it uses. The name of the table must be identical to the name defined in a library `noise_lut_template` group.
- Each table is two-dimensional. The indexes are `input_noise_width` and `total_output_net_capacitance`. The values in the table are the noise heights (that is, height as a function of width and output load).
- You can overwrite any or both indexes in a noise template. However, the overwrite must occur before the actual definition of the values.
- The height values of the table are stored in the `values` attribute. Each height value is the absolute difference of the noise bump height voltage and the related rail voltage and is, therefore, a positive number. Any point over this curve describes a height/width combination that causes functional failure.
- The unit for the height is the library voltage unit.
- For points outside table ranges, extrapolation is used.

Example

```
pin ( Y ) {
   ....
   timing () {
      noise_immunity_below_low (my_noise1) {
         values ("1, 0.8, 0.5", \
                  "1, 0.8, 0.5", \
                  "1, 0.8, 0.5");
      }
      noise_immunity_above_high (my_noise1){
         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(name_string) {
      ...
      poly_template(template_name_string) {
            variables(variable_ienum,..., variable_nenum);
            variable_i_range: (float, float);
            ...
            variable_n_range: (float, float);
            mapping(voltage_enum, power_rail_id);
            domain(domain_name_string); {
                  variable_i_range: (float, float);
                  ...
                  variable_n_range: (float, float);
            }
      }
      ...
}
```

Template Variables

The syntax of the `poly_template` group is the same as that used for the delay model, except that the variables used in the format are

- `input_noise_width`
- `total_output_net_capacitance`
- `voltage`, `voltage`*i*, `temperature`

The piecewise model through the `domain` group is also supported.

For the complete `poly_template` group syntax description, see the *Library Compiler Technology and Symbol Libraries Reference Manual*.

Example

```
poly_template (my_noise_reject) { /* existing syntax */
   variables (input_noise_width,voltage,voltage1, temperature, \
   total_output_net_capacitance);
   mapping(voltage1, VDD2);
   variable_1_range (0, 2);
   variable_2_range (1.4, 1.8);
   variable_3_range (1.1, 1.5);
   variable_4_range (-40, 125);
   variable_5_range (0.01, 1.0);
```

```
   domain (typ) {
      variables (input_noise_width,voltage,voltage1, \
                 temperature,total_output_net_capacitance);
      variable_1_range (0, 2);
   }
}
```

Defining the Noise Immunity Polynomial Groups

To represent noise immunity, use the `noise_immunity_above_high`, `noise_immunity_below_low`, `noise_immunity_high`, and `noise_immunity_low` groups within the `timing` group.

Syntax

```
      ...
      timing() {
            noise_immunity_above_high(template_name_string) {
                  orders("integer,..., integer");
                  coefs("float,..., float");
                  ...
                  domain(domain_name_string) {
                        orders("integer,..., integer");
                        coefs("float,...,float");
                  }
                  ...
            }
            noise_immunity_below_low(template_name_string) {
                  ...
            }
            noise_immunity_high(template_name_string) {
                  ...
            }
            noise_immunity_low(template_name_string) {
                  ...
            }
            ...
      }
```

The syntax for `poly_template`, `orders`, `and coefs` is the same as that described in the *Library Compiler User Guide: Modeling Timing and Power Technology Libraries*. Because the polynomial model is a superset of the lookup table model, all syntax supported in the lookup table is also supported in the polynomial model. For example, you can have a polynomial `noise_immunity_high` and a table `noise_immunity_low` defined in the same group in a scalable polynomial delay model library.

Example

```
noise_immunity_low (my_noise_reject) {
   domain (typ) {
```

```
      orders ("1, 1, 1, 1, 1")
      coefs ("1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, \
         1.0, 1.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, \
         1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0" );
      domain (min) {
         orders("1 3 1 1");
         coefs("-0.01, 0.02, 1.41, -0.54, \
                  1.85, 1.83, -5.58, -2.96, -0.0001, \
                  0.0001, -0.002, -0.0019, 0.002, \
                  0.0012, -0.010, -0.0061, 0.034, \
                  0.015, 2.08, -0.22, 4.13, 2.44, \
                  -14.02, -7.83, 7.09e-05, -1.98e-05, \
                  -0.0019, 0.0009, 0.0065, -0.0004, \
                  -0.027, -0.016");
      }
   }
}
```

Input Noise Width Ranges at the Pin Level

So that the PrimeTime *SI* tool can identify whether a noise immunity or propagation table is referenced within the noise range indexes, the Library Compiler syntax allows you to specify the minimum and maximum values of the input noise width.

Defining the input_noise_width Range Limits

You can specify two `float` attributes, `min_input_noise_width` and `max_input_noise_width`, at the pin level. These attributes are optional and specify the minimum and maximum values of the input noise width.

Syntax

```
pin(name_string) {
      ...
      /* used for noise immunity or propagation */
      min_input_noise_width : float;
      max_input_noise_width: float;
      ...
}
```

float

The values of `min_input_noise_width` and `max_input_noise_width` are the minimum and maximum input noise width, in library time units.

The following rules apply to `input_noise_width` range limits:

- The `min_input_noise_width` and `max_input_noise_width` attributes can be defined only on input or inout pins. Otherwise, Library Compiler issues an LBDB-619 error message to alert you that the `min_input_noise_width` and `max_input_noise_width` attributes cannot be specified on an output pin.
- The `min_input_noise_width` and `max_input_noise_width` attributes must both be defined. Otherwise, Library Compiler issues an LBDB-622 error message to alert you that the `max_input_noise_width` (or `min_input_noise_width`) attribute is missing for pin A.
- Library Compiler checks whether the `min_input_noise_width <= max_input_noise_width` constraints are met. If the constraints aren't met, Library Compiler issues an LBDB-623 error message to alert you that the `min_input_noise_width` attribute value is larger than the `max_input_noise_width` attribute value for pin A.
- Library Compiler does not check whether the specification of these noise range attributes is associated with noise groups.

Example

```
pin ( O) {
   direction : output ; /* existing syntax */
   capacitance : 1 ; /* existing syntax */
   fanout_load : 1 ; /* existing syntax */

   /* Noise range */
   min_input_noise_width : 0.0;
   max_input_noise_width : 2.0;

   /* Timing group defines what is acceptable noise on input pins. */

   timing () {
   /* Noise immunity.
   * Defines maximum allowed noise height for given pulse width.
   * Pulse height is absolute value from the signal level.
   * Any of the following four tables are optional. */

      noise_immunity_low (my_noise_reject) {
         values ("1.5, 0.9, 0.8, 0.65, 0.6");
      }
      noise_immunity_high (my_noise_reject) {
         values ("1.3, 0.8, 0.7, 0.6, 0.55");
      }
      noise_immunity_below_low (my_noise_reject_outside_rail) {
         values ("1, 0.8, 0.5");
      }
      noise_immunity_above_high (my_noise_reject_outside_rail) {
         values ("1, 0.8, 0.5");
      }
```

```
   } /* end of timing group */
} /* end of pin group */
```

Defining the Hyperbolic Noise Groups

To specify hyperbolic noise immunity information, use the `hyperbolic_noise_above_high`, `hyperbolic_noise_below_low`, `hyperbolic_noise_high`, and `hyperbolic_noise_low` groups within the `pin` group.

Syntax

```
pin(name_string) {
      ...
      hyperbolic_noise_above_high() {
            height_coefficient : float;
            area_coefficient : float;
            width_coefficient : float;
      }
      hyperbolic_noise_below_low() {
            ...
      }
      hyperbolic_noise_high() {
            ...
      }
      hyperbolic_noise_low() {
            ...
      }
      ...
}
```

float

The coefficient values for height, width, and area must be 0 or a positive number.

The following rules apply to noise immunity groups:

- The hyperbolic noise groups are optional, and each can be defined separately from the other three.
- For the same region (above-high, below-low, high, or low), the hyperbolic noise groups can coexist with normal noise immunity tables.
- For different regions (above-high, below-low, high, or low), a combination of tables and hyperbolic functions is allowed. For example, you might have a hyperbolic function for below and above the rails and have tables for high and low tables on the same pin.
- When no table or hyperbolic function is defined for a given pin, the application checks other measures for noise immunity, such as DC noise margins.

- The unit for `height` and `height_coefficient` is the library unit of voltage. The unit for `width` and `width_coefficient` is the library unit of time. The unit for `area_coefficient` is the library unit of voltage multiplied by the library unit of time.

Example

```
hyperbolic_noise_low() {
    height_coefficient : 0.4;
    area_coefficient : 1.1;
    width_coefficient : 0.1;
}
hyperbolic_noise_high() {
    height_coefficient : 0.3;
    area_coefficient : 0.9;
    width_coefficient : 0.1;
}
```

Representing Propagated Noise Information

In the Library Compiler syntax, you can represent propagated noise information at the timing level by using a

- Propagated Noise Lookup Table Model
- Propagated Noise Polynomial Model

Propagated Noise Lookup Table Model

You can represent propagated noise in your libraries by using lookup tables. To define your lookup tables, use the `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(name_string) {...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_name_string) {index_1
("float,..., float");index_2 ("float,..., float");index_3 ("float,...,
float");values("float,..., float,"..."float,..., float");}
   propagated_noise_height_below_low (temp_name_string) {...}
```

```
propagated_noise_width_above_high (temp_name_string) {...}
propagated_noise_width_below_low (temp_name_string) {...}
propagated_noise_height_high(template_name_string) {...}
propagated_noise_height_low(template_name_string) {...}
propagated_noise_width_high(template_name_string) {...}
propagated_noise_width_low(template_name_string) {...}...}
```

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 is used.

Example

```
propagated_noise_width_high(my_propagated_noise) {
   values ("0.01, 0.10, 0.15", "0.04, 0.14, 0.18",\
            "0.05, 0.15, 0.24","0.07, 0.17, 0.32");
}
propagated_noise_height_high(my_propagated_noise) {
   values ("0.01, 0.20, 0.25", "0.04, 0.24, 0.28",\
               "0.05, 0.25, 0.28","0.07, 0.27, 0.35");
}
```

Propagated Noise Polynomial Model

As with the lookup table model, you can describe propagated noise in your libraries by using polynomial representation. To define your polynomial, use the `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 syntax for `poly_template`, `orders`, and `coefs` is the same as that described in *Library Compiler Technology and Symbol Libraries Reference Manual*.

The `input_peak_time_ratio` is always specified as a ratio of width, so it is a value between 0.0 and 1.0.

Example

```
poly_template(my_propagated_noise) {
   variables (input_noise_width, input_noise_height,
input_peak_time_ratio,
             total_output_net_capacitance);
   variable_1_range (0.01, 2);
   variable_2_range (0, 0.8);
   variable_3_range (0.0, 1.0);
   variable_4_range (0, 2);
} /* poly_template(propagated_noise) */
```

Defining Propagated Noise Groups for Polynomial Representation

To specify polynomial representation, use the `propagated_noise_height_above_high`, `propagated_noise_height_below_low`, `propagated_noise_height_high`, `propagated_noise_height_low`, `propagated_noise_width_above_high`, `propagated_noise_width_below_low`, `propagated_noise_width_high`, `propagated_noise_width_low`, `propagated_noise_peak_time_ratio_above_high`, `propagated_noise_peak_time_ratio_below_low`, `propagated_noise_peak_time_ratio_high`, and `propagated_noise_peak_time_ratio_low` groups within the `timing` group to define the polynomial.

The `peak_time_ratio` groups are supported only in the polynomial model.

Syntax for Polynomial

```
timing() {
          ...
     propagated_noise_height_above_high (temp_name_string) {
          variable_i_range: (float, float);
          orders("integer,..., integer");
          coefs("float,..., float");
          domain(domain_name_string) {
               variable_i_range: (float, float);
               orders("integer,..., integer");
               coefs("float,..., float");
          }
          ...
     }
     propagated_noise_width_above_high (temp_name_string) {
          ...
     }
     propagated_noise_height_below_low (temp_name_string) {
          ...
     }
     propagated_noise_width_below_low (temp_name_string) {
          ...
     }
     propagated_noise_height_high(temp_name_string) {
          ...
     }
     propagated_noise_width_high(temp_name_string) {
          ...
     }
     propagated_noise_height_low(temp_name_string) {
          ...
     }
     propagated_noise_width_low(temp_name_string) {
          ...
     }
     propagated_noise_peak_time_ratio_above_high (temp_name_string) {
           ...
     }
     propagated_noise_peak_time_ratio_below_low(temp_name_string) {
           ...
     }
     propagated_noise_peak_time_ratio_high (temp_name_string) {
           ...
     }
     propagated_noise_peak_time_ratio_low (temp_name_string) {
           ...
     }
     ...
}
```

The syntax for `poly_template`, `orders`, `and coefs` is the same as that described in the *Library Compiler User Guide: Modeling Timing and Power Technology Libraries*.Because the polynomial model is a superset of the lookup table model, all syntax supported in the lookup table is also supported in the polynomial model. For example, you can have a `propagated_noise_width_high` polynomial and a `propagated_noise_width_low table` defined in the same group in a scalable polynomial delay model library.

Example

```
propagated_noise_width_high(my_propagated_noise) {
   orders("1, 1, 1, 1 ");
   coefs("1, 2, 3, 4 ,\
            1, 2, 3, 4 ,\
            1, 2, 3, 4 ,\
            1, 2, 3, 4 ");
}
propagated_noise_height_high(my_propagated_noise) {
   orders("1, 1, 1, 1 ");
   coefs("1, 2, 3, 4 ,\
            1, 2, 3, 4 ,\
            1, 2, 3, 4 ,\
            1, 2, 3, 4 ");
}
```

Examples of Modeling Noise

The examples in this section model libraries for noise extension for scalable polynomials and nonlinear lookup table model libraries.

Scalable Polynomial Model Noise 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(VDD1, 1.6);
      power_rail(VDD2, 1.3);
   }
   nom_voltage : 1.0;
   nom_temperature : 40;
   nom_process : 1.0;
```

```
/* Templates of DC noise margins and output levels */
input_voltage(MY_CMOS_IN) {
    vil : 0.3;
    vih : 1.1;
    vimin : -0.3;
    vimax : VDD + 0.3;
}
output_voltage(MY_CMOS_OUT) {
    vol : 0.1;
    voh : 1.4;
    vomin : -0.3;
    vomax : VDD + 0.3;
}
/* Template definitions for noise immunity. Variable:
* input_noise_width */
poly_template ( my_noise_reject ) {
        temperature,total_output_net_capacitance);
    variables ( input_noise_width, voltage, voltage1, \
                  temperature,total_output_net_capacitance);
    mapping(voltage, VDD1);
    mapping(voltage1, VDD2);
    variable_1_range (0, 2);
    variable_2_range (1.4, 1.8);
    variable_3_range (1.1, 1.5);
    variable_4_range (-40, 125);
    variable_5_range (0.0, 1.0);
    domain (typ) {
        variables ( input_noise_width, voltage, voltage1,\
               temperature, total_output_net_capacitance);
        variable_1_range (0, 2);
        variable_2_range (1.5, 1.7);
        variable_3_range (1.2, 1.4);
        variable_4_range (25, 25);
        variable_5_range (0.0, 1.0);
        mapping(voltage, VDD1);
        mapping(voltage1, VDD2);
    }
    domain (min) {
        variables ( input_noise_width, voltage, voltage1, \
                      temperature );
        variable_1_range (0, 2);
        variable_2_range (1.7, 1.8);
        variable_3_range (1.4, 1.5);
        variable_4_range (-40, -40);
        mapping(voltage, VDD1);
        mapping(voltage1, VDD2);
    }
    domain (max) {
        variables ( input_noise_width, voltage, voltage1, \
                      temperature );
        variable_1_range (0, 2);
        variable_2_range (1.6, 1.7);
        variable_3_range (1.1, 1.2);
        variable_4_range (125, 125);
        mapping(voltage, VDD1);
        mapping(voltage1, VDD2);
    }
} /* end poly_template (my_noise_reject) */
```

```
poly_template ( my_noise_reject_outside_rail ) {
    variables ( input_noise_width, voltage, voltage1, \
                    temperature );
    mapping(voltage, VDD1);
    mapping(voltage1, VDD2);
    variable_1_range (0, 2);
    variable_2_range (1.4, 1.8);
    variable_3_range (1.1, 1.5);
    variable_4_range (-40, 125);
} /* end poly_template ( my_noise_reject_outside_rail ) */
/* Template definitions for I-V characteristics. Variable:
* iv_output_voltage */
poly_template ( my_current_low ) {
    variables ( iv_output_voltage, voltage, voltage1, \
                    temperature );
    mapping(voltage, VDD1);
    mapping(voltage1, VDD2);
    variable_1_range (-1, 2);
    variable_2_range (1.4, 1.8);
    variable_3_range (1.1, 1.5);
    variable_4_range (-40, 125);
} /* end poly_template ( my_current_low ) */
poly_template ( my_current_high ) {
    variables ( iv_output_voltage, voltage, voltage1, \
                    temperature );
    mapping(voltage, VDD1);
    mapping(voltage1, VDD2);
    variable_1_range (-1, 2);
    variable_2_range (1.4, 1.8);
    variable_3_range (1.1, 1.5);
    variable_4_range (-40, 125);
} /* end poly_template ( my_current_high ) */
/* Template definitions for propagated noise. Variables:
* input_noise_width
* input_noise_height
* input_peak_time_ratio
* total_output_net_capacitance */
poly_template(my_propagated_noise) {
    variables ( input_noise_width, input_noise_height, \
                    input_peak_time_ratio \
                    total_output_net_capacitance, voltage, \
                    voltage1, temperature );
    mapping(voltage, VDD1);
    mapping(voltage1, VDD2);
    variable_1_range (0.01, 2);
    variable_2_range (0, 0.8);
    variable_3_range (0.0, 1.0);
    variable_4_range (0, 2);
    variable_5_range (1.4, 1.8);
    variable_6_range (1.1, 1.5);
    variable_7_range (-40, 125);
} /* end poly_template (my_propagated_noise) */
/* INVERTER */
cell ( INV ) {
    area : 1 ;
    pin ( A ) {
        direction : input ;
        capacitance : 1 ;
```

```
        fanout_load : 1 ;
            /* DC noise margins.
            * These are used for compatibility of level shifters.
            * In noise analysis, they are the least accurate way
            * to define noise margins.
            * Compatible: can coexist in the pin group with any
            * other noise margin definition. */
        input_voltage : MY_CMOS_IN ;
            /* Noise group defines what is acceptable noise on input
            * pins. */
            /* Hyperbolic noise immunity.
            * Another way to specify noise immunity.
            * Mutually exclusive: noise_immunity_low cannot be
            * together with
            *hyperbolic_noise_immunity_low, and so on.
            * Defines pulse_height = height_coefficient +
            * area_coefficient / (width - width_coefficient)
            * Characterization recommendation: Use
            * hyperbolic_noise_immunity_*
            * if it can fit the curve, otherwise use
            * table noise_immunity_* */
        hyperbolic_noise_low() {
            height_coefficient : 0.4;
            area_coefficient : 1.1;
            width_coefficient : 0.1;
        }
        hyperbolic_noise_high() {
            height_coefficient : 0.3;
            area_coefficient : 0.9;
            width_coefficient : 0.1;
        }
        hyperbolic_noise_below_low() {
            height_coefficient : 0.1;
            area_coefficient : 0.3;
            width_coefficient : 0.01;
        }
        hyperbolic_noise_above_high() {
            height_coefficient : 0.1;
            area_coefficient : 0.3;
            width_coefficient : 0.01;
        }
    } /* end pin (A) */
    pin ( Y ) {
        direction : output ;
        max_fanout : 10 ;
        function : " !A ";
        output_voltage : MY_CMOS_OUT ;
        timing () {
            related_pin : A ;
            /* Steady state drive resistance */
            steady_state_resistance_high : 1500;
            steady_state_resistance_low : 1100;
            steady_state_resistance_above_high : 200;
            steady_state_resistance_below_low : 100;
            /* I-V curve.
            * Describes how much current the pin can deliver in a given state for
            * a given voltage on the pin.
            * Voltage is measured from the pin to ground, current is measured
```

```
* flowing into the cell (both can be either positive or negative). */
steady_state_current_low(my_current_low) {
   orders ("3, 3, 0, 0");
   coefs ("8.4165, 0.3198, -0.0004, 0.0000, \
   1133.8274, 8.7287, -0.0054, 0.0000, \
   139.8645, -60.3898, 0.0589, -0.0000, \
   -167.4473, 95.7112, -0.1018, 0.0000");
}
steady_state_current_high(my_current_high) {
   orders ("3, 3, 0, 0");
   coefs ("10.9165, 0.2198, -0.0003, 0.0000, \
   1433.8274, 8.7287, -0.0054, 0.0000, \
   128.8645, -60.3898, 0.0589, -0.0000, \
   -167.4473, 95.7112, -0.1018, 0.0000");
}
/* Noise immunity.
* Defines maximum allowed noise height for given pulse width.
* Pulse height is absolute value from the signal level.
* Any of the 4 tables below are optional. */
noise_immunity_low (my_noise_reject) {
   domain (typ) {
      orders ("3, 3, 0, 0, 0");
      coefs ("11.4165, 0.2198, -0.0003, 0.0000, \
      1353.8274, 8.7287, -0.0054, 0.0000, \
      149.8645, -60.3898, 0.0589, -0.0000, \
      -167.4473, 95.7112, -0.1018, 0.0000");
   }
   domain (min) {
      orders ("3, 3, 0, 0");
      coefs ("6.964065, 0.134078, -0.000183, 0.0000, \
      825.834714, 5.324507, -0.003294, 0.0000, \
      91.417345, -36.837778, .035929, -0.0000, \
      -102.142853, 58.383832, -.062098, 0.0000");
   }
   domain (max) {
      orders ("3, 3, 0, 0");
      coefs ("19.065555, 0.367066, -0.000501, 0.0000, \
      2260.891758, 14.576929, -0.009018, 0.0000, \
      250.273715, -100.850966, 0.098363, -0.0000, \
      -279.636991, 159.837704, -0.170006, 0.0000");
   }
}
noise_immunity_high (my_noise_reject) {
   domain (typ) {
      orders ("3, 3, 0, 0, 0");
      coefs ("12.4165, 0.2198, -0.0003, 0.0000, \
      1353.8274, 8.7287, -0.0054, 0.0000, \
      129.8645, -60.3898, 0.0589, -0.0000, \
      -147.4473, 95.7112, -0.1018, 0.0000");
   }
   domain (min) {
      orders ("3, 3, 0, 0");
      coefs ("6.364065, 0.134078, -0.000183, 0.0000, \
      845.834714, 5.324507, -0.003294, 0.0000, \
      91.417345, -36.837778, .035929, -0.0000, \
      -103.142853, 58.383832, -.062098, 0.0000");
   }
   domain (max) {
```

```
            orders ("3, 3, 0, 0");
            coefs ("19.265555, 0.367066, -0.000601, 0.0000, \
            2460.891758, 14.576929, -0.009018, 0.0000, \
            250.273715, -130.850966, 0.098363, -0.0000, \
            -279.636991, 159.837704, -0.170006, 0.0000");
         }
      }
      noise_immunity_below_low (my_noise_reject_outside_rail) {
         orders ("3, 3, 0, 0");
         coefs ("10.4165, 0.1198, -0.0003, 0.0000, \
         1333.8274, 8.7287, -0.0054, 0.0000, \
         149.8645, -60.3898, 0.0589, -0.0000, \
         -167.4473, 95.7112, -0.1018, 0.0000");
      }
      noise_immunity_above_high (my_noise_reject_outside_rail) {
         orders ("3, 3, 0, 0");
         coefs ("12.4165, 0.2298, -0.0003, 0.0000, \
         1253.8274, 8.7287, -0.0054, 0.0000, \
         149.8645, -60.3898, 0.0589, -0.0000, \
         -167.4473, 95.7112, -0.1018, 0.0000");
      }
   /* Propagated noise.
      * It is a function of input noise width and height and output
      * capacitance. Width and height are in separate tables. */
      propagated_noise_width_high(my_propagated_noise) {
         orders ("1, 2, 1, 0, 0, 0, 0");
         coefs ("8.4165, 0.3198, -0.0004, 0.2000, \
         1.8645, -6.3898, 0.0589, -0.03000, \
         -1.4473, 9.7112, -0.1018, 0.3500, ");
      }
      propagated_noise_height_high(my_propagated_noise) {
         orders ("1, 2, 1, 0, 0, 0, 0");
         coefs ("0.4165, 0.3198, -0.0014, 0.2000, \
         0.8645, -6.3898, 0.0589, -0.03000, \
         -0.4473, 0.7112, -0.1018, 0.3500, ");
      }
      propagated_noise_peak_time_ratio_high(my_propagated_noise) {
         orders ("1, 2, 1, 0, 0, 0, 0");
         coefs ("0.4165, 0.3198, 0.0014, 0.2000, \
         0.8645, 0.3898, 0.0589, 0.03000, \
         0.4473, 0.7112, 0.1018, 0.3500, ");
      }
      propagated_noise_width_low(my_propagated_noise) {
         orders ("1, 2, 1, 0, 0, 0, 0");
         coefs ("8.4165, 0.3198, -0.0004, 0.2000, \
         1.8645, -6.3898, 0.0589, -0.03000, \
         -1.4473, 9.7112, -0.1018, 0.3500, ");
      }
      propagated_noise_height_low(my_propagated_noise) {
         orders ("1, 2, 1, 0, 0, 0, 0");
         coefs ("0.4165, 0.3198, -0.0014, 0.2000, \
         0.8645, -6.3898, 0.0589, -0.03000, \
         -0.4473, 0.7112, -0.1018, 0.3500, ");
      }
      propagated_noise_peak_time_ratio_low(my_propagated_noise) {
         orders ("1, 2, 1, 0, 0, 0, 0");
         coefs ("0.4165, 0.3198, 0.0014, 0.2000, \
         0.8645, 0.3898, 0.0589, 0.03000, \
```

```
                0.4473, 0.7112, 0.1018, 0.3500, ");
            }
            propagated_noise_width_above_high(my_propagated_noise) {
                orders ("1, 2, 1, 0, 0, 0, 0");
                coefs ("8.4165, 0.3198, -0.0004, 0.2000, \
                1.8645, -6.3898, 0.0589, -0.03000, \
                -1.4473, 9.7112, -0.1018, 0.3500, ");
            }
            propagated_noise_height_above_high(my_propagated_noise) {
                orders ("1, 2, 1, 0, 0, 0, 0");
                coefs ("0.4165, 0.3198, -0.0014, 0.2000, \
                0.8645, -6.3898, 0.0589, -0.03000, \
                -0.4473, 0.7112, -0.1018, 0.3500, ");
            }
            propagated_noise_peak_time_ratio_above_high(my_propagated_noise) {
                orders ("1, 2, 1, 0, 0, 0, 0");
                coefs ("0.4165, 0.3198, 0.0014, 0.2000, \
                0.8645, 0.3898, 0.0589, 0.03000, \
                0.4473, 0.7112, 0.1018, 0.3500, ");
            }
            propagated_noise_width_below_low(my_propagated_noise) {
                orders ("1, 2, 1, 0, 0, 0, 0");
                coefs ("8.4165, 0.3198, -0.0004, 0.2000, \
                1.8645, -6.3898, 0.0589, -0.03000, \
                -1.4473, 9.7112, -0.1018, 0.3500, ");
            }
            propagated_noise_height_below_low(my_propagated_noise) {
                orders ("1, 2, 1, 0, 0, 0, 0");
                coefs ("0.4165, 0.3198, -0.0014, 0.2000, \
                0.8645, -6.3898, 0.0589, -0.03000, \
                -0.4473, 0.7112, -0.1018, 0.3500, ");
            }
            propagated_noise_peak_time_ratio_below_low(my_propagated_noise) {
                orders ("1, 2, 1, 0, 0, 0, 0");
                coefs ("0.4165, 0.3198, 0.0014, 0.2000, \
                0.8645, 0.3898, 0.0589, 0.03000, \
                0.4473, 0.7112, 0.1018, 0.3500, ");
            }
            cell_rise(scalar) { values("0");}
            rise_transition(scalar) { values("0");}
            cell_fall(scalar) { values("0");}
            fall_transition(scalar) { values("0");}
         } /* end of timing group */
      } /* end of pin (Y) */
   } /* end of cell (INV) */
} /* end of library (my_noise_lib)
```

Nonlinear Delay Model Library With Noise Information

A nonlinear delay model noise library is limited to a fixed voltage.

```
library (my_noise_lib) {
   delay_model : "table_lookup";
   time_unit : "1ns";
   voltage_unit : "1V";
   current_unit : "1mA";
```

```
capacitive_load_unit (1,pf);
pulling_resistance_unit : 1kohm;
nom_voltage : 1.6;
nom_temperature : 40.0;
nom_process : 1.0;
/* Templates of input and output levels (used for DC noise margin) */
input_voltage(MY_CMOS_IN) {
   vil : 0.3;
   vih : 1.1;
   vimin : -0.3;
   vimax : VDD + 0.3;
}
output_voltage(MY_CMOS_OUT) {
   vol : 0.1;
   voh : 1.4;
   vomin : -0.3;
   vomax : VDD + 0.3;
}
/* Template definitions for noise immunity. Variable:
* input_noise_width */
noise_lut_template(my_noise_reject) {
   variable_1 : input_noise_width;
   variable_2 : total_output_net_capacitance;
   index_1("0, 0.1, 0.3, 1, 2");
   index_2("0, 0.1, 0.3, 1, 2");
}
noise_lut_template(my_noise_reject_outside_rail) {
   variable_1 : input_noise_width;
   variable_2 : total_output_net_capacitance;
   index_1("0, 0.1, 2");
   index_2("0, 0.1, 2");
}
/* Template definitions for I-V characteristics. Variable:
* iv_output_voltage */
iv_lut_template(my_current_low) {
   variable_1 : iv_output_voltage
   index_1("-1, -0.1, 0, 0.1 0.8, 1.6, 2");
}
iv_lut_template(my_current_high) {
   variable_1 : iv_output_voltage
   index_1("-1, 0, 0.3, 0.5, 0.8, 1.5, 1.6, 1.7, 2");
}
/* Template definitions for propagated noise. Variables:
* input_noise_width
* input_noise_height
* total_output_net_capacitance */
propagation_lut_template(my_propagated_noise) {
   variable_1 : input_noise_width;
   variable_2 : input_noise_height;
   variable_3 : total_output_net_capacitance;
   index_1("0.01, 0.2, 2");
   index_2("0.2, 0.8");
   index_3("0, 2");
}
cell (tieoff_30_esd) {
   pin (high) {
      direction : output;
      capacitance : 0;
```

```
            function : "1";
            /* noise information */
            timing() {
               tied_off : true;
               steady_state_resistance_high : 1.22;
               steady_state_resistance_above_high : 1.00;
               steady_state_current_high(iv1x5){
                  index_1("0.3,0.75,1.0,1.2,2");
                  values("-513.2,-447.9,-359.3,-245.7,497.3");
               }
            }
         }
         pin (low) {
            direction : output;
            capacitance : 0;
            function : "0";
            /* noise information */
            timing() {
               tied_off : true;
               steady_state_resistance_low : 0.1;
               steady_state_resistance_below_low : 0.4;
               steady_state_current_low(iv1x5){
                  index_1("-0.25,0.3,0.5,1.0,1.8");
                  values("-595.4,555.4,690.5,774.75,822.5");
               }
            }
         }
      }
}
/* INVERTER */
      cell ( INV ) {
         area : 1 ;
         pin ( A ) {
            direction : input ;
            capacitance : 1 ;
            fanout_load : 1 ;
            /* DC noise margins.
            * These are used for compatibility of level shifters. In noise
            * analysis they are the least accurate way to define noise margins.
            * Compatible: can coexist in the pin group with any other noise margin
            * definition. */
            input_voltage : MY_CMOS_IN ;
            /* Timing group defines what is acceptable noise on input pins. */
            /* Hyperbolic noise immunity.
            * Another way to specify noise immunity.
            * Mutually exclusive: noise_immunity_low cannot be together with
            * hyperbolic_noise_immunity_low, etc.
            * Defines pulse_height = height_coefficient +
            * area_coefficient / (width - width_coefficient)
            * Characterization recommendation: use hyperbolic_noise_immunity_*
            * if can fit the curve, otherwise table noise_immunity_* */
            hyperbolic_noise_low() {
               height_coefficient : 0.4;
               area_coefficient : 1.1;
               width_coefficient : 0.1;
            }
            hyperbolic_noise_high() {
               height_coefficient : 0.3;
```

```
        area_coefficient : 0.9;
        width_coefficient : 0.1;
    }
    hyperbolic_noise_below_low() {
        height_coefficient : 0.1;
        area_coefficient : 0.3;
        width_coefficient : 0.01;
    }
    hyperbolic_noise_above_high() {
        height_coefficient : 0.1;
        area_coefficient : 0.3;
        width_coefficient : 0.01;
    }
} /* end of pin A */
pin ( Y ) {
    direction : output ;
    max_fanout : 10 ;
    function : " !A ";
    output_voltage : MY_CMOS_OUT
    min_input_noise_width : 0.0;
    max_input_noise_width : 2.0;
    timing () {
        related_pin : A ;
        /* Steady-state drive resistance */
        steady_state_resistance_high : 1500;
        steady_state_resistance_low : 1100;
        steady_state_resistance_above_high : 200;
        steady_state_resistance_below_low : 100;
        /* I-V curve.
         * Describes how much current the pin can deliver in a given state for
        * a given voltage on the pin. The steady_state_resistance*_max is the
        * highest resistance in the I-V curve, the
        * steady_state_resistance*_min is the lowest.
        * Mutually exclusive: If steady_state_resistance_low* or
        * steady_state_resistance_max or steady_state_resistance_min is
        * specified, the I-V curve cannot be specified.
        * Characterization recommendation: Use steady_state_resistance* if
        * an I-V curve cannot be generated.
        * Voltage is measured from the pin to ground, current measured
        * flowing into the cell (both can be either positive or negative). */
        steady_state_current_low(my_current_low) {
            values("0.1, 0.05, 0, -0.1, -0.25, -1, -1.8");
        }
        steady_state_current_high(my_current_high) {
            values("2, 1.8, 1.7, 1.4, 1, 0.5, 0, -0.1, -0.8");
        }
        cell_rise(scalar) { values("0");}
        rise_transition(scalar) { values("0");}
        cell_fall(scalar) { values("0");}
        fall_transition(scalar) { values("0");}
        /* Noise immunity.
        * Defines the maximum allowed noise height for given pulse width.
        * Pulse height is the absolute value from the signal level.
        * Any of the following four tables are optional. */
        noise_immunity_low (my_noise_reject) {
            values ("1.5, 0.9, 0.8, 0.65, 0.6", \
            "1.5, 0.9, 0.8, 0.65, 0.6", \
      "1.5, 0.9, 0.8, 0.65, 0.6", \
```

```
            "1.5, 0.9, 0.8, 0.65, 0.6", \
            "1.5, 0.9, 0.8, 0.65, 0.6");
             }
             noise_immunity_high (my_noise_reject) {
                 values ("1.3, 0.8, 0.7, 0.6, 0.55", \
            "1.5, 0.9, 0.8, 0.65, 0.6", \
            "1.5, 0.9, 0.8, 0.65, 0.6", \
            "1.5, 0.9, 0.8, 0.65, 0.6", \
            "1.5, 0.9, 0.8, 0.65, 0.6");
             }
             noise_immunity_below_low (my_noise_reject_outside_rail) {
                 values ("1, 0.8, 0.5", \
            "1, 0.8, 0.5", \
            "1, 0.8, 0.5");
             }
             noise_immunity_above_high (my_noise_reject_outside_rail) {
                 values ("1, 0.8, 0.5", \
            "1, 0.8, 0.5", \
            "1, 0.8, 0.5");
             }
             /* Propagated noise.
             * A function of input noise width, height, and output
             * capacitance. Width and height are in separate tables. */
             propagated_noise_width_high(my_propagated_noise) {
                 values ("0.01, 0.10", "0.15, 0.04", "0.14, 0.18", \
            "0.05, 0.15", "0.24, 0.07", "0.17, 0.32");
             }
             propagated_noise_height_high(my_propagated_noise) {
                 values ("0.01, 0.10", "0.15, 0.04", "0.14, 0.18", \
            "0.05, 0.15", "0.24, 0.07", "0.17, 0.32");
             }
             propagated_noise_width_low(my_propagated_noise) {
                 values ("0.01, 0.10", "0.15, 0.04", "0.14, 0.18", \
            "0.05, 0.15", "0.24, 0.07", "0.17, 0.32");
             }
             propagated_noise_height_low(my_propagated_noise) {
                 values ("0.01, 0.10", "0.15, 0.04", "0.14, 0.18", \
            "0.05, 0.15", "0.24, 0.07", "0.17, 0.32");
             }
             propagated_noise_width_above_high(my_propagated_noise) {
                 values ("0.01, 0.10", "0.15, 0.04", "0.14, 0.18", \
            "0.05, 0.15", "0.24, 0.07", "0.17, 0.32");
             }
             propagated_noise_height_above_high(my_propagated_noise) {
                 values ("0.01, 0.10", "0.15, 0.04", "0.14, 0.18", \
            "0.05, 0.15", "0.24, 0.07", "0.17, 0.32");
             }
             propagated_noise_width_below_low(my_propagated_noise) {
                 values ("0.01, 0.10", "0.15, 0.04", "0.14, 0.18", \
            "0.05, 0.15", "0.24, 0.07", "0.17, 0.32");
             }
             propagated_noise_height_below_low(my_propagated_noise) {
                 values ("0.01, 0.10", "0.15, 0.04", "0.14, 0.18", \
            "0.05, 0.15", "0.24, 0.07", "0.17, 0.32");
             } /*end propagated noise groups */
    } /* end of timing group */
  } /* end of pin (Y)( */
} /* end of cell (INV) */
```

```
} /* end of library (my_noise_lib)
```

Checking Library Noise Data

You can use the noise data screener to check for errors in your library noise data. The screener checks the following data: DC noise margin, IV curves, noise immunity curve, and noise propagation data. For information about using the noise data screener, see “Noise Data Screener” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

8

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 can capture essential noise properties of digital circuits using a compact library representation. It enables fast and accurate gate-level noise analysis while maintaining a relatively simple library characterization. CCS noise modeling supports noise combination and driver weakening.

CCS noise is characterization data that provides information for noise failure detection on cell inputs, calculation of noise bumps on cell outputs, and noise propagation through the cell. For the best accuracy, you must add CCS timing data to the library in addition to the CCS noise data. The CCS noise data includes the following:

- Channel-connected block parameters
- DC current tables
- Timing tables for rising and falling transitions
- Timing tables for low and high propagated noise

Compiling a Library With CCS Signal Integrity Information

When you compile a library containing CCS noise information with the `read_lib` command in Library Compiler (after screening data without finding any errors), Library Compiler extracts the required information from the CCS noise data in the library.

Often you only need to screen the CCS noise information in the libraries to see if the data is acceptable. This involves including or excluding the noise compilation. The `lc_disable_ccs_noise_extraction` environment variable enables you to turn on or off the compilation of CCS noise information.

By default, `lc_disable_ccs_noise_extraction` is set to 0, which means that Library Compiler screens and extracts CCS noise information in the libraries with the `read_lib` command.

If `lc_disable_ccs_noise_extraction` is set to 1 before reading in a library with the `read_lib` command, the CCS noise information is screened but not extracted into the compiled library database. As a result, the compiled library database does not contain the CCS noise information.

CCS Signal Integrity Modeling Syntax

```
library (name) {
 ...
 lu_table_template(dc_template_name) {
    variable_1 : input_voltage;
    variable_2 : output_voltage;
 }
 lu_table_template(output_voltage_template_name) {
    variable_1 : input_net_transition;
    variable_2 : total_output_net_capacitance;
    variable_3 : time;
 }
 lu_table_template(propagated_noise_template_name) {
    variable_1 : input_noise_height;
    variable_2 : input_noise_width;
    variable_3 : total_output_net_capacitance;
    variable_4 : time;
 }
  cell (name) {
    pin (name) {
      ...
      ccsn_first_stage () {
           is_needed : boolean;
           is_inverting : boolean;
           stage_type : stage_type_value;
           miller_cap_rise : float;
           miller_cap_fall : float;
           dc_current (dc_current_template)
                index_1("float, ...");
                index_2("float, ...");
                values("float, ...");

           }

           output_voltage_rise ( )
                vector (output_voltage_template_name) {
                   index_1(float);
                   index_2(float);
                   index_3("float, ...");
                   values("float, ...");
                 }
                 ...
            }
           output_voltage_fall ( ) {
                vector (output_voltage_template_name) {
                   index_1(float);
                   index_2(float);
                   index_3("float, ...");
                   values("float, ...");
                }
                ...
```

```
      }
      propagated_noise_low ( ) {
            vector (propagated_noise_template_name) {
               index_1(float);
               index_2(float);
               index_3(float);
               index_4("float, ...");
               values("float, ...");
            }
             ...
      }
      propagated_noise_high ( ) {
            vector (propagated_noise_template_name) {
               index_1(float);
               index_2(float);
               index_3(float);
               index_4("float, ...");
               values("float, ...");
            }
            ...
      }
      when : boolean expression;
} /* ccsn_first_stage */
ccsn_last_stage () {
      is_needed : boolean;
      is_inverting : boolean;
      stage_type : stage_type_value;
      miller_cap_rise : float;
      miller_cap_fall : float;
      dc_current (dc_current_template)
            index_1("float, ...");
            index_2("float, ...");
            values("float, ...");

      }
      output_voltage_rise ( )
            vector (output_voltage_template_name) {
               index_1(float);
               index_2(float);
               index_3("float, ...");
               values("float, ...");
             }
             ...
       }
      output_voltage_fall ( ) {
            vector (output_voltage_template_name) {
               index_1(float);
               index_2(float);
               index_3("float, ...");
               values("float, ...");
            }
            ...
```

```
      }
      propagated_noise_low ( ) {
            vector (propagated_noise_template_name) {
               index_1(float);
               index_2(float);
               index_3(float);
               index_4("float, ...");
               values("float, ...");
            }
             ...
      }
      propagated_noise_high ( ) {
            vector (propagated_noise_template_name) {
               index_1(float);
               index_2(float);
               index_3(float);
               index_4("float, ...");
               values("float, ...");
            }
            ...
      }
      when : boolean expression;
    } /* ccsn_last_stage */
...
    timing() {
  ...
       ccsn_first_stage () {
      is_needed : boolean;
      is_inverting : boolean;
      stage_type : stage_type_value;
      miller_cap_rise : float;
      miller_cap_fall : float;
      dc_current (dc_current_template)
            index_1("float, ...");
            index_2("float, ...");
            values("float, ...");

      }

      output_voltage_rise ( )
            vector (output_voltage_template_name) {
               index_1(float);
               index_2(float);
               index_3("float, ...");
               values("float, ...");
             }
             ...
       }
      output_voltage_fall ( ) {
            vector (output_voltage_template_name) {
               index_1(float);
               index_2(float);
```

```
            index_3("float, ...");
            values("float, ...");
         }
         ...

   }
   propagated_noise_low ( ) {
         vector (propagated_noise_template_name) {
            index_1(float);
            index_2(float);
            index_3(float);
            index_4("float, ...");
            values("float, ...");
         }
          ...
   }
   propagated_noise_high ( ) {
         vector (propagated_noise_template_name) {
            index_1(float);
            index_2(float);
            index_3(float);
            index_4("float, ...");
            values("float, ...");
         }
         ...
   }
   when : boolean expression;
   } /* ccsn_first_stage */
   ccsn_last_stage () {
   is_needed : boolean;
   is_inverting : boolean;
   stage_type : stage_type_value;
   miller_cap_rise : float;
   miller_cap_fall : float;
   dc_current (dc_current_template)
         index_1("float, ...");
         index_2("float, ...");
         values("float, ...");

   }

   output_voltage_rise ( )
         vector (output_voltage_template_name) {
            index_1(float);
            index_2(float);
            index_3("float, ...");
            values("float, ...");
          }
          ...
    }
   output_voltage_fall ( ) {
         vector (output_voltage_template_name) {
            index_1(float);
```

```
                  index_2(float);
                  index_3("float, ...");
                  values("float, ...");
            }
            ...

      }
      propagated_noise_low ( ) {
            vector (propagated_noise_template_name) {
               index_1(float);
               index_2(float);
               index_3(float);
               index_4("float, ...");
               values("float, ...");
            }
             ...
      }
      propagated_noise_high ( ) {
            vector (propagated_noise_template_name) {
               index_1(float);
               index_2(float);
               index_3(float);
               index_4("float, ...");
               values("float, ...");
            }
            ...
      }
      when : boolean expression;
       } /* 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 channel-connected block attributes
- The dc_current group, which contains a two-dimensional DC current table
- The output_current_rise and output_current_fall groups, which contain two timing tables for rising and falling transitions.
- The propagated_noise_low and propagated_noise_high groups, which contain two noise tables for low and high propagated noise.

Note:

If the ccsn_first_stage and ccsn_last_stage groups are defined at the pin level, the ccsn_first_stage group can be defined only in an input pin or inout pin, and the ccsn_last_stage group can be defined only in an output pin or inout pin.

is_needed Attribute

The `is_needed` Boolean attribute determines whether the `dc_current`, `output_current_rise`, `output_current_fall`, `propagated_noise_low`, and `propagated_noise_high` channel-connected block attributes should be specified to include CCS noise data for a cell. The `is_needed` attribute is defined inside the `ccsn_first_stage` and `ccsn_last_stage` groups.

By default, the `is_needed` attribute is set to true, which means that CCS noise data is included in the `ccsn_first_stage` and `ccsn_last_stage` groups for the cell. The `is_needed` attribute should be set to false for cells that do not need a current-based driver model, such as diodes, antennas, and cload cells. When the attribute is set to false, CCS noise data, enabled by the channel-connected block attributes, is not included in the `ccsn_first_stage` and `ccsn_last_stage` groups.

is_inverting Attribute

The `is_inverting` attribute specifies whether the channel-connecting block is inverting. If the channel-connecting 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 may consist of multiple channel-connecting blocks.

stage_type Attribute

The `stage_type` attribute specifies the channel-connecting block’s output voltage stage type. The valid values are `pull_up`, which causes the channel-connecting block’s output voltage to be pulled up or to rise; `pull_down`, which causes the channel-connecting block’s output voltage to be pulled down or to fall; and `both`, which causes the channel-connecting 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 or falling channel-connecting 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.

dc_current Group

The `dc_current` group specifies the input and output voltage values of a two-dimensional current table for a channel-connecting 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-connecting block DC current values, in library current units, that are measured at the channel-connecting 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-connecting 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-connecting block output node.

propagated_noise_high and propagated_noise_low Groups

The `propagated_noise_low` and `propagated_noise_high` groups use `vector` groups to specify the three-dimensional `output_voltage` tables of the channel-connecting 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-connecting block output node.

when Attribute

The `when` attribute specifies the condition under which the channel-connecting block data is applied. The attribute is defined in the `ccsn_first_stage` and `ccsn_last_stage` groups both at the pin level and the timing level.

Checking CCS Signal Integrity Models

Library Compiler automatically checks the syntax for CCS signal integrity models and reports any problems it encounters. For information about the types of checks Library Compiler performs for CCS signal integrity models, see "Library Screener Checks" in the "Reading and Compiling Libraries" chapter in the *Library Quality Assurance System User Guide*.

CCS Noise Library Example

The following is a sample CCS noise library.

Example 8-1 CCS Noise Library

```
library (CCS_noise) {

  technology ( cmos ) ;
  delay_model     : table_lookup;
  time_unit : "1ps" ;
  leakage_power_unit : "1pW" ;
  voltage_unit : "1V" ;
  current_unit : "1uA" ;
  pulling_resistance_unit : "1kohm" ;
  capacitive_load_unit(1000.000,ff) ;

  nom_voltage     : 1.200;
  nom_temperature : 25.000;
  nom_process     : 1.000;

  operating_conditions("OC1") {
    process :  1.000;
    temperature :  25.000;
    voltage :  1.200;
    tree_type :  "balanced_tree";
  }
  default_operating_conditions:OC1;

  lu_table_template(del_0_5_7_t) {
   variable_1 : input_net_transition;
   index_1("10.000, 175.000, 455.000, 980.000, 2100.000");
   variable_2 : total_output_net_capacitance;
   index_2("0.000000, 0.004000, 0.007000, 0.019000, 0.040000, 0.075000,
0.175000");
  }

  lu_table_template(ccsn_dc_29x29) {
   variable_1 : input_voltage;
   variable_2 : output_voltage;
  }

  lu_table_template(ccsn_timing_lut_5) {
    variable_1 : input_net_transition;
    variable_2 : total_output_net_capacitance;
    variable_3 : time;
  }

  lu_table_template(ccsn_prop_lut_5) {
    variable_1 : input_noise_height;
    variable_2 : input_noise_width;
    variable_3 : total_output_net_capacitance;
    variable_4 : time;
  }

  lu_table_template(lu_table_template7x9) {
    variable_1 : input_net_transition;
```

```
    variable_2 : voltage;
  }
cell(inv) {
    area : 0.75;
    pin(I) {
      direction : input;
      max_transition : 2100.0;
      capacitance : 0.002000;
      fanout_load : 1;
    }
    pin(Z) {
      direction : output;
      max_capacitance : 0.175000;
      max_fanout : 58;
      max_transition : 1400.0;
               function  : "(I)'";
      timing() {
        related_pin     : "I";
        timing_sense    : negative_unate;
                      ...
        ccsn_first_stage ( ) {
          is_needed     : true;
          is_inverting    : true;
          stage_type      : both;
          miller_cap_rise : 0.00055;
          miller_cap_fall : 0.00084;

          dc_current (ccsn_dc_29x29) {
            index_1 ("-1.200, -0.600, -0.240, -0.120, 0.000, 0.060, 0.120, 0.180, \
                      0.240, 0.300, 0.360, 0.420, 0.480, 0.540, 0.600, 0.660, \
                      0.720, 0.780, 0.840, 0.900, 0.960, 1.020, 1.080, 1.140, \
                      1.200, 1.320, 1.440, 1.800, 2.400");
            index_2 ("-1.200, -0.600, -0.240, -0.120, 0.000, 0.060, 0.120, 0.180, \
                      0.240, 0.300, 0.360, 0.420, 0.480, 0.540, 0.600, 0.660, \
                      0.720, 0.780, 0.840, 0.900, 0.960, 1.020, 1.080, 1.140, \
                      1.200, 1.320, 1.440, 1.800, 2.400");
            values   ("619.332000, 0.548416, 0.510134, 0.491965, 0.470368, \
                                                      ...
                                      -0.390604, -0.394495, -0.403571, -579.968000");
          }
output_voltage_rise ( ) {
            vector (ccsn_timing_lut_5) {
              index_1(175.000);
              index_2(0.004000);
              index_3 ("104.222, 127.996, 144.729, 159.367, 176.983");
              values   ("1.080, 0.840, 0.600, 0.360, 0.120");
            }
                          ...
}


          output_voltage_fall ( ) {
            vector (ccsn_timing_lut_5) {
              index_1(175.000);
              index_2(0.004000);
              index_3 ("104.222, 127.996, 144.729, 159.367, 176.983");
              values   ("1.080, 0.840, 0.600, 0.360, 0.120");
            }
```

```
                          ...
}

          propagated_noise_low ( ) {
            vector (ccsn_prop_lut_5) {
              index_1(0.6000);
              index_2(1365.00);
              index_3(0.004000);
              index_4 ("640.90, 679.55, 711.76, 755.45, 793.68");
              values  ("0.0553, 0.0884, 0.1105, 0.0884, 0.0553");
            }
                          ...
                          vector (ccsn_prop_lut_5) {
              index_1(0.6500);
              index_2(2730.00);
              index_3(0.019000);
              index_4 ("1298.73, 1379.15, 1484.78, 1599.75, 1687.38");
              values   ("0.0927, 0.1483, 0.1854, 0.1483, 0.0927");
            }
          }

          propagated_noise_high ( ) {
            vector (ccsn_prop_lut_5) {
              index_1(0.6000);
              index_2(1365.00);
              index_3(0.004000);
              index_4 ("648.77, 688.99, 741.96, 793.08, 833.85");
              values   ("1.0592, 0.9748, 0.9184, 0.9748, 1.0592");
            }
                          ...
            vector (ccsn_prop_lut_5) {
              index_1(0.6500);
              index_2(2730.00);
              index_3(0.019000);
              index_4 ("1307.15, 1404.92, 1561.13, 1709.43, 1814.30");
              values   ("1.0028, 0.8844, 0.8055, 0.8844, 1.0028");
            }
          }
        } /* ccsn_first_stage */

      } /* timing I -> Z */
    } /* Z */
  }  /* cell(inv) */

} /* library *
```

Conditional Data Modeling in CCS Noise Models

Library Compiler supports conditional data modeling in pin-based CCS noise models. The `mode` and `when` attributes are provided in the CCS noise groups to support this feature:

- The `when` attribute in pin-based CCS noise models (in the `ccsn_first_stage` and `ccsn_last_stage` groups).
- The `mode` attribute in pin-based CCS noise data modeling.

Library Compiler provides mode support for pin-based CCS noise data modeling, as shown in the following syntax:

```
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 :<boolean>;
      when :  <boolean expression>;
      mode (mode_name, mode_value);
      …
    }
    …
    /* ccs noise first stage for Condition n */
    ccsn_first_stage() {
      is_needed :<boolean>;
      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 :<boolean>;
      when :  <boolean expression>;
      mode (mode_name, mode_value);
      …
    }
    …
    /* ccs noise last stage for Condition n */
    ccsn_last_stage() {
      is_needed :<boolean>;
      when :  <boolean expression>;
      mode (mode_name, mode_value);
      …
    }
    timing() {
      …
      /* following are arc-based ccs noise */
      ccsn_first_stage() {
          is_needed :<boolean>;
          …
      }
```

```
      …
      ccsn_last_stage() {
          is_needed :<boolean>;
          …
      }
    }
  }
}
```

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 */
      ccs_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 */
      ccs_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

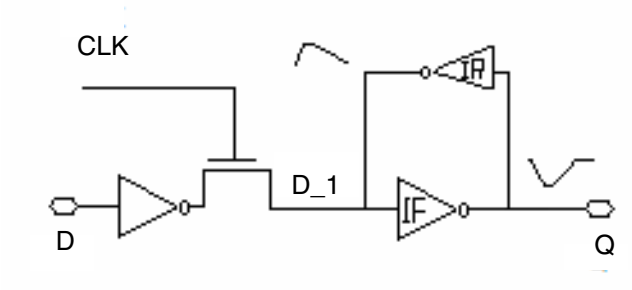
Unbuffered input and output latches are a special type of cell that has an internal memory node connected to an input or output pin. In order to increase the speed of the design and lower power consumption, these cells do not use inverters.

Figure 8-1 and Figure 8-2 show the schematics of a typical unbuffered output latch and an unbuffered input latch, respectively. The major difference between an unbuffered output cell and unbuffered input cell and a regular cell is as follows:

- Unbuffered Output Cell

  An unbuffered output cell has the *feedback*, or back-driving path, from the unbuffered output pin to an internal node. In Figure 8-1, Q is connected to internal node D_1 through the IR inverter.

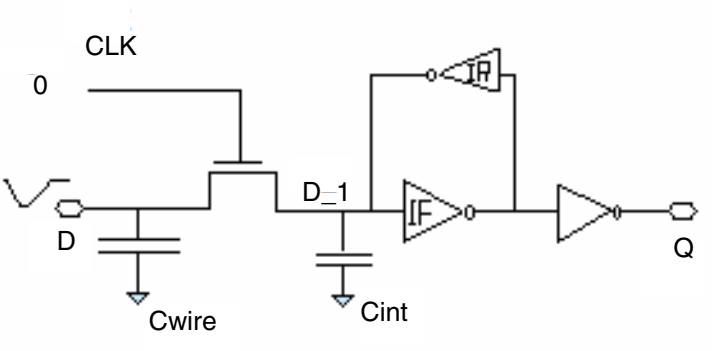
Figure 8-1 Unbuffered Output Latch



- Unbuffered Input Cell

  The input pin of an unbuffered cell is not buffered and can be connected through a pass gate to the internal node. (A pass gate is a special gate that has an input and an output and a control input. If the control is set to true, the output is driven by the input. Otherwise, it floats.) For example, in Figure 8-2, D is connected to internal node D_1 through a pass gate.

Figure 8-2 Unbuffered Input Latch



To correctly model this category of cells in Liberty syntax, you must determine:

- If a pin is buffered or unbuffered.
- If a pin is implemented with a pass gate.
- If the `ccsn_*_stage` information models a pass gate.

Syntax for Unbuffered Output Latches

The following syntax supports unbuffered output latches.

Syntax

```
/* unbuffered output pin */
```

```
  pin (<pin_name>) {
   direction : inout/output;
   is_unbuffered : true | false ;
   has_pass_gate : true | false ;
    ccsn_first_stage () {
     is_pass_gate : true | false;
     …
   }
     …
    ccsn_last_stage () {
     is_pass_gate : true | false;
     …
   }
     …
   timing() {
    ccsn_first_stage () {
     is_pass_gate : true | false;
     …
   }
     …
   ccsn_last_stage () {
     is_pass_gate : true | false;
     …
   }
   …
  }
}
pin (<pin_name>) {
  direction : input/inout;
  is_unbuffered : true | false ;
  has_pass_gate : true | false ;
  ccsn_first_stage () {
    is_pass_gate : true | false;
   …
   }
   …
   ccsn_last_stage () {
     is_pass_gate : true | false;
     …
     }
    …
    timing() {
        ccsn_first_stage () {
          is_pass_gate : true | false;
          …
         }
         …
         ccsn_last_stage () {
           is_pass_gate : true | false;
           …
         }
         …
    }
```

```
}
```

Pin-Level Attributes

The following attributes are pin-level attributes for unbuffered output latches.

is_unbuffered Attribute

The `is_unbuffered` simple Boolean attribute identifies whether the pin is unbuffered. This optional attribute can be specified on the pins of any library cell. The default value 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 is not new in this release. However, the syntax has been extended to model 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 value is false.

9

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

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 9-4. The leakage current syntax is as follows:

Example 9-1 Leakage Current Syntax

```
cell(<cell_name>) {
  ...
  leakage_current() {
    when : <boolean expression>;
    pg_current(<pg_pin_name>) {
      value : <float>;
    }
    ...
  leakage_current() { /* without the when statement */
    /* default state */
    ...
  }
}
```

Current conservation means that the sum of all current values must be zero. A positive value means power pin current, and a negative value means ground pin current.

If you have two power and ground pins in your design, and you have already specified the power current value to 2.0, you do not have to specify the ground current value, because the tool will infer that it must be -2.0 based on current conservation.

For multiple power and ground pins, you must use the regular format because it provides `pg_current`, which allows you to specify the power and ground names. For example, if you have two power pins, you must specify the value for each pin.

Again, a simplified format is allowed for a cell with a single power and ground pin. For this case, no `pg_current` group is required within a `leakage_current` group.

Example 9-2 Leakage Current Format Simplified

```
cell(<cell_name>) {
  ...
  leakage_current() { /* without pg_current group*/
    when : <boolean expression>;
    value : <float>;
  }

  leakage_current() { /* without the when statement */
    /* default state */
    ...
  }
}
```

Checking Leakage Current Syntax

Library Compiler automatically checks the syntax for leakage current and reports any problems it encounters. For information about the types of checks Library Compiler performs for leakage current, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Gate Leakage Modeling in Leakage Current

The syntax for these power models is described in the following section.

Syntax

```
cell(<cell_name>) {
  ...
  leakage_current() {
    when : <boolean expression>;
    pg_current(<a pg pin name>) {
       value : <float>;
}
```

```
...
gate_leakage(<an input pin name>) {
   input_low_value : <float>;
   input_high_value : <float>;
}
...
  }
  ...
  leakage_current() {
  /* group without when statement */
    /* default state */
    ...
  }
}
```

gate_leakage Group

This group specifies the cell's gate leakage current on input or inout pins within the `leakage_current` group in a cell. For information about cell leakage, see "Cell Leakage Current" on page 9-2.

The following information pertains to a `gate_leakage` group:

- Groups can be placed in any order if there are more than one `gate_leakage` groups within a `leakage_current` group.
- Leakage current of a cell is characterized with opened outputs, which means outputs of a modeling cell do not drive any other cells. Outputs are assumed to have zero static current during the measurement.
- A missing `gate_leakage` group is allowed for certain pins.
- Current conservation is applicable if it can be applied to higher error tolerance.

input_low_value Attribute

This attribute specifies gate leakage current on an input or inout pin when the pin is in a `low` state.

- A negative float value is required.
- The gate leakage current is measured from the power pin of the cell to the ground pin of its driver cell.
- The input pin is pulled low.
- The `input_low_value` is not required for a `gate_leakage` group.
- Defaults to 0 if no `gate_leakage` group is specified for certain pins.
- Defaults to 0 if no `input_low_value` is specified in gate_leakage group.

input_high_value Attribute

This attribute specifies gate leakage current on an input or inout pin when the pin is in a `high` state.

- A positive float value is required.
- The gate leakage current is measured from the power pin of its driver cell to the ground pin of the cell.
- The input pin is pulled high.
- The `input_high_value` is not required for a `gate_leakage` group.
- Defaults to 0 if no `gate_leakage` groups is specified for certain pins.
- Defaults to 0 if no `input_high_value` is specified in the `gate_leakage` group.

Checking Gate Leakage Current Syntax

Library Compiler automatically checks the conditions under a `leakage_current` group and reports any problems it encounters. For information about the types of checks Library Compiler performs for leakage current, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Intrinsic Parasitic Models

You can use the syntax in Example 9-3 for intrinsic parasitic models. The syntax consists of two parts: one is intrinsic resistance and the other is intrinsic capacitance.

Example 9-3 Intrinsic Parasitic Model

```
cell (<cell_name> ) {
  mode_definition (<mode_name>) {
    mode_value(namestring) {
    when : <boolean expression>;
    sdf_cond : <boolean expression>;
  }
  ...
  intrinsic_parasitic() {
    mode (mode_name, mode_value);
    when : <boolean expression>;
    intrinsic_resistance(<pg_pin_name>) {
      related_output : <output_pin_name>;
      value : < float>;
    }
    intrinsic_capacitance(<pg_pin_name>) {
      value : <float>;
    }
  }

  intrinsic_parasitic() {
without when statement */
    /* default state */

  }
}
```

Example 9-4 Conditional Data Modeling for Intrinsic Parasitic Model By Mode

```
library (csm13os120_typ) {
   technology ( cmos ) ;
   delay_model : table_lookup;
   lu_table_template(ccsn_dc_29x29) {
     variable_1 : input_voltage;
     variable_2 : output_voltage;
   }
   cell(inv0d0) {
     area : 0.75;
     pg_pin(V1) {
        voltage_name : VDD1;
        pg_type : primary_power;
     }
     pg_pin(G1) {
        voltage_name : GND1;
        pg_type : primary_ground;
     }
     mode_definition(rw) {
       mode_value(read) {
       when : "A1";
       sdf_cond : "A1 == 1";
     }
       mode_value(write) {
       when : "!A1";
```

```
        sdf_cond : "A1 == 0";
        }
      }
      pin(A1) {
        direction : input;
        capacitance : 0.1 ;
        related_power_pin : V1;
        related_ground_pin : G1;
      }
      pin(A2) {
        direction : input;
        capacitance : 0.1 ;
        related_power_pin : V1;
        related_ground_pin : G1;
      }
      pin(ZN) {
        direction : output;
        max_capacitance : 0.1;
        function :  "!A1+A2";
        related_power_pin : V1;
        related_ground_pin : G1;
        timing() {
         timing_sense : "negative_unate"
           related_pin : "A1”;
           ...
            }
         }
      pin(ZN1) {
        direction : output;
        max_capacitance : 0.1;
        function :  "!A1";
        related_power_pin : V1;
        related_ground_pin : G1;
        timing() {
         timing_sense : "negative_unate"
         related_pin : "A1 A2”;
       ...

           }
         }
        intrinsic_parasitic() {
               mode(rw, read);
               intrinsic_resistance(G1) {
               related_output : "ZN";
                  value : 9.0;
               }
               intrinsic_capacitance(G1) {
                  value : 8.2;
               }
         }
} /* cell(inv0d0) */
} /* library
```

Checking Intrinsic Parasitic Syntax

Library Compiler automatically checks intrinsic parasitic model syntax and reports any problems it encounters. For information about the types of checks Library Compiler performs for intrinsic parasitic syntax, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Parasitics Modeling in Macro Cells

For macro cells, the `total_capacitance` group is provided within the `intrinsic_parasitic` group.

total_capacitance Group

The `total_capacitance` group specifies the macro cell’s total capacitance on a power or ground net within the `intrinsic_parasitic` group.

- This group can be placed in any order if there is more than one `total_capacitance` group within an `intrinsic_parasitic` group.
- If the `total_capacitance` group is not defined for a certain power and ground pin, the value of capacitance defaults to 0.0. The default value is provided by the tool.
- The parasitics modeling of total capacitance in macros cells is not state dependent. This means that there is no state condition specified in `intrinsic_parasitic`.

Parasitics Modeling Syntax

```
cell (<cell_name> ) {
   power_cell_type : <enum(stdcell, macro)>;
   ...
   intrinsic_parasitic() {
      total_capacitance(<a pg pin name>) {
        value : <float>;
      }
      ...
   }
   ...
}
```

Checking Parasitic Model Syntax

Library Compiler automatically checks parasitic model syntax and reports any problems it encounters. For information about the types of checks Library Compiler performs for parasitic model syntax, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Dynamic Power

Because CCS power is current-based data, instantaneous power data on the power or ground pin is captured instead of internal energy specified in the nonlinear power model format. The current-based data provides higher accuracy than the existing model.

In the CCS modeling format, instantaneous power data is specified as a table of current waveforms. The table is dependent on the transition time of a toggling input and the capacitance of the toggling outputs.

As the number of output pins increases in a cell, the number of waveform tables becomes large. However, the cell with multiple output pins (more than one output) does not need to be characterized for all possible output load combinations. Therefore, two types of methods can be introduced to simplify the captured data.

- Cross type - Only one output capacitance is swept, while all other output capacitances are held in a typical value or fixed value.
- Diagonal type - The capacitance to all the output pins is swept together by an identical value.

A table that is modeled based on these two types is defined as a sparse table. Otherwise it is defined as a dense table, meaning that all combinations of the output load variable are specified in tables.

Dynamic Power and Ground Current Table Syntax

You can use the following syntax for dynamic current:

```
pg_current_template(<template_name_1>) {
  variable_1 : input_net_transition;
  variable_2 : total_output_net_capacitance;
  variable_3 : time;
  index_1(<float>, );                 /* optional */
  index_2(<float>, );                 /* optional */
  index_3(<float>, );                 /* optional */
}

pg_current_template(<template_name_2>) {
  variable_1 : input_net_transition;
  variable_2 : total_output_net_capacitance;
  variable_3 : total_output_net_capacitance;
  variable_4 : time;
  index_1(<float>, );                /* optional */
  index_2(<float>, );                /* optional */
  index_3(<float>, );                /* optional */
  index_4(<float>, );                /* optional */
}
```

Dynamic Power Modeling in Macro Cells

The extensions to CCS dynamic power format provides more accurate models for macro cells. The current dynamic power model only supports current waveforms for single-input events.

The model can also be applied to memory modeling with synchronous events, which are triggered by toggling either a single `read_enable` or `write_enable`.

However, for asynchronous event, the read access can be triggered by more than one bit of the address bus toggling. To support asynchronous memory access for macro cells, use `min_input_switching_count` and `max_input_switching_count`, in dynamic power as shown in the next section.

The following syntax for dynamic power format provides more accurate models for macro cells:

```
...
cell(<cell_name>) {
mode_definition (<mode_name>) {
      mode_value(namestring) {
        when : <boolean expression>;
        sdf_cond : <boolean expression>;
      }
 ...
  power_cell_type : <enum(stdcell, macro)>
  dynamic_current() {
```

```
      mode (mode_name, mode_value);
      when : <boolean expression>
      related_inputs : <input_pin_name>;
      switching_group() {
           min_input_switching_count : <integer>;
           max_input_switching_count :<integer>;
           pg_current(<pg_pin_name>) {
                vector(<template_name>) {
                     reference_time : <float>;
                     index_1(<float>);
                     index_2("<float>,…");
                     values("<float>, …");
                }     /* end vector group*/
                ...
            }     /end pg_current group */
              ...
        }  /* end switching_group */
        ..
  } /* end dynamic_current group*/
 ..,
} /* end cell group*/
...
```

The `min_input_switching_count` and `max_input_switching_count` attributes specify the number of bits in the input bus that are switching simultaneously while an asynchronous event occurs.

A single switching bit can be defined by setting the same value in both attributes. The following example shows that any three bits specified in `related_inputs` are switching simultaneously.

```
...
min_input_switching_count : 3;
max_input_switching_count : 3;
...
```

A range of switching bits can be defined by setting the minimum and maximum value. The following example shows that any 2, 3, 4 or 5 bits specified in `related_inputs` are switching simultaneously.

```
 ...
 min_input_switching_count : 2;
 max_input_switching_count : 5;
 ...
```

min_input_switching_count Attribute

This attribute specifies the minimum number of bits in an input bus that are switching simultaneously.

- The count must be integer.

- The count must greater than 0 and less than `max_input_switching_count`.

max_input_switching_count Attribute

This attribute specifies the maximum number of bits in an input bus that are switching simultaneously.

- The count must be integer.
- The count must greater than `min_input_switching_count`.
- The count must be less than the total number of bits listed in `related_inputs`.

Checking Dynamic Power Model Syntax

Library Compiler automatically checks dynamic power model syntax and reports any problems it encounters. For information about the types of checks Library Compiler performs for dynamic power models, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Examples for CCS Dynamic Power for Macro Cells

```
...
pg_current_template ( CCS_power_1 ) {
   variable_1 : input_net_transition ;
   variable_2 : time;
   }
   type(bus3) {
      base_type : array;
         bit_width : 3;
...
   }
...
cell ( example ) {
   bus(addr_in) {
   bus_type : bus3;
   direction : input;
   ...
}
   pin(data_in) {
   direction : input;
                                                    …
}
   ...
power_cell_type : macro;
   dynamic_current() {
      when: "!WE";
      related_inputs : "addr_in";
         switching_group ( ) {
            min_input_switching_count : 1;
            max_input_switching_count : 3;
```

```
            pg_current (VSS) {
               vector ( CCS_power_1 ) {
               reference_time : 0.01;
               index_1 ( "0.01" )
               index_2 ( "4.6, 5.9, 6.2, 7.3")
               values ( "0.002, 0.009, 0.134, 0.546")
            }
               ...
               vector ( CCS_power_1 ) {
               reference_time : 0.01;
               index_1 ( "0.03" )
               index_2 ( "2.4, 2.6, 2.9, 4.0" )
               values ( "0.012, 0.109, 0.534, 0.746")
            }
               vector ( CCS_power_1 ) {
               reference_time : 0.01;
               index_1 ( "0.08" )
               index_2 ( "1.0, 1.6, 1.8, 1.9" )
               values ( "0.102, 0.209, 0.474, 0.992")
            }
               ...
         }    /* pg_current */
         ...
      } /* switching_group */
      ...
   } /* dynamic_current */
   ...
   ...
   intrinsic_parasitic() {
      total_capacitance(VDD) {
      value : 0.2;
      }
                                                 …
   }  /* intrinsic_parasitic */
   ...
   ...
   leakage_current() {
      when : WE;
      gate_leakage(data_in) {
      input_low_value : -0.3;
      input_high_value : 0.5;
      }
      ...
   }  /* leakage_current */
   ...
   }   /* end of cell */
   ...
```

Conditional Data Modeling for Dynamic Current Model By Mode Example

```
library (csm13os120_typ) {
```

```
  technology ( cmos ) ;
  delay_model : table_lookup;
  lu_table_template(ccsn_dc_29x29) {
    variable_1 : input_voltage;
    variable_2 : output_voltage;
  }
  cell(inv0d0) {
    area : 0.75;
    pg_pin(V1) {
      voltage_name : VDD1;
      pg_type : primary_power;
    }
    pg_pin(G1) {
      voltage_name : GND1;
      pg_type : primary_ground;
    }
    mode_definition(rw) {
      mode_value(read) {
        when : "A1";
        sdf_cond : "A1 == 1";
      }
      mode_value(write) {
        when : "!A1";
        sdf_cond : "A1 == 0";
      }
    }
    power_cell_type : stdcell;
    dynamic_current() { /* dense table */
       mode(rw, read);
       related_inputs : "A2";
       related_outputs : "ZN ZN1";
       switching_group() {
         output_switching_condition(rise rise);
         pg_current(V1) {
            vector(test_1) {
              reference_time : 23.7;
              index_1("0.8");
              index_2("0.7");
              index_3("10.4");
              index_4("8.2 8.5 9.1 9.4 9.8");
              values("0.7 34.6 3.78 92.4 100.1");
          }
               ...
           }
    }
}
pin(A1) {
  direction : input;
  capacitance : 0.1 ;
  related_power_pin : V1;
  related_ground_pin : G1;
}
pin(A2) {
```

```
  direction : input;
  capacitance : 0.1 ;
  related_power_pin : V1;
  related_ground_pin : G1;
}
pin(ZN) {
  direction : output;
  max_capacitance : 0.1;
  function :  "!A1+A2";
  related_power_pin : V1;
  related_ground_pin : G1;
   timing() {
      timing_sense : "negative_unate"
      related_pin : "A1”;
      …
            }
       }
pin(ZN1) {
  direction : output;
  max_capacitance : 0.1;
  function :  "!A1";
  related_power_pin : V1;
  related_ground_pin : G1;
  timing() {
      timing_sense : "negative_unate"
      related_pin : "A1 A2”;
      …
            }
       }
} /* cell(inv0d0) */
} /* library
```

Checking Power and Ground Current Syntax

Library Compiler automatically checks power and ground current template syntax and reports any problems it encounters. For information about the types of checks Library Compiler performs for power and ground current syntax, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Dynamic Current Syntax

The syntax in Example 9-5 is used for instantaneous power data, which is captured at the cell level.

Example 9-5 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 */
```

Checking Dynamic Current Syntax

Library Compiler automatically checks dynamic current syntax and reports any problems it encounters. For information about the types of checks Library Compiler performs for dynamic current syntax, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Compact CCS Power Modeling

CCS power compaction uses base curve technology to significantly reduce the library size of CCS power libraries. Greater control of the Liberty file size allows you to include additional data points and more accurately capture CCS power data for dynamic current waveforms.

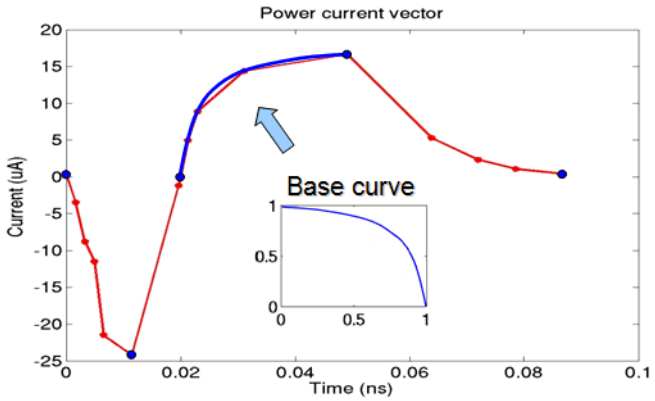
Base curve technology was first introduced for compact CCS timing. Each timing current waveform is split into two segments in the I-V domain, and the shape of each segment is modeled by a base curve that has a similar shape. This segmentation prevents direct modeling with the piecewise linear data points.

Compact CCS power modeling differs from compact CCS timing modeling in that power waveforms can include both positive sections and negative sections. Therefore, they must be modeled in an I(t) domain. In addition, the segmentation is more flexible. This is important because the current waveform shape may 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 9-1, the red curve shows a sample 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 9-1 Power Current Vector



Segmenting the waveform and using base curve technology for each segment provides greater accuracy. In Figure 9-1, the blue dots and blue curve show the waveform using base curve technology. The blue dots represent five segmentation points that divide the waveform into four sections. You can save the segmentation points as characteristic points and model the shape of each segment using a base curve. The blue curve is the third segment that can be represented by a base curve.

The following example describes the format for each current waveform:

t_{start}, I_{start}, $bcid_1$, t_{ip1}, I_{ip1}, $bcid_2$, [t_{ip2}, I_{ip2}, $bcid_3$, ...,] 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 end point.

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 9-15 for the `dynamic_current` syntax and see "Dynamic Power and Ground Current Table Syntax" on page 9-10 for the `pg_current_template` syntax.

The following requirements must be met in the `pg_current_template` group:

- The last `variable_*` value must be `time`. The `time` variable is required.
- The `input_net_transition` and `total_output_net_capacitance` values are available for all `variable_*` attributes except the last `variable_*`. They can be placed in any order except last.

The following conditions describe the `pg_current_template` group:

- There can be zero or one `input_net_transition` variable.
- There can be zero, one, or two `total_output_net_capacitance` variables.
- Each `vector` group in the `pg_current` group describes a current waveform that is compacted into a table in the compact CCS power model.

Similar to the compact CCS timing modeling syntax, compact CCS power modeling syntax uses the `base_curves` group to describe normalized base curves and the `compact_lut_template` group as the compact current waveform template. However, the

`compact_lut_template` group attributes are extended from three dimensions to four dimensions when CCS power models need two `total_output_net_capacitance` attributes.

The compact CCS power modeling syntax is as follows:

```
library (<my_library>) {
  base_curves(<bc_name>) {
    base_curve_type : enum (ccs_half_curve, ccs_timing_half_curve);
    curve_x ("float, …, float");
    curve_y ("integer, float, …, float"); /* base curve #1 */
    curve_y ("integer, float, …, float"); /* base curve #2 */
    …
    curve_y ("integer, float, …, float"); /* base curve #n */
  }
  compact_lut_template (<template_name>) {
    base_curves_group : <bc_name>;
    variable_1 : input_net_transition | total_output_net_capacitance;
    variable_2 : input_net_transition | total_output_net_capacitance;
    variable_3 : input_net_transition | total_output_net_capacitance;
    variable_4 : curve_parameters;
    index_1 ("float, …, float");
    index_2 ("float, …, float");
    index_3 ("float, …, float");
    index_4 ("string, …, string");
  }

  …
  cell(<cell_name>) {
    dynamic_current() {
      switching_group() {
        pg_current(<pg_pin_name>) {
          compact_ccs_power (<template_name>) {
            base_curves_group : <bc_name>;
            index_output : <pin_name>;
            index_1 ("float, …, float");
            index_2 ("float, …, float");
            index_3 ("float, …, float");
            index_4 ("string, …, string");
            values ("float/integer, …, float/integer");
          } /* end of compact_ccs_power */
          …
        } /* end of pg_current */
        …
      } /* end of switching_group */
      …
    } /* end of dynamic_current */
    …
  } /* end of cell */
  …
} /* end of library*/
```

Library-Level Groups and Attributes

This section describes library-level groups and attributes used for compact CCS power modeling.

base_curves Group

The `base_curves` group contains the detailed description of normalized base curves. The `base_curves` group has the following attributes:

base_curve_type Complex Attribute

The `base_curve_type` attribute specifies the type of base curve. The `ccs_half_curve` value allows you to model compact CCS power and compact CCS timing data within the same `base_curves` group. You must specify `ccs_half_curve` before specifying `ccs_timing_half_curve`.

curve_x Complex Attribute

The data array contains the X-axis values of the normalized base curve. Only one `curve_x` 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` attribute. You should define the `curve_x` base curve before `curve_y` for better readability 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 readability. 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>=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 9-20.

The `index_output` attribute is also optional. It is used only on cross type tables. For more information about the `index_output` attribute, see “Dynamic Current Syntax Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

values Attribute

The `values` attribute is required in the `compact_ccs_power` group. The data within the quotation marks (" "), or *line*, represent the current waveform for one index combination. Each value is determined by the corresponding curve parameter. In the following line,

```
"t0, c0, 1, t1, c1, 2, t2, c2, 3, t3, c3, 4, t4, c4"
```

the size is 14 = 8+3*2. Therefore, the curve parameters are as follows:

```
"init_time, init_current, bc_id1, point_time1, point_current1, bc_id2, \
point_time2, point_current2, bc_id3, point_time3, point_current3,
bc_id4,\
end_time, end_current"
```

The elements in the `values` attribute are floating-point numbers for time and current and integers for the base curve ID. The number of current waveform segments can be different for each slew and load combination, which means that each line size can be different. As a result, Liberty syntax supports tables with varying sizes, as shown:

```
compact_ccs_power (<template_name>) {
  …
  index_1("0.1, 0.2"); /* input_net_transition */
  index_2("1.0, 2.0"); /* total_output_net_capacitance */
  index_3 ("init_time, init_current, bc_id1, point_time1, point_current1,
\
bc_id2, [point_time2, point_current2, bc_id3, …], \
end_time, end_current"); /* curve_parameters */
  values
("t0, c0, 1, t1, c1, 2, t2, c2, 3, t3, c3, 4, t4, c4", \ /* segment=4 */
 "t0, c0, 1, t1, c1, 2, t2, c2", \           /* segment=2 */
 "t0, c0, 1, t1, c1, 2, t2, c2, 3, t3, c3", \ /* segment=3 */
 "t0, c0, 1, t1, c1, 2, t2, c2, 3, t3, c3");  /* segment=3 */
}
```

Variation-Aware CCS Power Leakage Current Modeling

Until now, CCS power leakage current modeling syntax required 2*N*+1 libraries, where *N* represents the number of base libraries, in order to describe leakage variations and other characterized quantities over the underlying parameter variations, where 2*N* must characterize two corners at each side of the nominal value of all leakage variation parameters.

Modeling 2*N*+1 libraries results in a large amount of data, particularly for libraries that include timing variation data. The variation-aware CCS power leakage current modeling syntax helps alleviate this library size problem by including all 2*N*+1 libraries in a single library file, and then using the compaction techniques available to CCS libraries to reduce the amount of data needed for leakage variation information.

The following syntax is used for variation-aware CCS power leakage current modeling. It is specified at the cell level.

```
library (<library_name>) {
mode_definition (<mode_name>) {
      mode_value(namestring) {
        when : <boolean expression>;
        sdf_cond : <boolean expression>;
       }
  ...
  va_parameters(<string> , ...);
  cell (<design_name>) {
     leakage_current() { … }
     ...
     cell_based_variation() {
       va_parameters(<string> , ...);
       nominal_va_values(<float>, ...);
       va_leakage_current() {
         va_values(<float> , ...);
         mode (mode_name, mode_value);
         when : <boolean expression>;
         pg_current(<a pg pin name>) {
           value : <float>;
         }
         …
         gate_leakage(<an input pin name>) {
           input_low_value : <float>;
           input_high_value : <float>;
         }
    … }  /* end of va_leakage_current */
  … } /* end of cell_based_variation */
 … } /* end of cell */
… } /* end of library */
```

The following syntax, in simplified format, includes a cell with a single power and ground pin. No pg_current group is required in the va_leakage_current group in this case.

```
cell (<design_name>) {
  leakage_current() { … } /* nominal CCS power leakage current */
  …
  cell_based_variation() {
   va_parameters(<string> , … );
   nominal_va_values(<float>,…);
   va_leakage_current() {
   va_values(<float> , …);
    when : <boolean expression>;
    value : <float>;
    …
    gate_leakage(<an input pin name>) {
    input_low_value : <float>;
    input_high_value : <float>;
    }
...}/* end of va_leakage_current */
… }/* end of cell_based_variation */
…}/* end of cell */
…}/* end of library */
```

cell_based_variation Group

The cell_based_variation group specifies all possible variation-aware models under a cell. The variation leakage current va_leakage_current group is currently the only supported model under the cell_based_variation group.

Note:

The leakage_current group is required if the va_leakage_current group is present within the same cell.

va_parameters Attribute

If the va_parameters attribute is specified at the library level, all variation groups with no parameters specified will default to the va_parameters parameters. There are currently two major variation groups: timing and leakage. If a different set of default parameters is needed for both variation models, the default parameters are only applicable to variation timing. When this happens, you can only specify the default parameters under each cell_based_variation group for variation leakage.

nominal_va_values Attribute

The nominal_va_values attribute characterizes nominal values for all variation parameters. It is required in the cell_based_variation group. The nominal_va_values value is mapped 1-to-1 to the corresponding va_parameters value.The values are applied to nominal CCS power leakage current groups.

va_leakage_current Group

The `va_leakage_current` group specifies characterization corners with variation values in the `cell_based_variation` group. The `va_leakage_current` group can be specified under different `cell_based_variation` groups if they use different `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 set of parameters, all other variations are assumed to be a nominal value. Therefore, a `cell_based_variation` group with *N* variation parameters and exactly 2*N* `va_leakage_current` groups that have same state condition are required.

If only a single power and ground pin exists on a cell, the simplified CCS power representation format can be used. Otherwise, the regular format is required. The only difference between the two formats is the inclusion of the `pg_current` group.

The same library screener checks that apply to nominal CCS power leakage current models also apply to all attributes under the `va_leakage_current` group.

mode Complex Attribute

You define the `mode` attribute within a `va_leakage_current` group. A `mode` attribute pertains to variation-aware leakage current of an individual cell. The variation-aware leakage current of the cell is active when `mode` is instantiated with a name and a value. You can specify multiple instances of the `mode` attribute but only one instance for each cell.

Syntax

```
mode (mode_name, mode_value);
```

Library Compiler issues an error message if the *mode_name* and *mode_value* strings are not already defined by a `mode_definition` group in the cell.

va_values Attribute

The `va_values` attribute defines the values of each variation parameter for all corners characterized in `va_leakage_current` groups. The `va_values` attribute is required for each `va_leakage_current` group. The value of this attribute is mapped 1-to-1 to the corresponding `va_parameters` value.

CCS Power Variation Leakage Example

The following example defines CCS Power variation leakage information.

```
library(my) {
     delay_model : table_lookup;
     voltage_map(VDD1, 2.0);
     voltage_map(VDD2, 2.5);
     voltage_map(VSS1, 0.0);
```

```
  voltage_map(VSS2, 0.2);
...
  cell (va_leakage_cell) {
    pin(A1) {
     direction : input;
     ...
    }
    pin(A2) {
     direction : input;
     ...
    }
    pin(A3) {
     direction : input;
     ...
    }
    pin(ZN1) {
     direction : output;
     ...
    }

    pin(ZN2) {
     direction : output;
     ...
    }
    pg_pin(VDD1) {
      voltage_name : VDD1;
      pg_type : primary_power;
    }
    pg_pin(VDD2) {
      voltage_name : VDD2;
      pg_type : backup_power;
    }
    pg_pin(VSS1) {
      voltage_name : VSS1;
      pg_type : primary_ground;
    }
    pg_pin(VSS2) {
   voltage_name : VSS2;
   pg_type : backup_ground;
   }

   leakage_current() {  /* nominal leakage power */
     when : "A1 & !A2 | ZN1";
     pg_current(VDD1) {
      value : 4.5;
     }
    pg_current(VDD2) {
      value : 4.5;
    }
    pg_current(VSS2) {
      value : -3.5;
    }
   gate_leakage(A2) {  /* must be either input or inout pin */
```

```
          input_high_value : 2.1;  /* must be positive or zero */
          input_low_value : -1.7;  /* must be negative or zero */
        }
      }

        leakage_current() {  /* nominal leakage power */
          when : "!A1 & !ZN1";
          pg_current(VDD1) {
           value : 4.5;
          }
          pg_current(VDD2) {
           value : 4.5;
          }
          pg_current(VSS2) {
           value : -2.5;
          }
          gate_leakage(A2) {  /* must be either input or inout pin */
           input_high_value : 2.1;  /* must be positive or zero */
           input_low_value : -1.7;  /* must be negative or zero */
        }
      }
      leakage_current() { /* nominal leakage power : default state with no
 state
  condition*/
        pg_current(VDD1) {
         value : 4.5;
        }

        pg_current(VSS2) {
         value : -14.5;
        }
        pg_current(VDD2) {
         value : 14.5;
        }
        gate_leakage(A3) {
          input_high_value : 6.3;
          input_low_value : -2.9;
        }
       }
      cell_based_variation() {
        va_parameters(var1, var2);
        nominal_va_values(10.0, 10.0);
        va_leakage_current() {  /* variation leakage power */
        when : "A1 & !A2 | ZN1";
        va_values(11.0, 10.0);
        pg_current(VDD1) {
         value : 4.5;
        }
        pg_current(VDD2) {
         value : 4.5;
        }
        pg_current(VSS2) {
         value : -3.5;
```

```
 }
 gate_leakage(A2) {  /*must be either input or inout pin */
  input_high_value : 2.1;  /*must be positive or zero */
  input_low_value : -1.7;  /*must be negative or zero */
 }
}
va_leakage_current() {  /* variation leakage power */
  when : "A1 & !A2 | ZN1";
  va_values(9.0, 10.0);
  pg_current(VDD1) {
    value : 4.5;
  }
  pg_current(VDD2) {
    value : 4.5;
  }
  pg_current(VSS2) {
    value : -3.5;
  }
 gate_leakage(A2) {  /* must be either input or inout pin */
  input_high_value : 2.1;  /*must be positive or zero */
  input_low_value : -1.7;  /*must be negative or zero */
   }
 }
va_leakage_current() {  /* variation leakage power */
  when : "A1 & !A2 | ZN1";
  va_values(10.0, 11.0);
  pg_current(VDD1) {
   value : 4.5;
  }
  pg_current(VDD2) {
   value : 4.5;
  }
  pg_current(VSS2) {
   value : -3.5;
  }
 gate_leakage(A2) {  /* must be either input or inout pin */
  input_high_value : 2.1;  /*must be positive or zero */
  input_low_value : -1.7;  /*must be negative or zero */
 }
}
va_leakage_current() {  /* variation leakage power */
  when : "A1 & !A2 | ZN1";
  va_values(10.0, 9.0);
  pg_current(VDD1) {
   value : 4.5;
  }
  pg_current(VDD2) {
   value : 4.5;
  }
  pg_current(VSS2) {
   value : -3.5;
  }
  gate_leakage(A2) {  /* must be either input or inout pin */
```

```
      input_high_value : 2.1;  /*must be positive or zero */
      input_low_value : -1.7;  /*must be negative or zero */
    }
  }
  va_leakage_current() {  /* variation leakage power */
    when : "!A1 & !ZN1";
    va_values(11.0, 10.0);
    pg_current(VDD1) {
     value : 4.5;
    }
    pg_current(VDD2) {
     value : 4.5;
    }
    pg_current(VSS2) {
     value : -2.5;
    }
    gate_leakage(A2) {  /* must be either input or inout pin */
    input_high_value : 2.1;  /* must be positive or zero */
    input_low_value : -1.7;  /* must be negative or zero */
    }
  }
 va_leakage_current() {  /* variation leakage power */
   when : "!A1 & !ZN1";
   va_values(9.0, 10.0);
   pg_current(VDD1) {
    value : 4.5;
   }
   pg_current(VDD2) {
    value : 4.5;
   }
   pg_current(VSS2) {
    value : -2.5;
   }
   gate_leakage(A2) {  /* must be either input or inout pin */
     input_high_value : 2.1;  /* must be positive or zero */
     input_low_value : -1.7;  /* must be negative or zero */
     }
  }
  va_leakage_current() {  /* variation leakage power */
   when : "!A1 & !ZN1";
   va_values(10.0, 11.0);
   pg_current(VDD1) {
     value : 4.5;
   }
   pg_current(VDD2) {
    value : 4.5;
   }
   pg_current(VSS2) {
    value : -2.5;
   }
   gate_leakage(A2) {  /* must be either input or inout pin */
    input_high_value : 2.1;  /* must be positive or zero */
    input_low_value : -1.7;  /* must be negative or zero */
```

```
        }
       }
      va_leakage_current() {  /* variation leakage power */
       when : "!A1 & !ZN1";
       va_values(10.0, 9.0);
       pg_current(VDD1) {
        value : 4.5;
       }
       pg_current(VDD2) {
        value : 4.5;
       }
       pg_current(VSS2) {
        value : -2.5;
       }
      gate_leakage(A2) {  /* must be either input or inout pin */
        input_high_value : 2.1;  /* must be positive or zero */
        input_low_value : -1.7;  /* must be negative or zero */
      }
    }
     va_leakage_current() {  /* variation leakage power  : default state
*/
      va_values(11.0, 10.0);
      pg_current(VDD1) {
       value : 4.5;
      }
      pg_current(VSS1) {
       value : -4.5;
      }
     pg_current(VSS2) {
       value : -14.5;
     }
     pg_current(VDD2) {
       value : 14.5;
     }
     gate_leakage(A3) {
       input_high_value : 6.3;
       input_low_value : -2.9;
     }
    }
    va_leakage_current() {  /* variation leakage power : default state
*/
     va_values(9.0, 10.0);
     pg_current(VDD1) {
       value : 4.5;
     }
     pg_current(VSS2) {
       value : -14.5;
     }
     pg_current(VDD2) {
       value : 14.5;
     }
    gate_leakage(A3) {
       input_high_value : 6.3;
```

```
        input_low_value : -2.9;
       }
      }
      va_leakage_current() {  /* variation leakage power : default state
*/
       va_values(10.0, 11.0);
       pg_current(VDD1) {
        value : 4.5;
       }
       pg_current(VSS2) {
        value : -14.5;
       }
       pg_current(VDD2) {
        value : 14.5;
       }
       gate_leakage(A3) {
         input_high_value : 6.3;
         input_low_value : -2.9;
       }
      }
 …
}  /* end of cell_based_variation */
  …
 } /* end of cell  */
 …
}  /* end of library */
```

Conditional Data Modeling for Variation-Aware Leakage Current Model by Mode Example

```
library(my) {
  delay_model : table_lookup;
  /* unit attributes */
  time_unit : "1ns";
  ...
cell (AND3) {
  pg_pin(V1) {
     voltage_name : VDD1;
     pg_type : primary_power;
  }
  pg_pin(V2) {
     voltage_name : VDD2;
     pg_type : backup_power;
  }
  pg_pin(G1) {
     voltage_name : GND1;
     pg_type : primary_ground;
  }
  pg_pin(G2) {
     voltage_name : GND2;
     pg_type : backup_ground;
  }
```

```
  power_cell_type : stdcell;
mode_definition(rw) {
mode_value(read) {
when : "A1";
sdf_cond : "A1 == 1";
}
mode_value(write) {
when : "!A1";
sdf_cond : "A1 == 0";
}
}
  cell_based_variation(){
    va_parameters(var1, var2);
    nominal_va_values(10.0, 10.0);
    va_leakage_current() {
       mode(rw, read);
       va_values(11.0, 10.0);
       pg_current(V1) {
           value : 4.5;
       }
       pg_current(V2) {
           value : 4.5;
       }
       pg_current(G2) {
           value : -3.5;
       }
       gate_leakage(A1) {
           input_high_value : 7.1;
           input_low_value : -8.7;
       }
}
…
       }
 area : 1.0 ;
 pin(A1) {
  direction : input;
  capacitance : 0.1 ;
  related_power_pin : V1;
  related_ground_pin : G1;
 }
 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”;
   …
  }//end of timing
} //end of pin ZN1
  } //end of cell
```

```
} // end of library
```

Checking Variation-Aware CCS Power Leakage Current Syntax

Library Compiler automatically checks variation-aware CCS power leakage current syntax and reports any problems it encounters. For information about the types of checks Library Compiler performs for variation-aware CCS power leakage current syntax, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Composite Current Source Dynamic Power Examples

This section provides the following CCS dynamic power examples:

- Design Cell With a Single Output Example
- Dense Table With Two Output Pins Example
- Cross Type With More Than One Output Pin Example
- Diagonal Type With More Than One Output Pin Example

For more information about the syntax in the following examples, see the Library Compiler Reference Manual: Technology and Symbol Libraries.

Design Cell With a Single Output Example

```
pg_current_template ( CCS_power_1 ) {
  variable_1 : input_net_transition ;
  variable_2 : total_output_net_capacitance ;
  variable_3 : time ;
}

cell (example) {

          dynamic_current() {
                 when: D;
                 related_inputs : CP;
                 related_outputs : Q;
                 switching_group ( ) {
                              input_switching_condition(rise);
                              output_switching_condition(rise);
            pg_current (VDD ) {
                 vector ( CCS_power_1 ) {
                                        reference_time : 0.01;
                     index_1 ( 0.01 )
                     index_2 ( 1.0 )
                     index_3 ( 0.000, 0.0873, 0.135, 0.764)
                     values ( 0.002, 0.009, 0.134, 0.546)
                   }
```

Dense Table With Two Output Pins Example

```
pg_current_template ( CCS_power_1 ) {
  variable_1 : input_net_transition ;
  variable_2 : total_output_net_capacitance ;
  variable_3 : total_output_net_capacitance ;
  variable_4 : time ;
             }

  cell ( example ) {

dynamic_current() {

      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_ntin_oload_time) {
            reference_time : 0.0015 ;
            index_1("0.0019");
            index_2("0.001");
            index_3("0, 0.006, 0.03, 0.07, 0.09, 0.1, 0.2, 0.3, 0.4,
0.5");
            values("5e-06, 0.001, 0.02, 0.03, 0.05, 0.08, 0.09, 0.04,
0.009,
                 5.0e-06");
          }
        }
      }
    }
}
```

Cross Type With More Than One Output Pin Example

```
pg_current_template ( CCS_power_1 ) {
  variable_1 : input_net_transition ;
  variable_2 : total_output_net_capacitance ;
  variable_3 : time ;
  }

cell ( example )

dynamic_current() {
      when: D;
        related_inputs : CP;
        related_outputs : Q QN QN1 QN2;
        typical_capacitances(10.0 10.0 10.0 10.0);
        switching_group ( ) {
          input_switching_condition(rise);
          output_switching_condition(rise, fall, fall,
                      fall);
          pg_current (VSS) {
            vector ( CCS_power_1 ) {
              index_output : Q;
              reference_time : 0.01;
              index_1 ( 0.01 )
              index_2 ( 5.0)
              index_3 ( 0.000, 0.0873, 0.135, 0.764)
              values ( 0.002, 0.009, 0.134, 0.546)
            }

            vector ( CCS_power_1 ) {
               index_output : QN;
               reference_time : 0.01;
               index_1 ( 0.01 )
               index_2 ( 1.0 )
               index_3 ( 0.000, 0.0873, 0.135, 0.764)
               values ( 0.002, 0.009, 0.134, 0.546)
            }
             vector ( CCS_power_1 ) {
               index_output : QN;
               reference_time : 0.01;
               index_1 ( 0.01 )
               index_2 ( 5.0 )
               index_3 ( 0.000, 0.0873, 0.135, 0.764)
               values ( 0.002, 0.009, 0.134, 0.546)
            }
```

Diagonal Type With More Than One Output Pin Example

```
pg_current_template ( CCS_power_1 ) {
  variable_1 : input_net_transition ;
  variable_2 : total_output_net_capacitance ;
  variable_3 : time ;
}

         cell ( example )
               dynamic_current() {
                 when: D ;
                 related_inputs : CP;
                 related_outputs : Q QN QN1 QN2;
                     switching_group ( ) {
                           input_switching_condition(rise);
                           output_switching_condition(rise,
fall, fall, fall);
                        }
                  }
               pg_current (VSS) {
                 vector ( CCS_power_1 ) {
                      reference_time : 0.01;
                   index_1 ( 0.01 )
                   index_2 ( 1.0 )
                   index_3 ( 0.000, 0.0873, 0.135, 0.764)
                   values ( 0.002, 0.009, 0.134, 0.546)
                   }
                }
```

10

Interface Timing of Complex Sequential Blocks

This chapter describes the procedure and methodology for specifying the static timing of complex sequential blocks used as black boxes.

The interface timing specification applies to ASIC libraries containing sequential blocks modeled as black boxes whose functionality is either too complex to model or not required.

These are the interface timing concepts described in this chapter:

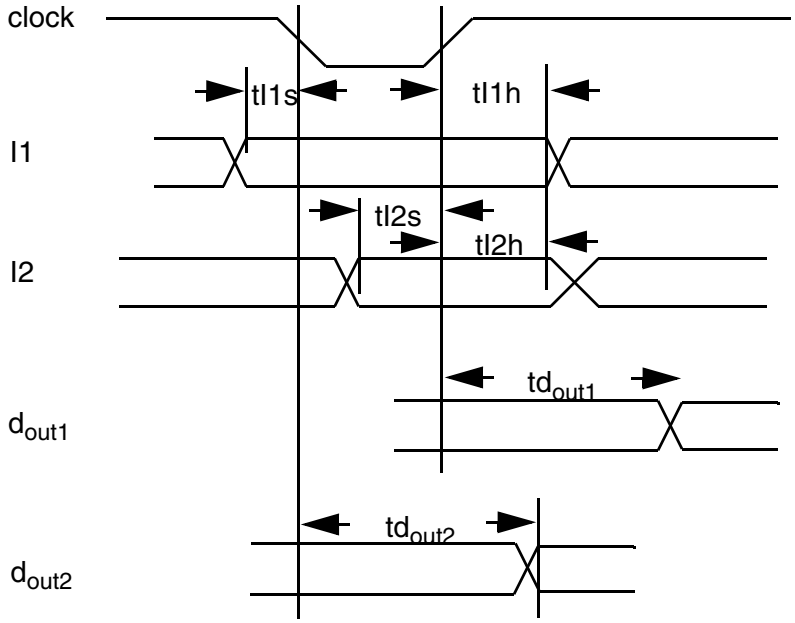
Interface Timing Specification

The interface timing specification of a sequential block describes the static timing of a block as seen from the outside. This specification does not include the timing of the internal paths of the block.

A complex sequential block contains more than one sequential device. It can have more than one driving clock to which you specify input constraints and output delays. Such blocks include RAMs and hard macros, such as the test access port (TAP) controller circuit in the Joint Test Action Group (JTAG).

Figure 10-1 is an example of the interface timing of a complex block.

Figure 10-1 Interface Timing of a Complex Block



In Figure 10-1,

- Input I1 has a setup constraint with respect to the falling edge of the clock and a hold constraint with respect to the rising edge of the clock.
- The setup constraints for inputs I1 and I2 are specified with respect to different edges of the clock.
- Output data d_{out1} is propagated on the rising edge of the clock. Output data d_{out2} is propagated on the falling edge of the clock.

Sequential Cell Timing Without Interface Timing Specification

In static timing, a sequential cell is considered to contain one or more edge-triggered flip-flops (rising or falling edge) or level-sensitive latches (negative or positive level). When functional information is not available during synthesis, the interpretation of the cell is based on the various timing relationships specified in the cell description. Cells are interpreted as follows:

- As a simple edge-triggered device if the setup arc and the output delay arc are specified for the same clock edge
- As a master-slave device if the setup arc and the output delay arc are specified for different edges of the clock
- As a level-sensitive device if it has at least one output that combines a combinational arc from a data input and a sequential arc from a clock. However, if cells do not have a sequential arc from the clock to the output port with a combinational arc from the D pin to the output port, then an edge-triggered register is inferred.

When a cell is interpreted as above, some of the timing information specified in the library is ignored. For example, if a cell is interpreted as a rising-edge-triggered device, Library Compiler checks the hold constraint on the rising edge even if this constraint is specified for the falling edge of the clock.

Supported Features of Interface Timing

Although the rules described in the previous section work well for simple sequential cells, they do not apply to complex sequential blocks. The interface timing capability supports the timing of complex sequential blocks without any assumptions about their underlying structure.

During synthesis, timing relationships are interpreted according to the following well-defined set of rules, in which each relationship is interpreted independently of other relationships:

- Interface timing supports multiple clock signals. A block can have one or more clock inputs with different waveforms for each clock.
- Setup and hold constraints of data inputs can be specified with respect to the rising or falling edges of one or more clocks.

  A control input can have recovery or removal arcs specified with respect to one or more clocks. Note that recovery and removal constraints are not checked by synthesis.
- Propagation delays can be specified with respect to the rising or falling edges of one or more clocks.

An output pin can have a propagation delay specified with respect to the falling or rising edge of any clock. One output can have several propagation delays specified with respect to several clock inputs.

- Combinational delay can be specified with respect to one or more data inputs.

  An output pin can also have a delay arc from one or more inputs. This arc can coexist with other arcs from clock inputs and can be a regular delay arc, an asynchronous reset, or a clear arc. Regular delay arcs include combinational arcs.

- Timing properties of clocks, such as minimum period and minimum pulse width, can be specified in the same way that they are specified for regular library cells. These constraints are not checked by synthesis.

The benefits of using the interface timing capability to describe the static timing of complex sequential blocks are:

- Higher-quality results from synthesis

  With accurate timing models for complex blocks, optimization of the logic around these blocks is better, because the synthesis process does not miss real violations or detect false violations. When the synthesis process detects false violations, it optimizes the wrong path, making the circuit larger and faster than it needs to be. Real violations missed in synthesis, which are detected in simulation after synthesis, require more synthesize-simulate iterations and, consequently, a longer development cycle.

- Support for complex sequential blocks

  The timing of complex sequential blocks can be specified for the synthesis tools to use when making optimization decisions.

- Abstraction and information hiding

  Because interface timing describes the static timing of a block as seen from the outside, it is not necessary to specify the internal structure of the block or the timing paths internal to the block, which are assumed to have been verified when the block was built.

  For example, if the block has a corresponding gate-level netlist, it is not necessary to export the netlist to be able to time through it. You can use interface timing specification to specify the static timing of the block while keeping it as a black box in terms of internal structure and functionality. You instantiate the block in the design, compile the design, and export it to downstream tools.

  During all the phases of the synthesis process, the block is treated as a leaf cell in the design. Functional information is needed only when the design is being verified through simulation when a functional simulation model of the block can be used.

Describing Blocks With Interface Timing

This section describes how to use Library Compiler syntax to specify the interface timing of complex blocks. The following sections describe the design flow of these blocks through synthesis.

Using Library Compiler Syntax

To specify the interface timing of a macro block, describe the block as a cell or model in the library and use `timing` group syntax to describe the timing arcs. Library Compiler treats this `cell` or `model` group like a regular `cell` or `model` group in an ASIC library.

Differences From Regular Cell Specification

These are the differences from regular cell description for specifying the interface timing of complex sequential blocks in Library Compiler syntax:

1. Add this attribute and value to the `cell` group:

   ```
   interface_timing : true;
   ```

   This indicates that the timing arcs must be interpreted according to interface timing specification semantics. If this attribute is missing or its value is false, the timing relationships are interpreted as those of a regular cell rather than according to interface timing specification semantics.

2. Cells with interface timing are strictly black boxes, in terms of functionality. In other words, the `function` attribute must not be specified for any output pin of the cell. In addition, the `cell` group cannot contain an `ff` group. As a consequence, cells with interface timing cannot be inferred but must be instantiated.

Advanced Modeling Technology

To work more effectively with PrimeTime to handle chip-level designs, Synopsys expanded the Library Compiler interface timing specification modeling capability to support the PrimeTime modeling requirement.

For more information about PrimeTime, see the PrimeTime suite of documents: *PrimeTime Modeling User Guide*, *PrimeTime Tutorial*, and *PrimeTime User Guide*.

The next two sections describe new attributes related to PrimeTime support.

model Group

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

Library Compiler generates hierarchical .db designs with the `model` group, which can accept all the groups and attributes that a `cell` group accepts, plus these additional two new attributes:

- `cell_name` simple attribute
- `short` complex attribute

Syntax

```
model(model_name) {
   area : float;
   ...
   cell_name : cell_string; /*optional*/
   ...
   pin(A, Y) { ...}
   short (A, Y);
}
```

cell_name Simple Attribute

The `cell_name` attribute specifies the name of the cell within a `model` group.

Syntax

```
cell_name : name_id ;
```

Example

```
model(modelA) {
 cell_name : ”cellA”;
 ...

}
```

short Complex Attribute

The `short` complex attribute lists the shorted ports that are connected by metal or polytrace. These ports are modeled within a `model` group.

The most common example of a shorted port is a feedthrough, where an input port is directly connected to an output port. Another example is two output ports that fan out from the same gate.

Syntax

```
short("name_list_id") ;
```

Example

```
short (b, y);
```

model and short Examples

Example 10-1 shows `model` and `short` descriptions.

Example 10-1 model Group With short Attributes

```
library(TEST) {

  model(modelA) {
    area : 0.4;
    short(b, y);
    short(c, y);
  cell_name : "cellA";
  pin(y) {
    direction : output;
    timing() {
      related_pin : a;
    }
  }
  pin(a) {
    direction : input;
    capacitance : 0.1;
  }
  pin(b) {
    direction : input;
    capacitance : 0.1;
  }
  pin(c) {
    direction : input;
    capacitance : 0.1;
    clock : true;
  }
 }
}
```

generated_clock Group

A `generated_clock` group is defined within a `cell` group or a `model` group to describe a new clock that is generated from a master clock by

- clock frequency division
- clock frequency multiplication
- edge derivation

Syntax

```
cell (name_id) {
 ...generated_clock description...
}
```

Simple Attributes

```
clock_pin
divided_by
duty_cycle
invert
master_pin
multiplied_by
```

Complex Attributes

```
edges
shifts
```

clock_pin Simple Attribute

Required in the `generated_clock` group. The `clock_pin` attribute identifies a pin connected to an input clock signal.

Syntax

```
clock_pin : name ;
```

Example

```
clock_pin : clk1;
```

divided_by Simple Attribute

The `divided_by` attribute specifies the frequency division factor, which must be a power of 2.

Syntax

```
divided_by : integer;
```

Example

```
generated_clock(genclk1) {
   clock_pin : clk1;
   master_pin : clk;
   divided_by : 2;
}
```

This code fragment shows a clock pin (clk1) generated by division of the original clock pin (clk) frequency by 2.

duty_cycle Simple Attribute

The `duty_cycle` attribute specifies the duty cycle (expressed as a floating-point number) for frequency multiplied clocks. Use this option for high pulse width.

Syntax

```
duty_cycle : float;
```

Example

```
generated_clock(genclk2) {
   clock_pin : clk1;
   master_pin : clk;
   multiplied_by : 2;
   duty_cycle : 50.0;
}
```

This code fragment shows a clock pin (clk1) generated by multiplication of the original clock pin (clk) frequency by 2 with a duty cycle of 50.

invert Simple Attribute

The `invert` attribute inverts the waveform generated by multiplication or division. Set this attribute to true to invert the waveform. Set it to false (default) if you do not want to invert the waveform.

Syntax

```
invert : true | false ;
```

Example

```
generated_clock(genclk1) {
   clock_pin : clk1;
   master_pin : clk;
   divided_by : 2;
   invert : true;
}
```

This code fragment shows a clock pin (clk1) generated by division of the original clock pin (clk) frequency by 2 and then inversion.

master_pin Simple Attribute

Required in the `generated_clock` group. The `master_pin` attribute identifies the master (original) clock pin.

Syntax

```
master_pin : name ;
```

Example

```
master_pin : clk;
```

multiplied_by Simple Attribute

The `multiplied_by` attribute specifies the frequency multiplication factor, which must be a power of 2.

Syntax

```
multiplied_by : integer;
```

Example

```
generated_clock(genclk2) {
   clock_pin : clk1;
   multiplied_by : 2;
   master_pin : clk;
}
```

This code fragment shows a clock pin (clk1) generated by multiplication of the original clock pin (clk) frequency by 2.

Syntax

```
edges (edge1,edge2,edge3);
```

Example

```
edges (1, 3, 5);
```

edges Complex Attribute

The `edges` attribute specifies a list of edges (in terms of the edge numbers) from the master clock that form the edges of the generated clock. You must specify three edges. Use this option when simple division or multiplication is insufficient to describe the generated clock waveform.

shifts Complex Attribute

The `shifts` attribute specifies the shifts (in time units) to be added to the edges specified in the edge list to generate the clock. The number of shifts must equal the number of edges (that is, three). This shift modifies the ideal clock edges (it is not considered as clock latency).

Syntax

```
shifts (shift1,shift2,shift3);
```

Example

```
shifts (5.0, -5.0, 0.0);
```

Generated Clock Example

Example 10-2 shows a generated clock description.

Example 10-2 Generated Clock Description

```
cell(acell) {
   ...
   generated_clock(genclk1) {
      clock_pin : clk1;
      master_pin : clk;
      divided_by : 2;
      invert : true;
   }
   generated_clock(genclk2) {
      clock_pin : clk1;
      master_pin : clk;
      multiplied_by : 2;
      duty_cycle : 50.0;
   }
   generated_clock(genclk3) {
      clock_pin : clk1;
      master_pin : clk;
      edges(1, 3, 5);
      shifts(5.0, -5.0, 0.0);
   }
   ....
   pin(clk) {
      direction : input;
      clock : true;
      capacitance : 0.1;
   }
   pin(clk1) {
      direction : input;
      clock : true;
      capacitance : 0.1;
   }
}
```

Similarities to Regular Cell Specification

These are the ways interface timing specification is similar to regular cell description:

1. Associate the block with a library. Use one of these two methods:

   - Specify the `cell` group or the `model` group within a `library` group and use the `read_lib` command to import the library into .db format.
   - Specify the `cell` group or the `model` group separately and use the `update_lib` command to associate it with a library that already exists in .db format.

2. Use the same delay model for interface timing of a block as the one defined in its associated library. For example, associate the block using the linear delay model to specify timing arcs with a library using the linear model. Similarly, associate the block using the nonlinear delay model with a library using the nonlinear model.
3. Use the same timing arc syntax for describing regular cells to specify the timing arcs that describe the interface timing relationships.

According to the interface timing specification policy, blocks with interface timing are always considered black boxes in terms of functionality. To simulate designs containing blocks with interface timing, you must provide a functional simulation model of such blocks.

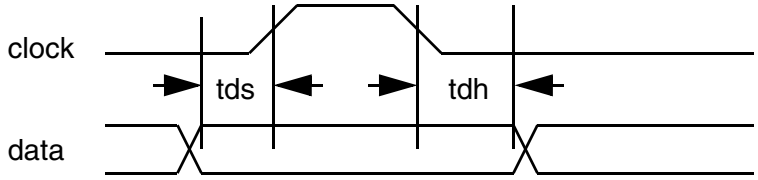
Examples of Interface Timing Relationships

This section describes the various timing relationships supported by the interface timing specification and provides an example of a fairly complex sequential block with two clocks, four data inputs, and two data outputs.

Input Setup and Hold

A data input can have a setup or hold constraint specified with respect to the rising or falling edge of any clock input. Figure 10-2 shows a data input with the setup constraint specified with respect to the rising edge of the clock and the hold constraint specified with respect to the falling edge of the clock.

Figure 10-2 Data Input With Specified Setup and Hold Constraints



Example 10-3 shows the Library Compiler description of these relationships.

Example 10-3 Description of Timing Relationships in Figure 10-2

```
pin (I2) {
   timing () {
      timing_type : setup_rising;
      intrinsic_rise : 2.20; /* tds */
      intrinsic_fall : 2.20;
      related_pin : "clock";
   }
   timing () {
      timing_type : hold_falling;
      intrinsic_rise : 0.95;   /* tdh */
      intrinsic_fall : 0.95;
```

```
      related_pin : "clock";
   }
}
```

Output Propagation

The output delay can be specified with respect to the rising or falling edge of the clock. Figure 10-3 shows the output section of a block. Output O1 is propagated on the rising edge of the clock, and output O2 is propagated on the falling edge.

Figure 10-3 Output Section of a Macro Block

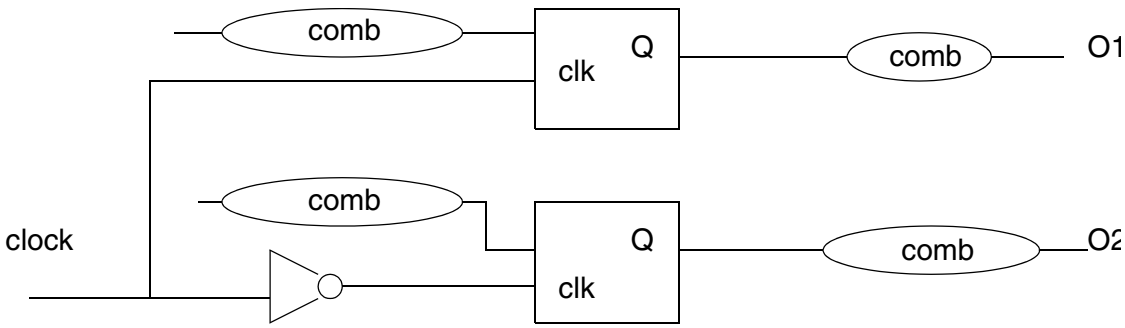
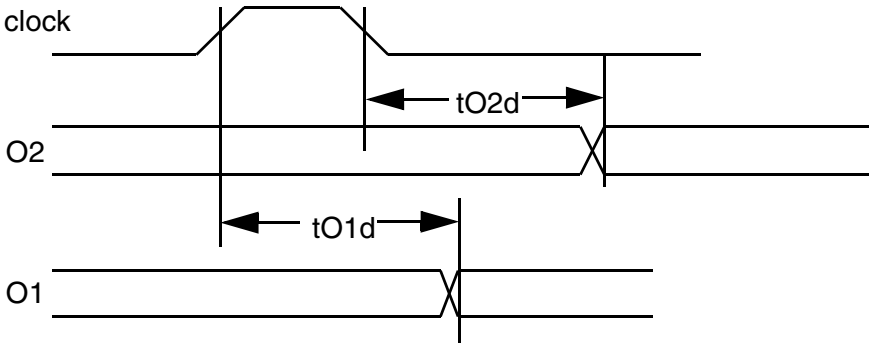


Figure 10-4 shows the interface timing relationships for the two outputs.

Figure 10-4 Output Timing Relationships for the Block in Figure 10-3



Example 10-4 shows the Library Compiler description of the relationships in Figure 10-4.

Example 10-4 Description of Timing Relationships in Figure 10-4

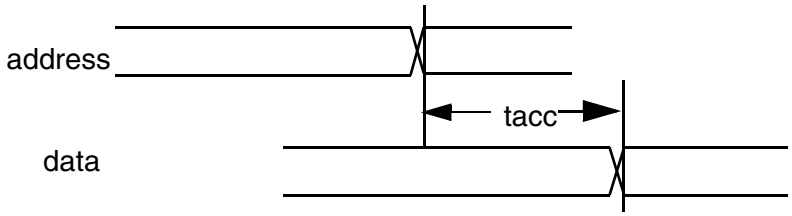
```
pin(O1){
   direction : output;
   timing () {
   timing_type : rising_edge; /*sequential delay arc */
      intrinsic_rise : 2.25; /* tO1d */
```

```
      rise_resistance : 0.020;
      intrinsic_fall : 2.57;
      fall_resistance : 0.017;
      related_pin : "clock";
   }
}

pin(O2){
   direction : output;
   timing () {
      timing_type : falling_edge; /*sequential delay arc */
      intrinsic_rise : 3.15; /* tO2d */
      rise_resistance : 0.020;
      intrinsic_fall : 3.00;
      fall_resistance : 0.017;
      related_pin : "clock";
   }
}
```

Through Combinational Paths

An output of a sequential block can have a propagation delay specified with respect to a nonclock input. This relationship indicates the existence of a direct combinational path between the output and the input. The output will always change (after some delay) when the input changes. Figure 10-5 shows such a case.

Figure 10-5 Timing Relationships for Through Combinational Paths



Example 10-5 shows the Library Compiler description of the relationships in Figure 10-5.

Example 10-5 Description of Timing Relationships in Figure 10-5

```
bus(address){
   direction : input;
   bus_type  : "bus10"
}

bus(data){
   direction : output;
   bus_type   : "bus8"
   timing () {
      timing_sense : non_unate;
      intrinsic_rise : 8.15; /* tacc */
      rise_resistance : 0.020;
```

```
        intrinsic_fall : 7.00;
        fall_resistance : 0.017;
        related_pin : "address";
    }
}
```

Interface Timing of a Complex Sequential Block

Figure 10-6 shows the internal structure of a complex block containing sequential and combinational logic. Note that the input constraints for the sequential elements in the circuit are specified with respect to both edges of the clock.

Figure 10-6 Complex Sequential Block Supported by the Interface Timing Specification

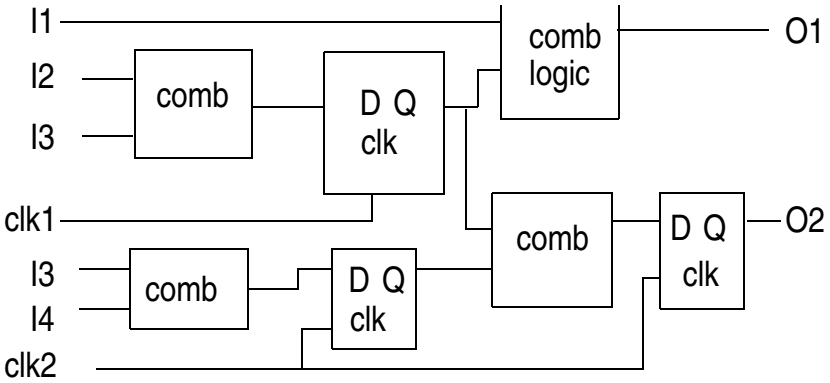
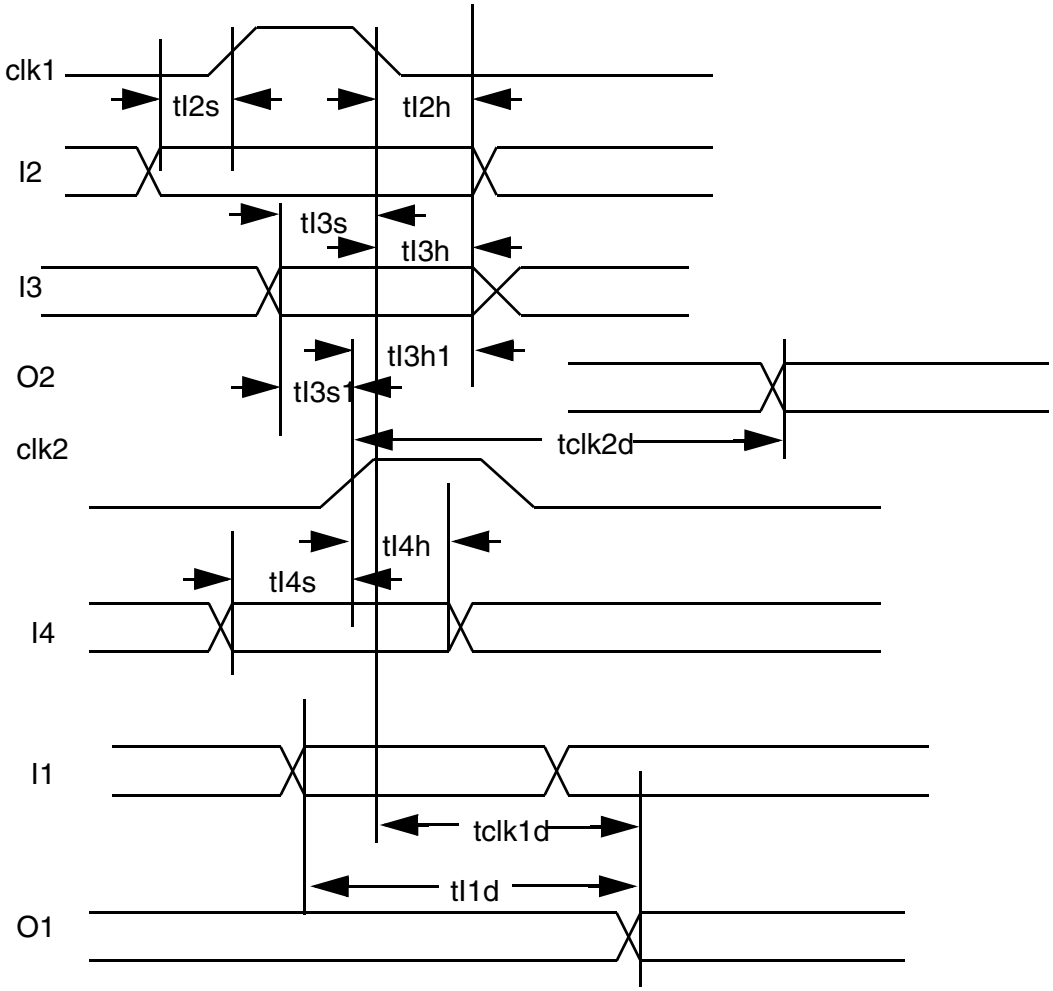


Figure 10-7 is the timing diagram illustrating the various interface relationships of this block.

- The block has two clock pins, clk1 and clk2.
- Input I2 has a setup constraint with respect to the rising edge of clk1 and a hold constraint with respect to the falling edge of the same clock.
- Input I3 has setup and hold constraints specified with respect to both clocks (rising for clk2 and falling for clk1).
- Output O1 is propagated on the falling edge of clk1, and output O2 is propagated on the rising edge of clk2.
- Output O1 combines a combinational path from I1 and a sequential path from clk1.

Figure 10-7 Timing Relationships for the Circuit in Figure 10-6



Example 10-6 shows the library description of the interface timing specification of the block shown in Figure 10-6. The CMOS delay model is used for this cell.

Example 10-6 Description of the Timing Relationships in Figure 10-6

```
cell (XR_2120) {
   area : 100.000000;
   pin (I1) {
      direction : input;
      capacitance : 1.46;
      fanout_load : 1.46;
   }
   pin (I2) {
      direction : input;
      capacitance : 1.46;
      fanout_load : 1.46;
      timing () {
         timing_type : setup_rising;
```

```
      intrinsic_rise : 2.20; /* tI2s */
      intrinsic_fall : 2.20;
      related_pin : "clk1";
   }
   timing () {
      timing_type : hold_falling;
      intrinsic_rise : 0.95;   /* tI2h */
      intrinsic_fall : 0.95;
      related_pin : "clk1";
   }
}
pin (I3) {
   direction : input;
   capacitance : 1.46;
   fanout_load : 1.46;
   timing () {
      timing_type : setup_falling;
      intrinsic_rise : 1.20; /* tI3s */
      intrinsic_fall : 1.20;
      related_pin : "clk1";
   }
   timing () {
      timing_type : hold_falling;
      intrinsic_rise : 1.95;   /* tI3h */
      intrinsic_fall : 1.95;
      related_pin : "clk1";
   }
   timing () {
      timing_type : setup_rising;
      intrinsic_rise : 0.70; /* tI3s1 */
      intrinsic_fall : 0.70;
      related_pin : "clk2";
   }
   timing () {
      timing_type : hold_rising;
      intrinsic_rise : 2.95;   /* tI3h1 */
      intrinsic_fall : 2.95;
      related_pin : "clk2";
   }

}
pin (I4) {
   direction : input;
   capacitance : 1.46;
   fanout_load : 1.46;
   timing () {
      timing_type : setup_rising;
      intrinsic_rise : 1.70; /* tI2s */
      intrinsic_fall : 1.70;
      related_pin : "clk2";
   }
   timing () {
      timing_type : hold_rising;
```

```
            intrinsic_rise : 0.80;   /* tI2h */
            intrinsic_fall : 0.80;
            related_pin : "clk2";
         }
      }
      pin (clk1) {
         direction : input;
         capacitance : 1.13;
         fanout_load : 1.13;
         clock : true;
         /* clock pins must be labeled as such */
      }
      pin (clk2) {
         direction : input;
         capacitance : 1.13;
         fanout_load : 1.13;
         clock : true;
         /* clock pins must be labeled as such */
      }
      pin (O1){
         direction : output;
         timing () {
            timing_sense : non_unate; /* combinational delay */
            intrinsic_rise : 3.25; /* tI1d */
            rise_resistance : 0.020;
            intrinsic_fall : 3.50;
            fall_resistance : 0.017;
            related_pin : "I1";
         }
         timing () {
            timing_type : falling_edge;
            /* sequential delay arc */
            intrinsic_rise : 1.25; /* tclk1d */
            rise_resistance : 0.020;
            intrinsic_fall : 1.50;
            fall_resistance : 0.017;
            related_pin : "clk1";
         }
      }
      pin (O2){
         direction : output;
         timing () {
            timing_type : rising_edge;
            /*sequential delay arc */
            intrinsic_rise : 2.25; /* tclk2d */
            rise_resistance : 0.020;
            intrinsic_fall : 2.57;
            fall_resistance : 0.017;
            related_pin : "clk2";
         }
      }
   } /* end cell */
```

Using Blocks With Interface Timing in Synthesis

To use a cell with interface timing, follow the same procedure used to instantiate technology-specific library cells in a design. Refer to the Design Compiler reference manuals for the details of this procedure. This is a brief review of those steps:

1. Instantiate the block in the design in the same way technology-specific cells are instantiated. Note that cells with interface timing cannot be inferred, because they are black boxes in terms of functionality.
2. For the design to link successfully, add the library containing the block to the `link_library dc_shell` variable.
3. Compile the design.
4. Use the `report_timing` and `report_constraint dc_shell` commands to obtain timing and constraint reports.

You can use multiple libraries in the `link_library dc_shell` variable. Different libraries can use different delay models to describe the timing of their member cells. For example, one library might describe the interface timing of complex blocks that use the nonlinear delay model, whereas another library in the list might use the CMOS delay model.

You can also use multiple libraries in the `target_library dc_shell` variable. The library that contains blocks with interface timing does not need to be included in the `target_library dc_shell` variable unless the library also contains regular ASIC cells that are not black boxes in terms of functionality.

To simulate a design containing a complex sequential block with interface timing, use the functional simulation model provided for the block.

Interpreting Timing Relationships

Synthesis tools interpret each of the timing relationships in the following manner:

Setup or hold arcs

A setup or hold arc between an input pin (clock or data) and a clock edge (rising or falling) implies the existence of an edge-triggered flip-flop with the setup or hold value.

If the setup and hold arcs are specified with respect to the same edge of the clock, one edge-triggered device is implied. Otherwise, two devices are implied, one for each relationship. In this case, the value of the other constraint for each device is assumed to be 0.0. For example, the value of the hold constraint for the device that checks the setup constraint is 0.0.

Note that a setup or hold relationship of an input pin with respect to a different clock input implies the existence of a separate edge-triggered device.

Output delay and clock edge

An output delay specified with respect to a clock edge implies an edge-triggered device clocked by that clock edge. An output delay arc specified with respect to a different clock pin implies the existence of a separate device (a multistage storage device).

Output delay and clock edge

An output delay specified with respect to a clock edge implies an edge-triggered device clocked by that clock edge. An output delay arc specified with respect to a different clock pin implies the existence of a separate device (a multistage storage device).

Output delay and nonclock input

An output delay with respect to a nonclock input implies the existence of a *direct* combinational path from the input to the output. This arc can coexist with other arcs from clock pins to the same output. Transparent devices and, therefore, time borrowing are not supported.

Other sequential arcs

Synthesis tools do not observe other types of sequential arcs (such as clear, preset, recovery, and removal).

Examples Using Interface Timing Specification

The following example shows how to specify the interface timing of a typical RAM, using the interface timing specification. Figure 10-8 and Figure 10-9 show the timing relationships of the RAM read and write cycles.

Figure 10-8 Read Cycle Timing of the RAM (WR = 1)

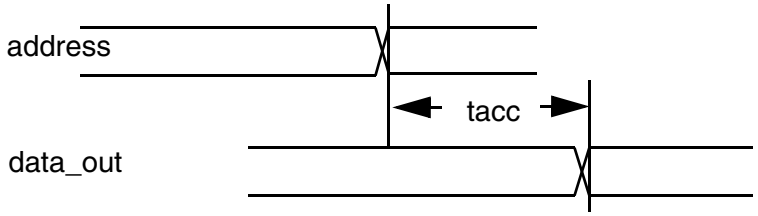
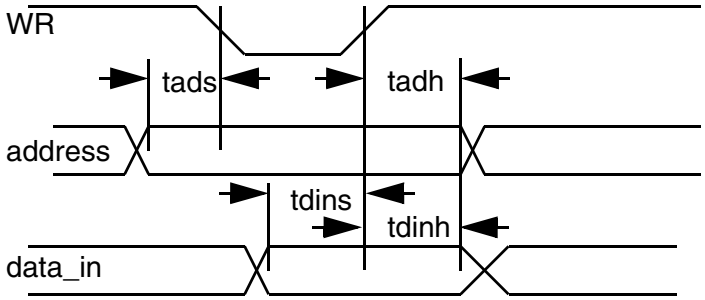


Figure 10-9 Write Cycle Timing of the RAM (WR = 0)



Example 10-7 describes the Library Compiler model of the RAM. Note that the delay between the output data and the address in the read cycle is modeled as a combinational delay. Also note that the WR signal is labeled as a clock.

Example 10-7 Library Compiler Model of RAM in Figure 10-8 and Figure 10-9

```
cell (RAM_10_8) {
   area : 2300.000000;
   interface_timing : TRUE;
   /* Apply interface timing semantics */
   bus (address) {
      bus_type  : "bus10"; /* defined in the library */
      direction : input;
      capacitance : 1.46;
      fanout_load : 1.46;
      timing () {
         timing_type : setup_falling;
         intrinsic_rise : 3.20; /* tads */
         intrinsic_fall : 3.20;
         related_pin : "WR";
      }
      timing () {
         timing_type : hold_rising;
         intrinsic_rise : 1.85;   /* tadh */
         intrinsic_fall : 1.85;
         related_pin : "WR";
      }
   }

   bus (data_in) {
      bus_type   : "bus8"
      direction : input;
      capacitance : 1.46;
      fanout_load : 1.46;
      timing () {
         timing_type : setup_rising;
         intrinsic_rise : 1.50; /* tdis */
         intrinsic_fall : 1.50;
```

```
            related_pin : "WR";
         }
         timing () {
            timing_type : hold_rising;
            intrinsic_rise : 0.65;   /* tdih */
            intrinsic_fall : 0.65;
            related_pin : "WR";
         }
      }
      pin (WR) {
         direction : input;
         capacitance : 1.13;
         fanout_load : 1.13;
         clock : true; /* WR pulse is modeled as a clock */
      }

      bus(data_out){
         bus_type  : "bus8";
         direction : output;
         timing () {
            timing_sense : non_unate;/* combinational delay */
            intrinsic_rise : 5.25;    /* tacc*/
            rise_resistance : 0.020;
            intrinsic_fall : 5.50;
            fall_resistance : 0.017;
            related_bus_pins : "address";
         }
      }
   } /* end cell */
```

Example 10-8 shows a memory model in the interface timing specification format that contains modes.

Example 10-8 Memory Model Showing the Use of Modes.

```
   library("ITS_LIB") {
      technology("cmos");
      delay_model : "table_lookup";
      time_unit : "1ps";

      lu_table_template("slewload") {
         variable_1 : "input_net_transition";
         index_1("1.0, 2.0, 3.0");
         variable_2 : "total_output_net_capacitance";
         index_2("1.0, 2.0, 3.0");
      }

      lu_table_template("slewcons") {
         variable_1 : "constrained_pin_transition";
         index_1("1.0, 2.0, 3.0";
         variable_2 : "related_pin_transition";
         index_2("1.0, 2.0, 3.0");
      }
```

```
nom_voltage : 3.0;
nom_temperature : 100.0;
type("bus_type1") {
   base_type : array;
   data_type : bit;
   bit_from : 1;
   bit_to : 0;
}

cell(ram_core) {
   interface_timing : true;
   area : 100.0;
   mode_definition("ram_modes") {
      mode_value(read1) {
      sdf_cond : "( RWN1 == 1 && CSN1 == 0 )";
      when : "(RWN1 & CSN1')";
   }
}

bus("DI1") {
   bus_type : bus_type1;
   direction : input;
   timing() {
      timing_type : "setup_rising";
      sdf_cond : "CSN1 == 0";
      when : "CSN1'";
      rise_constraint("slewcons") {...}
      fall_constraint("slewcons") {...}
      related_pin : "RWN1";
   }
   timing() {
      timing_type : "setup_rising";
      sdf_cond : "RWN1 == 0";
      when : "RWN1'";
      rise_constraint("slewcons") {...}
      fall_constraint("slewcons") {...}
      related_pin : "CSN1";
   }

   pin("DI1[1] DI1[0]") {
      direction : input;
      capacitance : 2.0;
      timing() {
         timing_type : "hold_rising";
         sdf_cond : "CSN1 == 0";
         when : "CSN1'";
         related_pin : "RWN1";

         rise_constraint("slewcons") {...}
         fall_constraint("slewcons") {...}
      }

      timing() {
```

```
      timing_type : "hold_rising";
      sdf_cond : "RWN1 == 0";
      when : "RWN1'";
      related_pin : "CSN1";

      rise_constraint("slewcons") {...}
      fall_constraint("slewcons") {...}
   }
}

bus("A1") {
   bus_type : bus_type1;
   direction : input;
   timing() {
      ........
   }
}
bus("DO1") {
   bus_type : bus_type1;
   direction : output;
   timing() {
      timing_sense : "positive_unate";
      mode(ram_modes, "read1");
      rise_transition("slewload") {...}
      fall_transition("slewload") {...}
      cell_rise("slewload") {...}
      cell_fall("slewload") {...}
      related_bus_pins : "A1";
   }
   timing() {
      timing_type : "falling_edge";
      sdf_cond : "RWN1 == 0";
      when : "RWN1'";
      rise_transition("slewload") {...}
      fall_transition("slewload") {...}
      cell_rise("slewload") {...}
      cell_fall("slewload") {...}
      related_pin : "CSN1";
   }
}

bus("DI2") {
   bus_type : bus_type1;
   direction : input;
   timing() {
      ..........
   }
}

bus("A2") {
   timing() {
      ........
   }
```

```
        }
        bus("DO2") {
           bus_type : bus_type11;
           direction : output;

           timing() {
              ............
           }
        }

        pin("RWN1") {
           direction : input;
           capacitance : 5.0;
           clock : true;
           min_pulse_width_low : 1.0;

           timing() {
              ..................
           }
        }

        pin("CSN1") {
           direction : input;
           capacitance : 1.0;
           clock : true;
           min_pulse_width_low : 1.0;

           timing() {
              ..............
           }
        }

        pin("RWN2") {
           direction : input;
           capacitance : 5.0;
           clock : true;
           min_pulse_width_low : 1.0;
           timing() {
                   .........
           }
        }
        pin("CSN2") {
           direction : input;
           capacitance : 1.0;
           clock : true;
           min_pulse_width_low : 2.0;
           timing() {
              ..........
           }
        }

     }
  }
```

11

Building Environments

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

Library Compiler gives you the ability to model these variations. Then Design Compiler can use the models to modify the synthesis and optimization environment.

To create these models, you use environment attribute statements in a technology library. All environment attributes have built-in default settings for typical delays. If you run Design Compiler without variables, the optimization process uses the default delays. Alternatively, you can create your own default settings at the library level.

The environment attributes include various attributes and tasks, covered in the following sections:

- Library-Level Default Attributes
- Defining Operating Conditions
- Defining Power Supply Cells
- Defining Wire Load Groups
- Selecting Wire Load Groups Automatically
- Specifying Delay Scaling Attributes

Library-Level Default Attributes

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

Setting Default Cell Attributes

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

default_cell_leakage_power Simple Attribute

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

Example

```
default_cell_leakage_power : 0.5;
```

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

Library Compiler looks for a definition for both the `default_cell_leakage_power` and `default_leakage_power_density` attributes. (For details about the `default_leakage_power_density` attribute, see "default_leakage_power_density Simple Attribute" on page 11-2). In checking these attributes, Library Compiler proceeds in the following manner:

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

default_leakage_power_density Simple Attribute

This library-level attribute provides the mean static leakage power per area unit in the given technology. This attribute must be a nonnegative floating-point number. If it is not defined, this attribute defaults to 0.0.

Example

```
default_leakage_power_density : 0.5;
```

Library Compiler looks for a definition for both the `default_leakage_power_density` and `default_cell_leakage_power` attributes. (For details about the `default_cell_leakage_power` attribute, see "default_cell_leakage_power Simple Attribute" on page 11-2). In checking for these attributes, Library Compiler proceeds in the following manner:

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

Setting Default Pin Attributes

Default pin attributes apply to all pins in a library and deal with timing. How you define default timing attributes in your library depends on the timing delay model you use. The CMOS nonlinear delay model does not support default timing attributes. Use only the default attributes that apply to your timing model, which can be one of the following:

- CMOS generic delay model
- CMOS piecewise linear delay model

See Chapter 2, "Delay Models," for details of these delay models.

All Delay Models

These are the defaults that apply to all pins in a library. The attributes described in this section apply to all delay models.

`default_inout_pin_cap : `*value*$_{float}$ `;`

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

`default_input_pin_cap : `*value*$_{float}$ `;`

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

```
default_output_pin_cap : value_float ;
```

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

```
default_max_fanout : value_float ;
```

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

```
default_max_transition : value_float ;
```

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

```
default_fanout_load : value_float ;
```

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

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

Example 11-1 Default Pin Attributes for a CMOS Library

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

CMOS Generic Delay Model

These are the default timing attributes for a library that uses a CMOS generic delay model. In each attribute, *value* is a floating-point number.

```
default_inout_pin_fall_res : value_float ;
```

Sets a default value for `fall_resistance` for all I/O pins in a library.

```
default_output_pin_fall_res : value_float ;
```

Sets a default value for `fall_resistance` for all output pins in the library.

```
default_inout_pin_rise_res : value_float ;
```

Sets a default value for `rise_resistance` for all I/O pins in the library.

```
default_output_pin_rise_res : value_float ;
```

Sets a default value for `rise_resistance` for all output pins in the library.

```
default_slope_fall : value_float ;
```

Sets a default value for `slope_fall` for all output pins in the library.

`default_slope_rise : value`$_{float}$ `;`

Sets a default value for `slope_rise` for all output pins in the library.

`default_intrinsic_fall : value`$_{float}$ `;`

Sets a default value for `intrinsic_fall` for all timing arcs in the library.

`default_intrinsic_rise : value`$_{float}$ `;`

Sets a default value for `intrinsic_rise` for all timing arcs in the library.

Example 11-2 sets the default timing attributes for a generic delay model.

Example 11-2 Setting Standard Timing Default Attributes

```
library (example) {
 ...
   /* default timing attributes */
   default_inout_pin_fall_res  : 12.0;
   default_output_pin_fall_res : 12.0;
   default_inout_pin_rise_res  : 15.2;
   default_output_pin_rise_res : 15.3;
   default_intrinsic_fall      : 1.0;
   default_intrinsic_rise      : 1.0;
   ...
}
```

Piecewise Linear Delay Model

These are the default timing attributes for a library that uses a piecewise linear delay model. In each attribute, *value* is a floating-point number.

`default_rise_delay_intercept : value`$_{float}$ `;`

Sets a default value for `rise_delay_intercept` on all output pins in the library.

`default_fall_delay_intercept : value`$_{float}$ `;`

Sets a default value for `fall_delay_intercept` on all output pins in the library.

`default_rise_pin_resistance : value`$_{float}$ `;`

Sets a default value for `rise_pin_resistance` on all output pins in the library.

`default_fall_pin_resistance : value`$_{float}$ `;`

Sets a default value for `fall_pin_resistance` on all output pins in the library.

`default_intrinsic_fall : value`$_{float}$ `;`

Sets a default value for `intrinsic_fall` for all timing arcs in the library.

`default_intrinsic_rise : value`$_{float}$ `;`

Sets a default value for `intrinsic_rise` for all timing arcs in the library.

Example 11-3 shows the default timing attributes setting for a piecewise linear timing delay model.

Example 11-3 Setting Piecewise Linear Default Timing Attributes

```
library (example) {
   ...
   /* default timing attributes */
   default_rise_delay_intercept : 1.0;
   default_fall_delay_intercept : 1.0;
   default_rise_pin_resistance  : 1.5;
   default_fall_pin_resistance  : 0.4;
   default_intrinsic_fall       : 1.0;
   default_intrinsic_rise       : 1.0;
   ...
}
```

Setting Wire Load Defaults

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

default_wire_load Attribute

Assigns the default values to the `wire_load` group, unless you assign a different value for `wire_load` before compiling the design. You can select a `wire_load` automatically by using `wire_load_selection`. This allows Design Compiler to select a `wire_load` group automatically and use it for wire load estimation.

Syntax

```
default_wire_load : wire_load_name;
```

Example

```
default_wire_load : WL1;
```

Synchronizing Design and Model Modes

The calculations the synthesis tool uses to develop net delays depend, in part, on the design's wire load mode. When you attach a wire load mode to a design, that new mode must match the sampling mode that the semiconductor vendor or floorplan manager used when creating the design's wire load model.

default_wire_load_capacitance Attribute

Specifies a value for the default wire load capacitance.

Syntax

```
default_wire_load_capacitance : value;
```

Example

```
default_wire_load_capacitance : .05;
```

default_wire_load_resistance Attribute

Specifies a value for the default wire load resistance.

Syntax

```
default_wire_load_resistance : value;
```

Example

```
default_wire_load_resistance : .067;
```

default_wire_load_area Attribute

Specifies a value for the default wire load area.

Syntax

```
default_wire_load_resistance : value;
```

Example

```
default_wire_load_area : 0.33;
```

Setting Other Environment Defaults

Use the following library-level attributes to set other environment defaults.

default_max_utilization Attribute

Defines the upper limit placed on the utilization of a chip.

Syntax

```
default_max_utilization : value;
```

Example

```
default_max_utilization : 0.08;
```

Utilization is the percentage of total die size taken up by the total cell area. For example, if the total cell area is 100,000 and the total die size on which these cells will be placed is 150,000, then utilization is 66.6 percent. The utilization of a chip implicitly defines how much room there is for routing between layers of silicon.

Generally, the higher the utilization you place on a chip, the more difficult a design is to route, because there is less physical area available for doing the routing.

See the Design Compiler reference manuals for more information on using `default_max_utilization`.

default_min_porosity Attribute

Defines the routability optimization constraint, with this particular library as the target library for optimizing a design for routability.

Example

```
default_min_porosity : 1.03 ;
```

You can override `default_min_porosity` by using the `set_min_porosity` command in `dc_shell`.

default_operating_conditions Attribute

Assigns a default `operating_conditions` group name for the library. It must be specified after all `operating_conditions` groups. If this attribute is not used, nominal operating conditions apply. See "Defining Operating Conditions" on page 11-10.

Syntax

```
default_operating_conditions : operating_condition_name;
```

Example

```
default_operating_conditions : WCCOM1;
```

default_connection_class Attribute

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

Example

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

Examples of Library-Level Default Attributes

Example 11-4 illustrates a library-level default attribute setting for a CMOS library. The `wire_load` and `operating_conditions` group statements illustrate the requirement that group names that are referred to by the default attributes, such as WL1 and OP1, must be defined in the library.

Example 11-4 Setting Library-Level Defaults for a CMOS Library

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

   default_cell_leakage_power :  0.2;
```

```
   /* default pin attributes */

   default_inout_pin_cap :  1.0;
   default_input_pin_cap :  1.0;
   default_intrinsic_fall :  1.0;
   default_intrinsic_rise :  1.0;
   default_output_pin_cap :  0.0;
   default_slope_fall :  0.0;
   default_slope_rise :  0.0;
   default_fanout_load :  1.0;
   default_max_fanout : 10.0;

   /* default timing attributes */

   default_inout_pin_fall_res : 0.2;
   default_output_pin_fall_res : 0.2;
   default_inout_pin_rise_res : 0.33;
   default_output_pin_rise_res : 0.4;

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

em_temp_degradation_factor Attribute

Specifies the electromigration temperature exponential degradation factor. The factor is specified as a floating-point number.

If this optional attribute is not defined, the nominal temperature electromigration tables are used for all operating temperatures.

Syntax

```
em_temp_degradation_factor : "float" ;
```

float

A floating-point number in centigrade units consistent with other temperature specifications throughout the library.

Example

```
em_temp_degradation_factor : 40.0 ;
```

Defining Operating Conditions

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

operating_conditions Group

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

Syntax

```
library ( lib_name ) {operating_conditions ( name ) {... operating conditions description ...}}
```

name

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

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

`calc_mode` : *name* ;

An optional attribute that you use to identify the associated process mode.

`power_rail` (*name* , *value*)

Identifies all power supplies that have the nominal operating conditions (defined in the `operating_conditions` group) and the nominal voltage values.

`process` : *multiplier* ;

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

`temperature` : *value* ;

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

`voltage` : *value* ;

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

Note:
Define voltage units consistently. See the section on the `voltage_unit` attribute in Chapter 6 in *Library Compiler Methodology and Modeling Functionality in Technology Libraries User Guide* for more information.

`tree_type` : *model* ;

The definition for the environment interconnect model. Design Compiler uses the interconnect model to select the formula for calculating interconnect delays. The model is one of the following three models:

- *best_case_tree*

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

- *balanced_tree*

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

- *worst_case_tree*

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

For additional information about the interconnect models, see “Interconnect Delay” on page 3-11.

Determining Operating Conditions

Design Compiler determines which operating conditions to use, according to the following hierarchy:

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

Defining Operating Conditions at Runtime

To define an `operating_conditions` group at runtime, use the `set_operating_conditions` command in dc_shell.

Syntax

```
set_operating_conditions operating_cond_name
```

operating_cond_name

Specifies the name of an `operating_conditions` group defined in the library. For example, to use the operating conditions group `MY_BEST`, enter

```
dc_shell> set_operating_conditions MY_BEST
```

For additional information, see the `set_operating_conditions` command in the Synopsys man pages.

timing_range Group

A `timing_range` group models statistical fluctuations in the defined operating conditions defined for your design during the optimization process. A `timing_range` group is defined at the library-group level:

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

A `timing_range` group defines two multipliers that scale the signal arrival times computed by the timing analyzer when it evaluates timing constraints. In the following attributes, *multiplier* is a floating-point number:

`faster_factor` : *multiplier* ;

Scaling factor applied to the signal arrival time to model the fastest-possible arrival time.

`slower_factor` : *multiplier* ;

Scaling factor applied to the signal arrival time to model the slowest-possible arrival time.

Example 11-5 describes the SLOW_RANGE and FAST_RANGE timing ranges.

Example 11-5 Defining Timing Ranges

```
library (example) {
   ...
  timing_range (SLOW_RANGE) {
    faster_factor : 1.05;
    slower_factor : 1.2;
```

```
  }
  timing_range (FAST_RANGE)  {
    faster_factor : 0.90;
    slower_factor : 0.96;
  }
  ...
}
```

In the SLOW_RANGE timing range, the possible delays are from 1.05 to 1.2 times the delay calculated by the timing analyzer. In the FAST_RANGE timing range, the possible delays are 0.90 to 0.96 times the delay calculated by the timing analyzer.

Defining Timing Ranges at Runtime

You can define one or two timing ranges in dc_shell to use when optimizing a design in Design Compiler. The `set_timing_ranges` command defines the timing ranges used in the design.

Syntax

`set_timing_ranges ( range_name [, range_name2] )`

Note:

The modeling method previously described produces accurate results in cases in which all the delays have a linear dependency on the operating parameters. In cases in which delays do not scale linearly with operating conditions, this method provides only a first-order approximation. Remember that timing ranges influence constraint evaluation only; they do not affect timing reports in Design Compiler. You can obtain information on timing ranges that are defined in a library by using the `report_lib` command.

Following is an example of the timing-range information from the `set_timing_ranges` command. This example illustrates two ranges, FAST_RANGE and SLOW_RANGE, with their respective limits.

```
Timing Ranges:

Name         Library    Slower factor    Faster factor
-----------------------------------------------------
FAST_RANGE   test           0.9600           0.9000
SLOW_RANGE   test           1.2000           1.0500
```

Defining Power Supply Cells

Use the `power_supply` group to model multiple power supply cells.

power_supply group

The `power_supply` group captures all nominal information on voltage variation. It is defined before the `operating_conditions` group and before the `cell` groups. All the power supply names defined in the `power_supply` group exist in the `operating_conditions` group.

Define the `power_supply` group at the library level.

Syntax

```
power_supply () {default_power_rail : string ;power_rail (string, float)
;  power_rail (string, float) ;     ...}
```

Example

```
power_supply () {
   default_power_rail ; VDD0;
   power_rail (VDD1, 5.0) ;
   power_rail (VDD2, 3.3) ;
}
```

Defining Wire Load Groups

Use the `wire_load` group and the `wire_load_selection` group to specify values for the `capacitance` factor, `resistance` factor, `area` factor, `slope`, and `fanout_length` you want to apply to the wire delay model for different sizes of circuitry.

wire_load Group

The `wire_load` group has an extended `fanout_length` complex attribute.

Define the `wire_load` group at the library level.

Syntax

```
wire_load(name){resistance : value ; capacitance : value ; area : value ;
   slope : value ;fanout_length(fanout_int, length_float,\
average_capacitance_float,standard_deviation_float,\
number_of_nets_int);}
```

A `wire_load` group contains all the information Design Compiler needs in order to estimate interconnect wiring delays. In a `wire_load` group, you define the estimated wire length as a function of fanout. You can also define scaling factors to derive wire resistance, capacitance, and area from a given length of wire.

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

You can define the following simple attributes in a `wire_load` group:

`resistance` : *value* ;

Specifies a floating-point number representing wire resistance per unit length of interconnect wire.

`capacitance` : *value* ;

Specifies a floating-point number representing capacitance per unit length of interconnect wire.

`area` : *value* ;

Specifies a floating-point number representing the area per unit length of interconnect wire.

`slope` : *value* ;

Specifies a floating-point number representing slope. This attribute characterizes linear fanout length behavior beyond the scope of the longest length described by the `fanout_length` attributes.

You can define the following complex attribute in a `wire_load` group:

`fanout_length` ($fanout_{int}$, $length_{float}$, $average\_capacitance_{float}$ \ $standard\_deviation_{float}$, $number\_of\_nets_{int}$);

`fanout_length` is a complex attribute that defines values that represent fanout and length. The *fanout* value is an integer; *length* is a floating-point number.

The Floorplan Manager tool is the only Synopsys product that uses `average_capacitance`, `standard_deviation`, and `number_of_nets`. For more information, see the *Floorplan Manager User Guide*.

When you create a wire load manually, define only *fanout* and *length*. When you generate the wire load with the `create_wire_load` command, the *fanout* and *length* values are generated automatically.

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

When Design Compiler encounters multiple *fanout* and *length* points, it uses linear interpolation to determine points between them, and extrapolation beyond the last point.

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

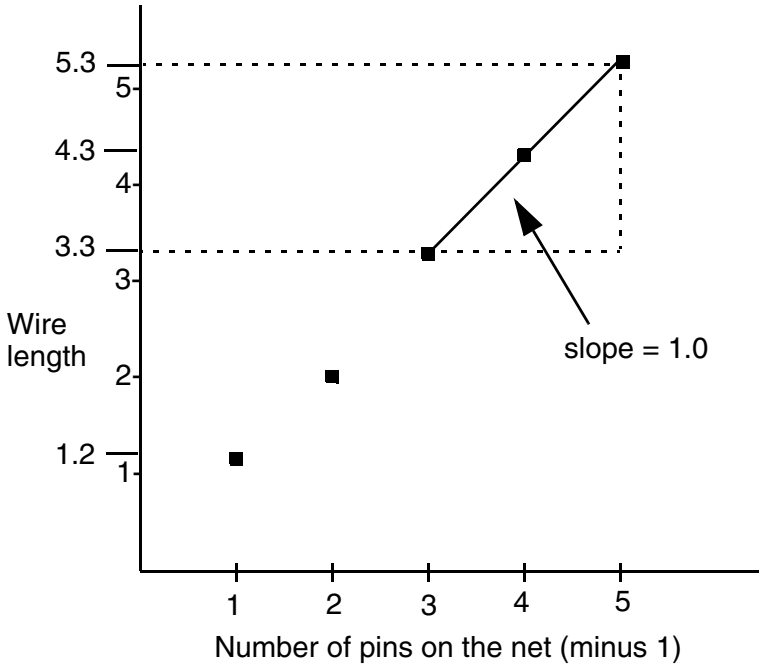
The `interconnect_delay` group specifies the lookup table template and the wire delay values.

Specify the `interconnect_delay` values as you would for any Library Compiler table.

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

Figure 11-1 illustrates the correlation between the number of pins on the net, excluding the driver pin, and the estimated wire length of the metal between all the pins.

Figure 11-1 Wire Load Definition Graph



Example 11-6 gives the library description for the model in Figure 11-1.

Note:

In Example 11-6 and Example 11-7, the name 90x90 is enclosed in quotation marks because it is a string that begins with a number.

Example 11-6 Wire Load Definition

```
library (example) {
 ...
  wire_load ("90x90") {
     resistance : 0;
     capacitance : 1;
```

```
      area : 0;
      slope : 1.0;
      fanout_length( 1, 1.2 );
      fanout_length( 2, 2.0 );
      fanout_length( 3, 3.3 );
   }
}
```

Example 11-7 shows how to estimate the wire delay by using a 3-D lookup table.

Example 11-7 Wire Delay Estimation, Using 3-D Lookup Table

```
library (example) {
 ...
   lu_table_template (wire_delay_table_template) {
      variable_1 : fanout_number;
      variable_2 : fanout_pin_capacitance;
      variable_3 : driver_slew;
      index_1( "0.12,3.4");
      index_2( "0.12,4.24" );
      index_3( "0.1,2.7,3.12" );
   }
}
```

You can define one `wire_load` group per design at Design Compiler runtime. This is the syntax for the `set_wire_load` command in dc_shell :

```
set_wire_load wire_load_name [-library][-mode]
```

The *wire_load_name* is the name of a `wire_load` group defined in the library. To use the `wire_load` group defined in Example 11-6, enter

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

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

If you don't define a `wire_load` group, Library Compiler uses the `default_wire_load_resistance` and `default_wire_load_capacitance` attribute values, respectively, for wire resistance and wire capacitance.

The `report_lib` command reports the information for `default_wire_load_capacitance`, `default_wire_load_resistance` , and `default_wire_load_area`. The additional wire load information is reported in the Library Compiler report if it is available.

Example 11-8 shows the report for a wire load model containing new *fanout* and *length* information:

Example 11-8 Wire Load Model Containing New Fanout Length Information

```
Wire Loading Model:
Name       : 05x05
Location   : wl library_name
```

```
Resistance : 0
Capacitance: 1
Area       : 0
Slope      : 0.186

Fanout   Length     Points       AverageCap       StdDeviation
------------------------------------------------------------
1        0.39         50          1.30               0.02
```

Calculating Wire Area

Library Compiler calculates wire area by using one of three methods:

- Using values specified when the fanout values correspond to one of the values specified in the model
- Extrapolation
- Interpolation

Example 11-9 shows a wire load model.

Example 11-9 Wire Load Model

```
wire_load(standard) {
     resistance : 0.001;
     capacitance : 1.2 ;
     area : 0.5;
     slope : 0.311 ;
     fanout_length(1,0.53);
     fanout_length(2,0.63);
     fanout_length(5,0.83);
}
```

Using Values Specified

In the example, the area value that is specified in the wire load model is the area/unit length of wire. The `fanout_length` is an estimation of the wire length for a given number of fanouts. So if the driving cell has a fanout of 2, area is calculated as

```
Net area = area/unit_length x fanout_length(for given fanout)
         = 0.5 x 0.63 = 0.315
```

Extrapolation

For fanout greater than the largest fanout in the table, Library Compiler uses the slope (specified in the wire load model) and the largest fanout values. In the example, the fanout is 6, so area is calculated as

```
Net area[n] =  Net area[last fanout] + ((fanout-last fanout) x
                slope x area/unit_length)
Net area[6] = {(Net area[5]) + ( (6-5) x 0.311 x 0.5 )}
Net area[5] =  Area/unit_length X fanout_length (for fanout of 5)
```

```
       =  0.5 x 0.83 = 0.42
```

Therefore, the net area for fanout 6 is

```
Net area[6]   =   {(0.42) + (1 x 0.311 x 0.5)} = 0.5755
```

For other fanout values, the calculation is

```
Net area[7]   =   {(0.42) + (2 x 0.311 x 0.5)}
Net area[8]   =   {(0.42) + (3 x 0.311 x 0.5)}
```

or

```
Net area[7]   =   {(0.5755) + (1 x 0.311 x 0.5)}
```

Interpolation

When values are missing from the model, Library Compiler uses interpolation. For a fanout of 3, there is no value in the model, so the calculation is

```
Slope[between 2 and 5] = (length[5]-length[2]) / (5-2)
                        = (0.83-0.63) / (5-2) = 0.0666
Net area[2] = 0.5 x 0.63 = 0.32
Net area[n] = Net area[last fanout] + ((fanout-last fanout) x
              slope x area/unit_length)
Net area[3] = Net area[2] + ((3-2) x slope[between 2 and 5] x
                           area/unit_length)
Net area[3] = 0.32 + (1 x 0.0666 x 0.5) = 0.3533
```

The two fanouts closest to 3 are 2 and 5. Library Compiler calculates the slope between fanout 2 and 5.

For a net area of 4, the calculation is

```
Net area[4]  = { 0.32  + (2 x 0.0666 x 0.5)} = 0.3866
```

or

```
Net area[4]  = { 0.3533  + (1 x 0.0666 x 0.5)} = 0.3866
```

wire_load_table Group

You can use the `wire_load_table` group to estimate accurate connect delay. Compared to the `wire_load` group, this group is more flexible, because wire capacitance and resistance no longer have to be strictly proportional to each other. In some cases, this results in more-accurate connect delay estimates.

Syntax

```
wire_load_table(name_string) {fanout_length(fanout_int, length_float);
fanout_capacitance(fanout_int, capacitance_float);
fanout_resistance(fanout_int, resistance_float);fanout_area(fanout_int,
area_float);}
```

In the `wire_load` group, the `fanout_capacitance` , `fanout_resistance` , and `fanout_area` values represent per-length coefficients. In the `wire_load_table` group the values are exact.

The `report_lib` command reports the `wire_load_table` group. Example 11-10 shows a source file containing a `wire_load_table` group and the report for the table:

Example 11-10 Wire_load_table With the Table Report

```
library(wlut) {
   wire_load_table("05x05") {
      fanout_length(1, 0.2) ;
      fanout_capacitance(1, 0.15);
      fanout_resistance(1, 0.17) ;
      fanout_area(1, 0.2) ;
      fanout_length(2, 0.35) ;
      fanout_capacitance(2, 0.39) ;
      fanout_resistance(2, 0.25) ;
      fanout_area(2, 0.41) ;
   }
}

Name          :   05x05
Location      :   wlut
Fanout       Length      Capacitance    Resistance     Area
------------------------------------------------------------
1                 0.2       0.15          0.17          0.2
2                 0.35      0.39          0.25          0.41
```

Selecting Wire Load Groups Automatically

The `wire_load_selection` groups that are defined at the `library` group level let Design Compiler select a `wire_load` group for wire load estimation automatically. Selection is based on the total cell area of the design.

You use the `selection_group` option of the `set_wire_load` command to determine the `wire_load_selection` group. If no group is selected, the `default_wire_load_selection` library variable determines the default `wire_load_selection` group.

Typically, definitions for `wire_load` groups are built according to statistical experiments with blocks of different sizes. The `wire_load_selection` group incorporates this information in the technology library. Design Compiler uses the information during compilation to achieve better wiring-area and delay-estimation accuracy.

Use multiple selection groups (see Example 11-11) to allow for different amounts of wire load pessimism or in cases where the same library is used for different fabrication processes.

Example 11-11 Specification of Multiple wire_load_selection Groups

```
wire_load_selection(really_pessimistic) {
   wire_load_from_area(min_area1,max_area1,wire_load_name1);
   wire_load_from_area(min_area2,max_area2,wire_load_name2);
}

wire_load_selection(somewhat_pessimistic) {
   wire_load_from_area(min_area3,max_area3,wire_load_name3);
   wire_load_from_area(min_area4,max_area4,wire_load_name4);
}
default_wire_load_selection : somewhat_pessimistic;
```

With multiple `wire_load_selection` groups, you can put two-layer as well as three-layer wire load models into a single library. Without multiple `wire_load_selection` groups, you must update your library with `update_lib` if you use a library compiled with a single `wire_load_selection` group and want to use a different group. When using multiple `wire_load_selection` groups, you must include the `default_wire_load_selection` library variable to determine the default `wire_load_selection` group.

Within a given `wire_load_selection` group, use `wire_load_from_area` to associate each `wire_load` group with a different area range. This practice is useful for compiling different levels of the design hierarchy separately.

wire_load_from_area Attribute

Use `wire_load_from_area` to associate each `wire_load` group with a different area range. This practice is useful for compiling different levels of the design hierarchy separately.

This attribute must be specified after all `wire_load` groups.

Syntax

```
wire_load_selection(name1) {
   wire_load_from_area(min_area1,max_area1,wire_load_name1);
   min_area2,max_area2,wire_load_name2
}
wire_load_selection(name-n) {
   ...
}
```

The *min_area* and *max_area* values give the area range in library cell area units. This range must not overlap a range defined in another `wire_load_from_area` line. The `wire_load_name` is the name of the `wire_load` group to use when the area of the design falls within the defined range, as shown in Example 11-12. A design with an area outside the listed range is assigned the `wire_load` group of the next area range. In Example 11-12, a design with an area of 120 gets the 10x10 `wire_load` group.

Example 11-12 Wire Load Group Selection

```
wire_load_selection(name1) {
      wire_load_from_area(0,100,"05x05");
      wire_load_from_area(150,200,"10x10");
}
```

In Example 11-13, a design with an area of 20 gets the 05x05 `wire_load` group.

Example 11-13 Wire Load Group Selection

```
wire_load_selection(name1)
  wire_load_from_area(50,100,"05x05");
}
```

Specifying Default Wire Load Settings

It is practical to set the `default_wire_load_mode` to enclosed or segmented instead of top. If the `default_wire_load_mode` is set to top, all nets in both the top design and subblocks use the wire load model selected from the area of the top design. If the `default_wire_load_mode` is set to enclosed, the nets fully enclosed within a design use the wire load model selected from the area of that subdesign. If the `default_wire_load_mode` is set to segmented, the nets partially enclosed within a design use the wire load model selected from the area of that subdesign.

Although the previous mechanism for defining a single `wire_load_selection` group is available, it is the exception. Use the `default_wire_load_selection` attributes at the library level when you want to specify more than one group:

```
default_wire_load_selection : name1;
```

Until the design is completely mapped, the total area is unknown. Therefore, Design Compiler uses the `default_wire_load` value, if defined, or none if not defined. Design Compiler selects the wire load model to use before and during timing operations and operations that modify the design, such as update_timing, report_timing, compile, and translate.

You can override the Design Compiler `wire_load_model` selection with the `set_wire_load` command. You can turn automatic selection of `wire_load_model` off when you are working in dc_shell, by setting the `auto_wire_load_selection` to false. For example, in dc_shell, enter

```
dc_shell> auto_wire_load_selection = false
```

You can also turn automatic selection of the `wire_load_model` off by modifying the .synopsys_dc.setup file. Set the `auto_wire_load_selection` to false.

See the Synopsys man pages for more information about `auto_wire_load_selection`.

The `wire_load_model` Design Compiler selects is reported by `report_lib`, which lists available wire loads in the library. The compile log reports changes in the wire load model with an informational message.

Specifying Delay Scaling Attributes

Design Compiler calculates delay estimates by using the scaling factors set in the technology library environment. These k-factors (attributes that begin with k_) are multipliers that scale defined library values, taking into consideration the effects of changes in process, temperature, and voltage.

Library Compiler models the effects of process, temperature, and voltage variations on circuit timing, using the following:

- k-factors that apply to the entire library and are defined at the library level. Library Compiler assigns a value of 0 (zero) to any k-factors not defined in the library.
- User-selected operating conditions that override the values in the library for an individual cell (see “Scaling Factors for Individual Cells” on page 11-38).

The scaling factors you define for your library depend on the timing delay model you use.

The following k-factors are specific to timing delay models:

- Intrinsic delay factors
- Slope sensitivity factors (CMOS generic delay model)
- Drive capability factors (CMOS generic delay model)
- Pin and wire capacitance factors
- Wire resistance factors
- Pin resistance factors (CMOS piecewise linear delay model)
- Intercept delay factors (CMOS piecewise linear delay model)
- Power scaling factors

  Note:
  Scaling factors have no effect on the scalable polynomial delay model.

- Timing constraint factors

Using these values, Design Compiler uniformly scales all timing numbers for timing analysis. See Chapter 3, "Delay Models," for descriptions of the delay analysis equations Design Compiler uses.

Calculating Delay Factors

Any delay equation factor not affected by process, temperature, or voltage must have the corresponding k-factor set to 0. The default value for any unspecified k-factor is 0.

Calculating Voltage Delay Factors

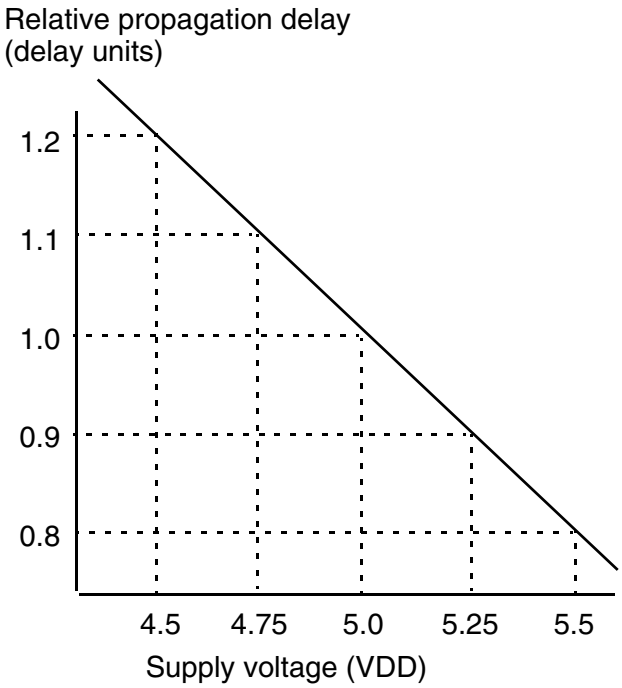
Figure 11-2 illustrates the relationship between supply voltage and propagated delay for a typical technology library.

Figure 11-2 shows that as voltage increases, the relative propagation delay scaling decreases. For example, at 5.25 volts, the delay scaling is 0.90. Determine the k-factors related to voltage by calculating the slope of the line as follows:

```
k_volt = delay scaling units / volt
       = (0.80 - 1.20) / (5.50 - 4.50)
       = -0.4
```

If the slope of the line is –0.4 delay units per volt, the value of the k_volt delay factor is –0.4.

Figure 11-2 Propagation Delay as a Function of Supply Voltage



Calculating Temperature Delay Factors

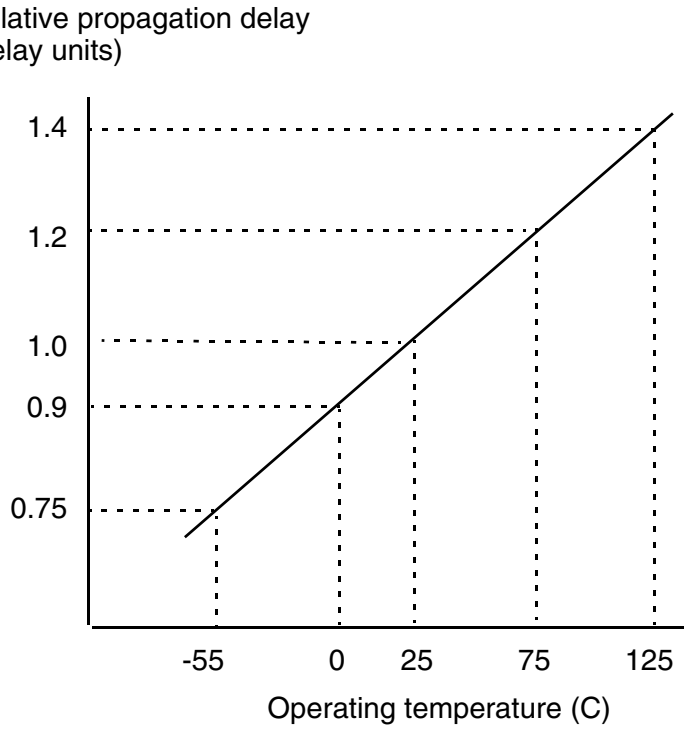
Figure 11-3 illustrates the relationship between operating temperature and propagated delay for a typical technology library.
The figure shows that the scaling factor for the delays in the timing equation increases as temperature increases.

Determine the k-factor values related to temperature by calculating the slope of the line as follows:

```
k_temp = delay scaling units / degree Centigrade
       = (1.4 - 0.9) / (125 - 0)
       = 0.004
```

If the slope of the line is 0.004 delay units per degree centigrade, the value of the k_temp delay factor is 0.004.

Figure 11-3 Propagation Delay as a Function of Temperature



Assigning Process Delay Factors

You scale for process by arbitrarily assigning numbers based on the effects of fabrication on timing. There are usually three process-scaling factors available for a library: best case (value less than 1), nominal (value equal or close to 1.0), and worst case (value greater than 1).

Because process-scaling factors are unitless, no graphical representation or slope value is used for process k-factors. The portions of the delay equation affected by process scaling should have a value of 1.

Setting Combined Scaling Factor

According to the technology, you might want to calculate a single scaling factor value to account for the voltage, temperature, and process effects. To implement this type of scaling,

1. Set k-factors for voltage and temperature to 0, which nullifies the scaling effects of voltage and temperature.
2. Set the k-factors for the parts of the delay equation you want to scale to 1.0.

3. Set the `nom_process` attribute value to 1.0. In this case, the `nom_voltage` and `nom_temperature` attribute settings do not matter, because both the voltage and temperature scaling effects are ignored.
4. In the `operating_conditions` groups, set the combined process, temperature, and voltage scaling factor.

Intrinsic Delay Factors

Intrinsic delay factors scale the intrinsic delay according to process, temperature, and voltage variations. Each attribute gives the multiplier for a certain portion of the intrinsic-rise delay or intrinsic-fall delay of all cells in the library. In the following syntax, *multiplier* is a floating-point number:

`k_process_intrinsic_fall : multiplier ;`

Scaling factor applied to the intrinsic fall delay of a timing arc to model process variation.

`k_process_intrinsic_rise : multiplier ;`

Scaling factor applied to the intrinsic rise delay of a timing arc to model process variation.

`k_temp_intrinsic_fall : multiplier ;`

Scaling factor applied to the intrinsic fall delay of a timing arc to model temperature variation.

`k_temp_intrinsic_rise : multiplier ;`

Scaling factor applied to the intrinsic rise delay of a timing arc to model temperature variation.

`k_volt_intrinsic_fall : multiplier ;`

Scaling factor applied to the intrinsic fall delay of a timing arc to model voltage variation.

`k_volt_intrinsic_rise : multiplier ;`

Scaling factor applied to the intrinsic rise delay of a timing arc to model voltage variation.

Slope Sensitivity Factors

You can define slope sensitivity factors only in a library using a CMOS linear delay model. See Chapter 2, “Delay Models,” for details on CMOS linear delay analysis equations.

The slope sensitivity factors scale the delay slope according to process, temperature, and voltage variations. Each attribute gives the multiplier for a certain portion of the rise-delay slope or fall-delay slope of all cells in the library. In the following syntax, *multiplier* is a floating-point number:

`k_process_slope_fall : multiplier ;`

Scaling factor applied to timing arc fall slope sensitivity to model process variation.

`k_process_slope_rise` : multiplier ;

Scaling factor applied to timing arc rise slope sensitivity to model process variation.

`k_temp_slope_fall` : multiplier ;

Scaling factor applied to timing arc fall slope sensitivity to model temperature variation.

`k_temp_slope_rise : multiplier ;`

Scaling factor applied to timing arc rise slope sensitivity to model temperature variation.

`k_volt_slope_fall : multiplier ;`

Scaling factor applied to timing arc fall slope sensitivity to model voltage variation.

`k_volt_slope_rise : multiplier ;`

Scaling factor applied to timing arc rise slope sensitivity to model voltage variation.

Drive Capability Factors

You can define drive capability factors only in a library using a CMOS linear delay model. See Chapter 3, "Delay Models," for details on CMOS linear delay analysis equations.

The drive capability factors scale the drive capability of a pin according to process, temperature, and voltage variations. Each attribute gives the multiplier for a certain portion of a pin's drive capability in rise-delay or fall-delay analysis. In the following syntax, *multiplier* is a floating-point number:

`k_process_drive_current : multiplier ;`

Scaling factor applied to timing arc `drive_current` to model process variation.

`k_process_drive_fall : multiplier ;`

Scaling factor applied to timing arc `fall_resistance` to model process variation.

`k_process_drive_rise : multiplier ;`

Scaling factor applied to timing arc `rise_resistance` to model process variation.

`k_temp_drive_current : multiplier ;`

Scaling factor applied to timing arc `drive_current` to model temperature variation.

`k_temp_drive_fall : multiplier ;`

Scaling factor applied to timing arc `fall_resistance` to model temperature variation.

`k_temp_drive_rise : multiplier ;`

Scaling factor applied to timing arc `rise_resistance` to model temperature variation.

`k_volt_drive_current : multiplier ;`

Scaling factor applied to timing arc `drive_current` to model voltage variation.

`k_volt_drive_fall : multiplier ;`

Scaling factor applied to timing arc `fall_resistance` to model voltage variation.

`k_volt_drive_rise : multiplier ;`

Scaling factor applied to timing arc `rise_resistance` to model voltage variation.

Pin and Wire Capacitance Factors

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

`k_process_pin_cap : multiplier ;`

Scaling factor applied to pin capacitance to model process variation.

`k_process_wire_cap : multiplier ;`

Scaling factor applied to wire capacitance to model process variation.

`k_temp_pin_cap : multiplier ;`

Scaling factor applied to pin capacitance to model temperature variation.

`k_temp_wire_cap : multiplier ;`

Scaling factor applied to wire capacitance to model temperature variation.

`k_volt_pin_cap : multiplier ;`

Scaling factor applied to pin capacitance to model voltage variation.

`k_volt_wire_cap : multiplier ;`

Scaling factor applied to wire capacitance to model voltage variation.

CMOS Wire Resistance Factors

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

`k_process_wire_res` : *multiplier* ;

Scaling factor applied to wire resistance to model process variation.

`k_temp_wire_res : multiplier ;`

Scaling factor applied to wire resistance to model temperature variation.

`k_volt_wire_res : multiplier ;`

Scaling factor applied to wire resistance to model voltage variation.

Pin Resistance Factors

You can define pin resistance factors only in libraries that use a CMOS piecewise linear delay model. See Chapter 3, "Delay Models," for details on CMOS piecewise linear delay analysis equations.

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

`k_process_rise_pin_resistance : multiplier ;`

Scale factor applied to `rise_pin_resistance` to model process variation.

`k_process_fall_pin_resistance : multiplier ;`

Scale factor applied to `fall_pin_resistance` to model process variation.

`k_temp_rise_pin_resistance : multiplier ;`

Scale factor applied to `rise_pin_resistance` to model temperature variation.

`k_temp_fall_pin_resistance : multiplier ;`

Scale factor applied to `fall_pin_resistance` to model temperature variation.

`k_volt_rise_pin_resistance : multiplier ;`

Scale factor applied to `rise_pin_resistance` to model voltage variation.

`k_volt_fall_pin_resistance : multiplier ;`

Scale factor applied to `fall_pin_resistance` to model voltage variation.

Intercept Delay Factors

You can define intercept delay factors only in libraries that use a CMOS piecewise linear delay model. See Chapter 3, "Delay Models," for details on CMOS piecewise linear delay analysis equations.

Intercept delay factors scale the delay intercept according to process, temperature, and voltage variations. These factors are used with slope- or intercept-type timing equations.

Each attribute gives the multiplier for a certain portion of the rise and fall intercepts of all cells in the library. In the following syntax, *multiplier* is a floating-point number:

`k_process_rise_delay_intercept : multiplier ;`

Scale factor applied to `rise_delay_intercept` to model process variation.

`k_process_fall_delay_intercept : multiplier ;`

Scale factor applied to `fall_delay_intercept` to model process variation.

`k_temp_rise_delay_intercept : multiplier ;`

Scale factor applied to `rise_delay_intercept` to model temperature variation.

`k_temp_fall_delay_intercept : multiplier ;`

Scale factor applied to `fall_delay_intercept` to model temperature variation.

`k_voltage_rise_delay_intercept : multiplier ;`

Scale factor applied to `rise_delay_intercept` to model voltage variation.

`k_voltage_fall_delay_intercept : multiplier ;`

Scale factor applied to `fall_delay_intercept` to model voltage variation.

Power Scaling Factors

You can define power factors only in libraries that use a CMOS technology. Power scaling factors scale the power computation in Design Compiler by process, temperature, and voltage. The power scaling factors are listed below. In the following syntax, *multiplier* is a floating-point number:

`k_process_cell_leakage_power : multiplier ;`

Specifies environmental derating factors for the `cell_leakage_power` attribute.

`k_process_internal_power : multiplier ;`

Specifies environmental derating factors for the `internal_power` attribute.

`k_temp_cell_leakage_power : multiplier ;`

Specifies environmental derating factors for the `cell_leakage_power` attribute.

`k_temp_internal_power : multiplier ;`

Specifies environmental derating factors for the `internal_power` attribute.

`k_volt_cell_leakage_power : multiplier ;`

Specifies environmental derating factors for the `cell_leakage_power` attribute.

`k_volt_internal_power` : *multiplier* ;

Specifies environmental derating factors for the `internal_power` attribute.

Timing Constraint Factors

Timing-constraint factors scale the following timing constraint values to account for the effects of changes in process, temperature, and voltage:

- Setup time
- Hold time
- No-change time
- Recovery time
- Removal time
- Minimum pulse width
- Minimum clock period
- Skew

An example of a timing constraint factor is `k_volt_hold_rise`, which is the scaling factor applied to hold constraints to model voltage variation. For setup, hold, and recovery time constraints, the k-factors containing the rise suffix are applied to the related `intrinsic_rise` value of the constraint `timing` group. Those with the fall suffix are applied to the related `intrinsic_fall` value.

For minimum pulse width constraints, the factors with the high suffix are applied to the `min_pulse_width_high` constraint. Those with the low suffix are applied to the `min_pulse_width_low` constraint.

In the following syntax examples, the scaling factor (multiplier) for temperature and voltage constraints is a floating-point number; for process constraints, the factor is a nonnegative floating-point number.

`k_process_hold_rise : `*`multiplier`*` ;`

Scaling factor applied to hold constraints to model process variation.

`k_process_hold_fall : `*`multiplier`*` ;`

Scaling factor applied to hold constraints to model process variation.

`k_process_removal_fall : `*`multiplier`*` ;`

Scaling factor applied to removal constraints to model process variation.

`k_process_removal_rise : `*`multiplier`*` ;`

Scaling factor applied to removal constraints to model process variation.

`k_temp_hold_rise : `*`multiplier`*` ;`

Scaling factor applied to hold constraints to model temperature variation.

`k_temp_hold_fall : multiplier ;`

Scaling factor applied to hold constraints to model temperature variation.

`k_temp_removal_fall : multiplier ;`

Scaling factor applied to removal constraints to model temperature variation.

`k_temp_removal_rise : multiplier ;`

Scaling factor applied to removal constraints to model temperature variation.

`k_volt_hold_rise : multiplier ;`

Scaling factor applied to hold constraints to model voltage variation.

`k_volt_hold_fall : multiplier ;`

Scaling factor applied to hold constraints to model voltage variation.

`k_volt_removal_fall : multiplier ;`

Scaling factor applied to removal constraints to model voltage variation.

`k_volt_removal_rise : multiplier ;`

Scaling factor applied to removal constraints to model voltage variation.

`k_process_setup_rise : multiplier ;`

Scaling factor applied to setup constraints to model process variation.

`k_process_setup_fall : multiplier ;`

Scaling factor applied to setup constraints to model process variation.

`k_temp_setup_rise : multiplier ;`

Scaling factor applied to setup constraints to model temperature variation.

`k_temp_setup_fall : multiplier ;`

Scaling factor applied to setup constraints to model temperature variation.

`k_volt_setup_rise : multiplier ;`

Scaling factor applied to setup constraints to model voltage variation.

`k_volt_setup_fall : multiplier ;`

Scaling factor applied to setup constraints to model voltage variation.

`k_process_nochange_rise : multiplier ;`

Scaling factor applied to no-change constraints to model process variation.

`k_process_nochange_fall : multiplier ;`

Scaling factor applied to no-change constraints to model process variation.

`k_temp_nochange_rise : multiplier ;`

Scaling factor applied to no-change constraints to model temperature variation.

`k_temp_nochange_fall : multiplier ;`

Scaling factor applied to no-change constraints to model temperature variation.

`k_volt_nochange_rise : multiplier ;`

Scaling factor applied to no-change constraints to model voltage variation.

`k_volt_nochange_fall : multiplier ;`

Scaling factor applied to no-change constraints to model voltage variation.

`k_process_recovery_rise : multiplier ;`

Scaling factor applied to recovery constraints to model process variation.

`k_process_recovery_fall : multiplier ;`

Scaling factor applied to recovery constraints to model process variation.

`k_temp_recovery_rise : multiplier ;`

Scaling factor applied to recovery constraints to model temperature variation.

`k_temp_recovery_fall : multiplier ;`

Scaling factor applied to recovery constraints to model temperature variation.

`k_volt_recovery_rise : multiplier ;`

Scaling factor applied to recovery constraints to model voltage variation.

`k_volt_recovery_fall : multiplier ;`

Scaling factor applied to recovery constraints to model voltage variation.

`k_process_min_pulse_width_high : multiplier ;`

Scaling factor applied to minimum pulse width constraints to model process variation.

`k_process_min_pulse_width_low : multiplier ;`

Scaling factor applied to minimum pulse width constraints to model process variation.

`k_temp_min_pulse_width_high : multiplier ;`

Scaling factor applied to minimum pulse width constraints to model temperature variation.

`k_temp_min_pulse_width_low : multiplier ;`

Scaling factor applied to minimum pulse width constraints to model temperature variation.

`k_volt_min_pulse_width_high : multiplier ;`

Scaling factor applied to minimum pulse width constraints to model voltage variation.

`k_volt_min_pulse_width_low : multiplier ;`

Scaling factor applied to minimum pulse width constraints to model voltage variation.

`k_process_min_period : multiplier ;`

Scaling factor applied to minimum period constraints to model process variation.

`k_temp_min_period : multiplier ;`

Scaling factor applied to minimum period constraints to model temperature variation.

`k_volt_min_period : multiplier ;`

Scaling factor applied to minimum period constraints to model voltage variation.

`k_process_skew_fall : multiplier ;`

Scaling factor applied to skew constraints to model process variation.

`k_process_skew_rise : multiplier ;`

Scaling factor applied to skew constraints to model process variation.

`k_temp_skew_fall : multiplier ;`

Scaling factor applied to skew constraints to model temperature variation.

`k_temp_skew_rise : multiplier ;`

Scaling factor applied to skew constraints to model temperature variation.

`k_volt_skew_fall : multiplier ;`

Scaling factor applied to skew constraints to model voltage variation.

`k_volt_skew_rise : multiplier ;`

Scaling factor applied to skew constraints to model voltage variation.

Table 11-1 shows how the constraints are calculated.

Table 11-1 Timing Constraint Equations

Setup Rise	intrinsic_rise (1 + Δvoltage k_volt_setup_rise) (1 + Δtemp k_temp_setup_rise) (1 + Δprocess k_process_setup_rise)
Setup Fall	intrinsic_fall (1 + Δvoltage k_volt_setup_fall) (1 + Δtemp k_temp_setup_fall) (1 + Δprocess k_process_setup_fall)
Hold Rise	intrinsic_rise (1 + Δvoltage k_volt_hold_rise) (1 + Δtemp k_temp_hold_rise) (1 + Δprocess k_process_hold_rise)
Hold Fall	

Table 11-1 Timing Constraint Equations (Continued)

Recovery Rise	intrinsic_rise + (1 + Δvoltage k_volt_recovery_rise) (1 + Δtemp k_temp_recovery_rise) (1 + Δprocess k_process_recovery_rise)
Recovery Fall	intrinsic_fall + (1 + Δvoltage k_volt_recovery_fall) (1 + Δtemp k_temp_recovery_fall) (1 + Δprocess k_process_recovery_fall)
Minimum Pulse Width High	min_pulse_width_high + (1 + Δvoltage k_volt_min_pulse_width_high) (1 + Δtemp k_temp_min_pulse_width_high) (1 + Δprocess k_process_min_pulse_width_high)
Minimum Pulse Width Low	min_pulse_width_low + (1 + Δvoltage k_volt_min_pulse_width_low) (1 + Δtemp k_temp_min_pulse_width_low) (1 + Δprocess k_process_min_pulse_width_low)
Minimum Clock Period	min_period + (1 + Δvoltage k_volt_min_period) (1 + Δtemp k_temp_min_period) (1 + Δprocess k_process_min_period)

Delay Scaling Factors Example

Example 11-14 shows delay scaling factors for a CMOS generic delay model.

Example 11-14 Setting k-Factors

```
library (example) {
 ...
   k_process_drive_fall     :  1.0;
   k_process_drive_rise     :  1.0;
   k_process_hold_rise      :  1.0;
   k_process_hold_fall      :  1.0;
   k_process_intrinsic_fall :  1.0;
   k_process_intrinsic_rise :  1.0;
   k_process_pin_cap        :  0.0;
   k_process_slope_fall     :  1.0;
   k_process_slope_rise     :  1.0;
   k_process_wire_cap       :  0.0;
   k_process_wire_res       :  1.0;
   k_temp_drive_fall        :  0.004;
   k_temp_drive_rise        :  0.004;
   k_temp_hold_rise         :   .0037;
   k_temp_hold_fall         :   .0037;
   k_temp_intrinsic_fall    :  0.004;
   k_temp_intrinsic_rise    :  0.004;
   k_temp_pin_cap           :  0.0;
   k_temp_slope_fall        :  0.0;
   k_temp_slope_rise        :  0.0;
   k_temp_wire_cap          :  0.0;
   k_temp_wire_res          :  0.0;
   k_volt_drive_fall        : -0.4;
   k_volt_drive_rise        : -0.4
   k_volt_hold_rise         : -0.26;
   k_volt_hold_fall         : -0.26;
   k_volt_intrinsic_fall    : -0.4;
   k_volt_intrinsic_rise    : -0.4;
   k_volt_pin_cap           :  0.0;
   k_volt_slope_fall        :  0.0;
   k_volt_slope_rise        :  0.0;
   k_volt_wire_cap          :  0.0;
   k_volt_wire_res          :  0.0;
   ...
}
```

In Example 11-14, only the intrinsic-rise, intrinsic-fall, rise-resistance, and fall-resistance delays are affected by a change in operating voltage. Setting the other voltage factors to 0.0 negates changes to them.

Scaling Factors for Individual Cells

The k-factors you define for a library as a whole do not produce accurate timing for certain cells, because not all cells in the same library scale uniformly: Pads scale differently from core cells, the timing of voltage-level pads varies with temperature, and some cells are designed to produce a constant delay and do not scale at all.

Other cells do not scale in a linear manner for process, voltage, or temperature. You can define a special set of scaling factors in a library-level group called `scaling_factors` and apply the
k-factors in this group to selected cells by using the `scaling_factors` attribute.

You can apply library-level k-factors to the majority of cells in your library and use this construct to provide additional accurate scaling factors for special cells. Example 11-15 uses the special scaling factors.

Example 11-15 Individual Scaling Factors

```
library (example) {
   k_volt_intrinsic_rise : 0.987 ;
   ...
   scaling_factors("IO_PAD_SCALING") {
      k_volt_intrinsic_rise : 0.846 ;
      ...
   }
   cell (INPAD_WITH_HYSTERESIS) {
      area : 0 ;
      scaling_factors : IO_PAD_SCALING ;
      ...
   }
    ...
}
```

Example 11-15 defines a scaling factor group called IO_PAD_SCALING that contains k-factors that are different from the library-level k-factors.

You can use any k-factors in a `scaling_factors` group. The `scaling_factors` attribute in the INPAD_WITH_HYSTERESIS cell is set to IO_PAD_SCALING, so all k-factors set in the IO_PAD_SCALING group are applied to the cell.

By default, all cells without a `scaling_factors` attribute continue to use the library-level k-factors.

You can model cells that do not scale at all, by creating a `scaling_factors` group in which all the k-factor values are set to 0.0.

Scaling Factors Associated With the Nonlinear Delay Model

As with the other delay models, the CMOS nonlinear delay model scaling factors scale the delay based on the variation in process, temperature, and voltage. The following scaling factors are specific to the CMOS nonlinear delay model:

- `k_process_cell_rise`
- `k_temp_cell_rise`
- `k_volt_cell_rise`
- `k_process_cell_fall`
- `k_temp_cell_fall`
- `k_volt_cell_fall`
- `k_process_rise_propagation`
- `k_temp_rise_propagation`
- `k_volt_rise_propagation`
- `k_process_fall_propagation`
- `k_temp_fall_propagation`
- `k_volt_fall_propagation`
- `k_process_rise_transition`
- `k_temp_rise_transition`
- `k_volt_rise_transition`
- `k_process_fall_transition`
- `k_temp_fall_transition`
- `k_volt_fall_transition`

The process, temperature, and voltage scaling factors for other CMOS delay models are not valid with the CMOS nonlinear timing model. Library Compiler checks these attributes as invalid and issues an error message. Only the scaling factors listed above are valid with the nonlinear timing model.

Design Compiler uses the same factors described for the CMOS generic delay model to scale pin and wire capacitance and wire resistance for the nonlinear delay model.

Define the scaling factors for setup, hold, recovery, removal, and skew in the nonlinear delay model as you do for the other delay models. For the nonlinear delay model, however, they apply to the values given in the associated constraint table.

For example, the timing constraint equation for setup rise is

```
rise_constraint * (1 + delta_voltage * k_volt_setup_rise)
* (1 + delta_temp * k_temp_setup_rise)
* (1 + delta_process * k_process_setup_rise)
```

where `rise_constraint` is a value in the `rise_constraint` table of the timing arc.

These are the scaling factors for the `setup_rising` timing constraint:

- `k_volt_setup_rise`
- `k_temp_setup_rise`
- `k_process_setup_rise`

The `scaling_factors` group is not affected by the change from linear to nonlinear delay model. The scaling factors for setup rise are valid in the `scaling_factors` group.

Specifying Nonlinear Delay Tables

These are two methods for specifying nonlinear delay tables.

Method 1

Use rise/fall propagation with rise/fall transition tables in which the total cell delay is the sum of propagation delay and transition delay. Use these attributes: `k_process_rise_propagation`, `k_temp_rise_propagation`, `k_volt_rise_propagation`, and so on.

Method 2

Use the cell-rise and cell-fall delay tables in which the transition delay is the index into the cell-delay table. Use these attributes: `k_process_cell_rise`, `k_temp_cell_rise`, `k_volt_cell_rise`, and so on.

You cannot use these k-factors interchangeably. For example, `cell_rise` delay is not scaled by `k_process_rise_propagation`, `k_temp_rise_propagation`, and so on; the `rise_propagation` delay is not scaled by `k_process_cell_rise`, and so on.

12

Verifying CMOS Libraries

Design Compiler uses information from technology libraries to drive its optimization strategies and to check that solutions adhere to the designer's specifications. The results of optimization are only as accurate as the technology library used.

To verify and validate CMOS technology and symbol libraries, you must understand the information in the following sections:

Library Verification

To ensure optimal results, you must make sure technology libraries have the following properties:

- Accuracy: The timing values and functional descriptions must be correct and consistent.
- Completeness: Describe all cells available from the ASIC vendor in the technology library. Use a complete delay calculation model.
- Consistency: All cells in the technology library must have an equivalent graphic representation in the symbol library.

If you already have the information needed to create a library file in another database, you can develop tools that automatically create a technology library from the information in your simulation libraries. Synopsys encourages the development of such tools. An automated system for creating technology libraries ensures

- Accuracy, by eliminating manual errors
- Completeness, by deriving information from the core database
- Consistent results, by using a single source of data

You can test a synthesis library by using the Synopsys VHDL Simulator. Library Compiler automatically generates the simulation models, test benches, test vectors, and a shell script. See *Library Compiler VHDL Libraries Reference Manual* for additional information.

To verify a library, you need

- The new library you want to verify
- A known, *golden* library
- A specially designed test circuit
- The simulator and timing analyzer from your development environment

Both the library to be verified and the known golden library are used in synthesis and optimization under very controlled circumstances. By comparing the results, you can determine whether the new library is accurate.

The procedures work well on combinational logic and simple cells. Libraries that contain complex, sequentially modeled cells can be tested, but the results may not be robust. For additional support, contact the Synopsys support center at (800) 245-8005.

Checking Library Consistency

The `compare_lib` command compares a technology library and a symbol library for consistency. These libraries must already be loaded in Design Compiler. Use the `list` command to display the libraries that are resident in Design Compiler.

This is the syntax for the `compare_lib` command:

```
compare_lib filename1 filename2
```

filename1, *filename2*

The names of the libraries to be compared. One must be a technology library and the other a symbol library. You can write these names in any order.

The `compare_lib` command performs two checks. First, it verifies that each cell in the technology library has a corresponding symbol definition in the symbol library. Second, it checks that the pin names of each cell in the technology library match the pin names defined for the cell's corresponding symbol.

In the following example, the `list` command verifies that the libraries you want to compare are already resident. Then the `compare_lib` command compares the cmos.db technology library with the cmos.sdb symbol library.

```
dc_shell> list -libraries

Library    File      Path
-------    ----      ----
cmos       cmos.db   /home/user/libraries
cmos       cmos.sdb  /home/user/libraries

dc_shell> compare_lib cmos.db cmos.sdb
```

Design Compiler reports discrepancies between the libraries.

Preparing a Test Circuit

You must prepare a test circuit design for each technology library you want to test. Most of the test and evaluation procedures described in this chapter assume that you are using a test circuit design with a specific topology. After you create this design, you can use it in each of the verification procedures.

The test design contains at least one of each cell in the technology library being tested. Connect each cell's input in a design to separate input pins in the module, and connect its output pins directly to a separate module output. Each clock pin should be driven from a separate pin. You need multiple clock pins if cells in the library require more than a single clock.

Example 12-1 shows a sample VHDL format netlist created for a library that contains the cells INV, NAND2, OR2, MUX21, AOI22, and DFF.

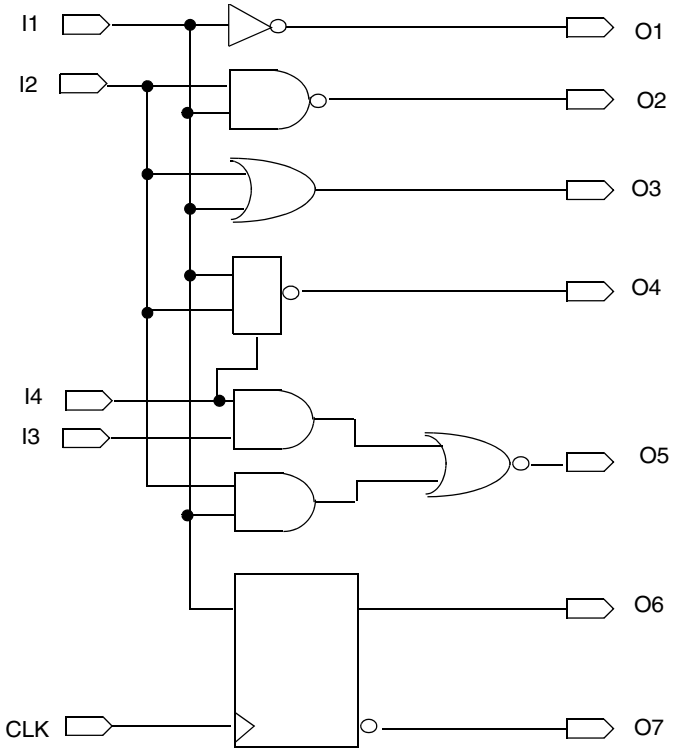
Example 12-1 Sample VHDL Format Netlist

```
entity TEST_CIRCUIT is
port( I1, I2, I3, I4, CLK : in BIT; O1, O2, O3, O4, O5, O6, O7 : out BIT);

end TEST_CIRCUIT;

architecture STRUCTURAL_VIEW of TEST_CIRCUIT is

   component DFF
      port( D, CP : in BIT; Q, QN : out BIT);
   end component;

   component AOI22
      port( A, B, C, D : in BIT; Z : out BIT);
   end component;

   component NAND2
      port( A, B : in BIT; Z : out BIT);
   end component;

   component MUX21
      port( A, B, S : in BIT; Z : out BIT);
   end component;

   component OR2
      port( A, B : in BIT; Z : out BIT);
   end component;
   component INV
      port( A : in BIT; Z : out BIT);
   end component;

begin
         O1_label : INV port map( A => I1, Z => O1);
         O2_label : NAND2 port map( A => I1, B => I2, Z => O2);
         O3_label : OR2 port map( A => I1, B => I2, Z => O3);
         O4_label : MUX21 port map( A => I1, B => I2, S => I3, Z => O4);
         O5_label : AOI22 port map( A => I1, B => I2, C => I3, D => I4, Z => O5);
```

```
        U6 : DFF port map( D => I1, CP => CLK, Q => O6, QN => O7);
end STRUCTURAL_VIEW;
```

Figure 12-1 shows the six-gate schematic for Example 12-1.

Figure 12-1 Test Circuit Netlist Network



Sharing the input pins makes creating input test vectors easier. In this example, it is practical for a single netlist to contain all cells of the technology library being verified. When you are testing a large library, however, you might want to use more than one netlist.

Verifying Functionality

To optimize logic networks properly, Design Compiler requires a database that describes the Boolean function of the elements in the target technology library. If these Boolean functions are incorrectly described in the library, Design Compiler can create invalid circuits. To prevent this, you must ensure that the Boolean `function` attributes given in the technology library match the functions described in the corresponding simulation library.

Preparing for Function Verification

To follow the function verification procedure, use an accurate simulator and a simulation library that is known to be functionally correct.

To prepare for testing, you need the following items:

- The Design Compiler software
- A technology library to test
- The test circuit netlist described in the previous section
- The simulator from the resident CAE environment
- The simulation library corresponding to the technology library you want to test
- The simulation test vectors
- The simulation output comparison program (optional)
- The test-mapping library created specifically for function verification

When you are testing the function descriptions in the technology library, you need to include only cells with `function` attributes in the test circuit netlist. Avoid including cells without function statements, unless you have simulation models. When the simulation comparison is performed, unmapped gates are compared against copies of themselves.

To verify the functionality of a technology library,

1. Create the test-mapping library
2. Map the test circuit netlist onto the test-mapping library

   Note:

   At this point in the process, your design is described only in terms of the cells from the test-mapping library. The design remains functionally identical to the design described in the test circuit netlist, but it is described solely in terms of the gate types in the test-mapping library.

3. Simulate both the original test circuit netlist and the mapped design
4. Compare the results of both simulations

Creating the Test-Mapping Library

The first new item to prepare is a test-mapping library. Use this as the target library for mapping the test circuit netlist you created. This library is as simple and small as possible, containing only two cells: an inverter and a 2-input NOR gate.

These two gates constitute the absolute minimal library used for mapping and optimization with Design Compiler. By using a simple library, you can ensure that the results of the mapping reflect the validity of the technology library being tested. Example 12-2 is a sample test-mapping library.

Example 12-2 Sample Test-Mapping Library

```
library(test_map_lib) {
   /* 2-input NOR gate */
   cell(NOR_2) {
      pin(A) {
         direction : input ;
      }
      pin(B) {
         direction : input ;
      }
      pin(X) {
         direction : output ;
         function : "(A+B)'" ;
         timing() {
            related_pin : "A B" ;
         }
      }
   }
   /* INVERTER gate */
   cell(INV) {
      pin(A) {
      direction : input ;
      }
      pin(X) {
         direction : output ;
         function : "A'" ;
         timing() {
            related_pin : "A" ;
      }
      }
   }
}
```

Include only the minimum information required for each cell in the test library file. Verify this file by visual inspection.

Note:

Design Compiler reads in a technology library containing any cell. Any library can be used to generate a VHDL library. The technology library must contain at least a 2-input NOR gate and an inverter. The minimum, however, is not sufficient for all designs. The minimum sufficient technology library to handle a design consists of the set of cells described in "Verifying Technology and VHDL Libraries Together" on page 12-10.

The following commands prepare the library for the function verification test and compile and save it:

```
lc_shell> read_lib test_map_lib.lib
lc_shell> write_lib test_map_lib
```

The result is a technology library named test_map_lib.db.

Mapping the Test Circuit Netlist

The next step is to map the test circuit netlist of library cells onto the two cells described in test_map_lib.db. To do this, follow these steps:

1. Set the following variables in dc_shell:

```
dc_shell> link_library = {your_technology_library.db,test_map_lib.db }
dc_shell> target_library = test_map_lib.db
```

2. Read in the test circuit netlist. This example uses the VHDL format, but you can use any supported netlist format.

3. Map the netlist onto the test library with the following commands:

```
dc_shell> read -f vhdl TEST_CIRCUIT.NET
dc_shell> translate
```

Note:

At this point in the process, you have a design containing only the cells from the test-mapping library. The design is functionally identical to the design described in the test circuit netlist, and the design is described solely in terms of the two gate types in the test-mapping library.

4. Export the design to the simulation environment:

```
dc_shell> write -f vhdl TEST_CIRCUIT -o TEST_MAPPED.NET
```

This command saves the mapped design in a file called TEST_MAPPED.NET.

Creating the Simulation Test Vectors

You now have two netlists:

- TEST_CIRCUIT.NET*,* which uses the technology library you are verifying
- TEST_MAPPED.NET, which is the result of mapping your test circuit netlist onto the two cells that you know are accurately modeled

The next step is to simulate both of the netlists and compare the reported functionality. To do this,

1. Create a set of simulation input vectors to use in both simulations.
2. Generate the pattern file by using the language of your simulation environment. You can simulate all possible input patterns in cases in which the number of input pins is small.
3. Separately test those cells having more than 20 input pins. Use the same methodology, coupled with a targeted function test-vector set.

Simulating the Mapped and Original Test Netlists

Run the simulator on both netlists. For convenience, you can create a super netlist that contains both the mapped and the original netlists. On this netlist, each pair of corresponding output pins connected with an XOR gate identifies all situations where two corresponding output pins are not identical.

The process is taken one step further by connecting the output pins of the exclusive-OR gates with OR gates. With this connection, there is a single super module output to monitor. The library is functionally correct if the output of this OR gate is not high after all events from each input vector have settled.

Identifying Function Description Errors

After you have the results of both simulations, compare the results and search for discrepancies. Many vendors provide a tool for comparing the results of two simulation output pins. If you have this capability, run that tool.

If you do not have a simulation-output comparison program, you must perform the discrepancy check manually. Every discrepancy in the simulations is a functional inconsistency between your simulation library and your technology library.

When no discrepancies exist, the technology and simulation libraries are functionally consistent.

Combinational cells with the `dont_touch` attribute are not verified by this procedure; like black boxes, these cells are not mapped.

Verifying Technology and VHDL Libraries Together

You can verify the functionality of your technology library and the corresponding VHDL library at the same time. The steps are the same as those listed in "Preparing for Function Verification" on page 12-6. Variations on this basic procedure are needed to test the two libraries concurrently.

Creating the Test-Mapping Library

You need the following set of cells in the test-mapping library to support all technology libraries for this procedure:

- An inverter.
- A 2-input NOR gate.
- A three-state buffer, if used in the library.
- A D flip-flop with preset, clear, and two complementary output pins—when both control signals are active, the output pins are both 0.
- A D flip-flop with preset, clear, and two complementary output pins—when both control signals are active, the output pins are both 1.

When you create the DB image of the test-mapping library, create a VHDL library as well and analyze it:

```
lc_shell> read_lib test_map_lib.lib
lc_shell> write_lib test_map_lib
lc_shell> write_lib -f vhdl test_map_lib
```

Mapping the Test Circuit Netlist

To map the test circuit netlist onto your test map library, use these commands:

```
dc_shell> vhdlout_write_components = FALSE
dc_shell> vhdlout_bit_type = "MVL7";
dc_shell> vhdlout_bit_vector_type = "MVL7_VECTOR";
dc_shell> vhdlout_use_packages = {synopsys.types.all
          technology.COMPONENTS.all}
dc_shell> link_path={technology.db, test_map_lib.db}
dc_shell> target_library={test_map_lib.db}
dc_shell> symbol_library={generic.sdb}
dc_shell> read -f tdl test_circuit.tdl
dc_shell> write -f vhdl test_circuit -output net_lib.vhd
dc_shell> max_area 0
```

```
dc_shell> set_structure false -timing false
dc_shell> set_flatten -phase true -effort low
dc_shell> compile
dc_shell> write -f vhdl test_circuit -output
          net_test_map.vhd
```

Completing the Verification

After you create the simulation test vectors (see “Creating the Simulation Test Vectors” on page 12-9), run VHDL simulation on both net_lib.vhd and net_test_map.vhd and compare the results. Simulations differ in unknown behavior; see “Ambiguity Resolution” in the *Library Compiler VHDL Libraries Reference Manual* for additional information.

Verifying Timing Parameters

You need to verify timing parameters for two reasons:

1. When you work with several different tools in a single design environment, you must ensure that all the tools use consistent technology information.
2. Design Compiler must have accurate timing information to use when it synthesizes and optimizes your designs. Inaccurate timing information can cause Design Compiler to create less-than-optimal designs. Bad timing information can even cause Design Compiler to create new circuits that represent a degradation in performance from the original.

Both of these requirements are met by a determination of how closely the results from the Design Compiler timing calculations track the results generated by the timing simulator.

Testing Parameters

The following sections describe the method for verifying the timing parameters in a technology library, describing important parameters to test, and devising tests for them:

- General Testing Methodology
- Debugging With the `report_delay_calculation` Command
- Testing Basic Cell-Timing Parameters
- Testing the Interconnect Models
- Testing Environmental Scaling

To verify that the parameter value entry is correct, use the `report_lib -timing` command, described in Chapter 6, “Advanced Composite Current Source Modeling.”

The divide-and-conquer approach to parameter testing is important in determining sources of discrepancies and in differentiating algorithmic differences in the tools from library parameter errors.

To verify the timing parameters, you need

- The test circuit netlist
- The Design Compiler software
- A timing simulator or analyzer

This is the procedure:

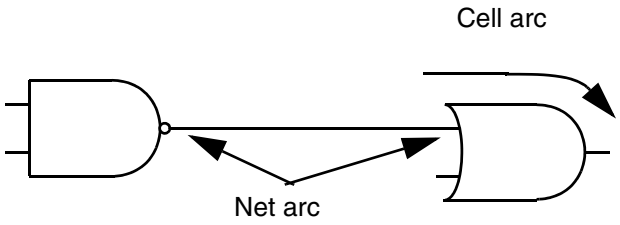
1. Run a Design Compiler timing report on the test circuit netlist.
2. Run a simulation timing analyzer on the test circuit netlist.
3. Compare the results.

General Testing Methodology

This section describes a general methodology for testing timing arcs associated with nets and cells with the `report_delay_calculation` command.

Use the `report_delay_calculation` command to display detailed timing calculation information about both cell arcs and net arcs, as shown in Figure 12-2. This information is especially helpful for debugging or verifying timing data.

Figure 12-2 Net and Cell Arcs



Note:

If you want the `report_delay_calculation` command to display confidential library data, you must load the library in source format. You can do this with the `read_lib` command.

This is the syntax of the `report_delay_calculation` command:

```
report_delay_calculation -from pin_name -to pin_name
```

-from *pin_name*

Defines the starting point of a timing arc within your design.

-to *pin_name*

Defines the ending point of a timing arc within your design.

For a *cell* timing arc, use the input and output pins of a common leaf cell that has a timing arc defined between them in the library.

For a *net* timing arc, use a driver and a load on a common net. Port names are allowed in place of a pin name for net arcs.

If there is a cell timing arc between pins, the details of the *cell* arc delay calculation will be displayed in the timing arc report. If there are pins that are part of the same net, the details of the *net* arc delay calculation will be displayed in the timing arc report.

The following conditions produce an error with no delay information displayed:

- A technology library was not loaded in source format.
- There are no cell or net delay arcs between the specified pins.
- You specified an undefined pin.
- The specified pin is associated with a nonleaf cell.
- You specified more than one pin for the `-from` or `-to` option.
- You specified an arc type between pins that is not supported.

The format of the timing report output varies, depending on the type of delay model and the interconnect delay tree type (for net delay arcs). If delay values have been back-annotated for a specified arc, the annotated delay, rather than calculated delay, is given.

Example 12-3 shows one type of timing arc report that is generated for the output of a cell arc in a generic CMOS library. The timing reports for different types of libraries, such as nonlinear, look similar.

Example 12-3 Cell Arc Output Report

```
From :                                U1/A
To :                                  U1/X
arc type:                             cell
arc sense:                            inverting
Input net transition times: Dt_rise = 0, Dt_fall = 0

Rise Delay computation:
rise_intrinsic                        0 +
rise_slope * Dt_fall                  0 * 0 +
rise_resistance * (pin_cap + wire_cap) / driver_count
0.2 * (0 + 0.53) / 1
```

```
rise_transition_delay :             0.106
-----------------------------------------
Total                               0.106
Fall Delay computation:
fall_intrinsic                      0 +
fall_slope * Dt_rise                0 * 0 +
fall_resistance * (pin_cap + wire_cap) / driver_count
0.15 * (0 + 0.53) / 1
fall_transition_delay :             0.0795
-----------------------------------------
Total                               0.0795
```

Debugging With the report_delay_calculation Command

You can use the `report_delay_calculation` command to help debug delay calculations along a critical path. To do this, first locate a critical path in a design with the `report_timing -path full -input_pins` command to get a listing similar to this one:

```
  Point                                    Incr       Path
  ------------------------------------------------------------
  input external delay                     0.00       0.00 f
  i2 (in)                                  0.00       0.00 f
  cell1/i2 (lower1)                        0.00       0.00 f
  cell1/C/B (AN2)                          0.00       0.00 f
  cell1/C/Z (AN2)                          0.82       0.82 f
  cell1/o1 (lower1)                        0.00       0.82 f
  cell2/i1 (lower2)                        0.00       0.82 f
  cell2/C/A (IV)                           0.00       0.82 f
  cell2/C/Z (IV)                           0.38       1.20 r
  cell2/o1 (lower2)                        0.00       1.20 r
  o1 (out)                                 0.00       1.20 r
  data arrival time                                   1.20
```

The `-input_pins` option causes the input pins to be listed in the path, in addition to the output pins typically seen. Next, you can use the `report_delay_calculation` command to print the details of a cell delay arc, by giving it the input and output pin of a leaf cell along the path. This is an example:

```
report_delay_calculation -from cell1/C/B -to cell1/C/Z
```

To print the details of a net delay arc, use a driver and a load pin on the same net along the path. The pins must be associated with leaf cells. This is an example that is valid, because both cells are leaf cells:

```
report_delay_calculation -from cell1/C/Z -to cell2/C/A
```

The following command is invalid, because there is no net delay arc associated with the from pin, because it is not on a leaf cell:

```
report_delay_calculation -from cell2/i1 -to cell2/C/A
```

Operating conditions and wire loads are considered when you generate a timing arc report, but timing ranges are not. This is because timing ranges typically apply to an entire path, as opposed to a single timing arc.

Testing Basic Cell-Timing Parameters

Test all timing parameters and related information used to describe cells in a technology library. These parameters include

- Intrinsic delay
- Resistance
- Capacitance
- Input slope
- Setup
- Hold
- Rising and falling edge
- Area
- Ranges for piecewise linear model

Intrinsic Delays

To test intrinsic delays that are unloaded-gate delay values, run a Design Compiler timing report on the test design described earlier. Use the following commands:

```
dc_shell> read -f vhdl TEST_CIRCUIT.NET
dc_shell> report_timing > test.intrinsic
```

Then run a timing simulation or analysis on the test design.

Compare the results of these tests. If the output capabilities of the simulator allow it, write a small program to compare the two output values. The development of such a program is strongly encouraged.

If you have a `default_wire_load` or `default_operating_conditions` attribute defined in your library, use the `set_wire_load` or `set_operating_conditions` command to set the wire load and operating conditions to null, as in the following example:

```
dc_shell> set_wire_load ""
dc_shell> set_operating_conditions ""
```

Resistance (Generic Delay Model)

Test the resistance parameters described in the technology library for each cell under at least two loading conditions. These are the resistance parameters used in the standard timing delay model:

- `rise_resistance`
- `fall_resistance`

Using the same design as for intrinsic delay, vary the load at each output pin with the Design Compiler `set_load` command for each output pin, as illustrated in Example 12-4.

Example 12-4 Testing Generic-Delay Resistance Parameters

```
dc_shell> read -f vhdl TEST_CIRCUIT.NET
dc_shell> set_load 2 all_outputs()
dc_shell> report_timing > test.resis_load_2
dc_shell> set_load 4 all_outputs()
dc_shell> report_timing > test.resis_load_4
```

Run a simulation by using the same two loading conditions if your simulator lets you set loading at output pins. If the simulator cannot set alternative load values at module output pins, you must generate this effect by adding more load gates to each module output.

To add more load gates to each module output:

1. Add one inverter.
2. Add two inverters to each output pin by connecting the input pins and connecting the output pins.

Resistance (Piecewise Linear Delay Model)

Test the resistance parameters for each cell under at least two loading conditions. These are the resistance parameters used in the piecewise linear timing delay model:

- `rise_pin_resistance`
- `fall_pin_resistance`
- `rise_delay_intercept`
- `fall_delay_intercept`

Use the test design described earlier in this chapter. Vary the wire length and load at each output pin. To vary the wire length,

1. Connect the output of the test circuit to a load circuit. Use a small gate such as an inverter for the load circuit.
2. Make sure you check each range defined by the `piece_define` statement. For example, pieces 0, 1, and 2 are tested in the example below. The wire load model defines the wire length added for each extra loading gate. This is an example:

```
dc_shell> read -f vhdl TEST_CIRCUIT.NET
dc_shell> set_wire_load default
dc_shell> report_timing > test.resis_load_piece_0
dc_shell> read -f vhdl TEST_CIRCUIT_piece_1.NET
dc_shell> set_wire_load default
dc_shell> report_timing > test.resis_load_piece_1
dc_shell> read -f vhdl TEST_CIRCUIT_piece_2.NET
dc_shell> set_wire_load default
dc_shell> report_timing > test.resis_load_piece_2
```

3. Run a simulation on each of the different netlists.
4. Compare the results of the simulation with the timing reports generated by Design Compiler.

Capacitance

Testing capacitance output pin values, which are often 0 in technology libraries, is implicit in the verification of both intrinsic delay and resistance. You need to devise additional tests only for input pin capacitance values. Use the same design as for intrinsic delay, but vary the drive value at each input pin with the Design Compiler `set_drive` command.

Example

```
dc_shell> read -f vhdl TEST_CIRCUIT.NET
dc_shell> set_drive 2 all_inputs()
dc_shell> report_timing > test.incap_drive_2
dc_shell> set_drive 4 all_inputs()
dc_shell> report_timing > test.incap_drive_4
```

If your simulator allows you to vary the effective drive at design input pins, input pin capacitance validation is not difficult. If the simulator cannot vary the effective drive at the input pins, you must add inverters or buffers at every input pin to generate a drive at the pin.

The pin's delay value, in turn, depends on the input pin capacitance under test. For this test, you need only a single simulation with a nonzero drive value at the input pins.

Input Slope

Use the same basic design used for function verification to begin testing each gate's sensitivity to changes in rise time and input fall time. The test for the input sensitivity to step functions is implicit in the tests for intrinsic delay and resistance parameters.

To test nonzero rise and fall times, use the Design Compiler `set_load` and `set_drive -rise` or `set_drive -fall` commands to change the input rise and fall time, as in the following example:

Example

```
dc_shell> read -f vhdl TEST_CIRCUIT.NET
dc_shell> set_load 2 all_inputs()
dc_shell> set_drive 2 all_inputs()
dc_shell> report_timing > test.slope_rc_4
dc_shell> set_load 2 all_inputs()
dc_shell> set_drive 4 all_inputs()
dc_shell> report_timing > test.slope_rc_8
```

Setup

Setup times are treated as constraint equation parameters by Design Compiler. To check the parameters of constraint equations, devise dynamic tests that check both the violating condition and the complying condition at the boundary of the violation.

Implement the test in Design Compiler by using a combination of the `set_input_delay` command and the `max_period` constraint to set up the dynamic conditions for a setup check.

For example, assume you have a rising-edge-triggered D flip-flop with a setup time requirement of 4. For simplicity, assume that only integer time values are assigned. To test the boundary violation condition, use the following commands:

```
dc_shell> set_input_delay -fall 1 d
Performing set_input_delay on port 'd'.
1
dc_shell> create_clock 3 CLK
Performing create_clock on clock 'CLK'
1
dc_shell> report_constraint -verbose
Information: Updating design information... (UID-85)

***********************************
Report : constraint
    -verbose
Design : DESIGN
Version : v3.0
Date    : Tues Jan 14 14:05:20 2002
***********************************
1. max_period 3.000000 { CLK }

Worst:                  q_reg/D
Max Data Path             1.00
+ Library Setup           4.00
- Min Active Clock        0.00
- (Clock Period *         3.00
```

```
    Period Multiplier)   1.00
+ Minus Clock Skew       0.00
-----------------------------
Delta Cost              2.00 (VIOLATED)

The minimum clock period is 5.00 for signal 'CLK' (on'q_reg/
D').
1
```

In the simulator, you can use input vectors that generate the `set_input_delay` and `max_period` signal behavior for the setup check. If setup and hold checking is provided by the simulator, you can use the output listing to verify that both systems have found that the signal complies with, or violates, the timing requirements of the cell. In the simulator, the vector specification might look something like this:

```
time 0 : D_PORT = 0, CLOCK_PORT = 0; /*init signals */
time 100: D_PORT = 1;     /*apply data signal change */
time 103: CLOCK_PORT = 1; /*apply clk at period boundary of
3*/
```

The simulator output will also show a violation.

Validation of the boundary condition for compliance looks like this:

```
dc_shell> set_input_delay -fall 1 D
Performing set_input_delay on port 'D'.
1
dc_shell> create_clock 5 CLK
Performing create_clock on clock 'CLK'.
1
dc_shell> report_constraint -verbose
Information: Updating design information... (UID-85)

***********************************
Report : constraint
    -verbose
Design : flop
Version : v3.0
Date    : Tues Jan 14 14:05:20 2002
***********************************

1. max_period 5.000000 { CLK }

Worst:                 q_reg/D
Max Data Path            1.00
+ Library Setup          4.00
- Min Active Clock       0.00
- (Clock Period *        5.00
   Period Multiplier)    1.00
+ Minus Clock Skew       0.00
-----------------------------
```

```
Delta Cost                0.00 (MET)

The minimum clock period is 5.00 for signal 'CLK' (on 'q_reg/
d')
1
```

In the simulator, the vector specification looks like this:

```
time 0 : D_PORT = 0, CLOCK_PORT = 0;   /*init signals */
time 100: D_PORT = 1;     /*apply data signal change */
time 104: CLOCK_PORT = 1; /*apply clk at period boundary of
4*/
```

The simulator output will show compliance. In some simulators, you may have to run tools that postprocess the simulation results to determine if they adhere to timing rules.

Hold

Hold times are treated as constraint equation parameters by Design Compiler. To check the values of constraint equation parameters in the technology library, devise dynamic tests that check both the violating condition and the complying condition at the boundary of the violation.

For hold time verification, use the same approach you did for setup time verification. Use the same example as described for setup time validation, but assign a hold time constraint of 2 and test the boundary violation condition with the following commands:

```
dc_shell> set_input_delay 1 D_PORT
dc_shell> create_clock 4 CLOCK_PORT
dc_shell> report_constraint -verbose

2. Clocks_at_hold 4.000000 { CLOCK_PORT }

Worst:                  U1/D
Max Active Clock        0.00
+ Library Hold          2.00
- Min Data Path         1.00
-----------------------------
Delta Cost              1.00 (VIOLATED)
```

In the simulator, the equivalent vector specification looks like this:

```
time 0 : D_PORT = 0, CLOCK_PORT = 0; /* init signals */
time 100: CLOCK_PORT = 1; /* apply clk */
time 101: D_PORT = 1;      /* apply data signal change */
```

The simulator output will also show a violation.

Use the following commands to validate the boundary condition for compliance in Design Compiler:

```
dc_shell> set_input_delay 2 D_PORT
dc_shell> create_clock 4 CLOCK_POR;
dc_shell> report_constraint -verbose

2. Clocks_at_hold 4.000000 { CLOCK_PORT }

Worst:                  U1/D
Max Active Clock        0.00
+ Library Hold          2.00
- Min Data Path         2.00
-----------------------------
Delta Cost              0.00 (MET)
```

In the simulator, the equivalent vector specification looks like this:

```
time 0 : D_PORT = 0, CLOCK_PORT = 0;   /* init signals */
time 100: CLOCK_PORT = 1; /* apply clk */
time 102: D_PORT = 1;   /* apply data signal change */
```

The simulator output will also show compliance. In some simulators, you might have to run tools that postprocess the simulation results to determine if they adhere to timing rules.

Rising and Falling Edges

You can use the `timing_type` command to validate pins that have timing arcs, by using a combination of the Design Compiler reporting features and the logic simulator of the target environment. For example, use the Design Compiler `set_input_delay -rise` and `set_input_delay -fall` commands, so that events occur at the input pins at the same time as they occur in the simulation. Then, in the simulator, design a two-vector input set that applies both the active edge and the inactive edge to the edge-sensitive input pin.

If you specify the event trigger correctly in the technology library, only the trigger event propagates through the cell in the Design Compiler timing report. The nontriggering event is masked. These results will be reflected in the simulator's output.

Area

You can use the test circuit netlist to verify the area parameters of the technology library you want to test. The `report -area` command in Design Compiler reports the total area of the design. To compare this value with the actual area by manual inspection, use the following command:

```
dc_shell> report_area
```

Ranges for Piecewise Linear Model

Using the `piece_define` attribute statement, you can define the ranges for pieces in the piecewise linear timing delay model at the library level. To verify the ranges, use the `report_lib` command:

```
lc_shell> report_lib TEST_CIRCUIT
```

Look for the following section in the library report:

```
Piecewise Linear Delay Model:

 Piece             Length
 ----------------------------------------
   1         0.0 <= length < 50.0
   2        50.0 <= length < 100.0
   3       100.0 <= length
```

The piecewise linear equations in this library use three pieces with breakpoints at 50 and 100 mils.

When the `piece_type` is capacitance, the report displays the following:

```
Piecewise Linear Delay Model:

Piece               Pin Capacitance
-----------------------------------------------
  1         0.0 <= capacitance <     1.5
  2         1.5 <= capacitance <     4.3
  3         4.3 <= capacitance
```

Testing the Interconnect Models

You must test each `wire_load` group in the technology library. You can use a simple inverter tree design to test the wire load specifications. Set up the test design to test a range of fanout counts.

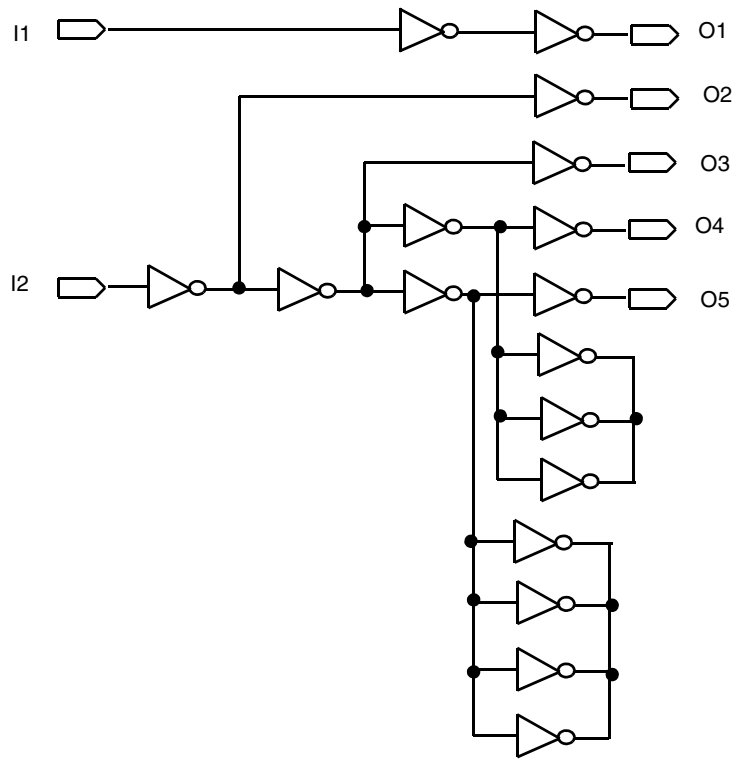
For example, design a circuit in which one input inverter supplies the input signals for two inverters, one of which supplies the input signal for three inverters, and the other supplies the input signal for four, and so on, out to the highest fanout value defined in the `wire_load` group.

Example 12-5 shows a possible test circuit in netlist format, and Figure 12-3 shows the same test circuit as a schematic.

Example 12-5 Fanout Test Netlist

```
entity FANOUT_TEST is
   port( I1, I2 : in BIT;  O1, O2, O3, O4, O5 : out BIT);

end FANOUT_TEST;

architecture STRUCTURAL_VIEW of FANOUT_TEST is
   component INVH
      port( A : in BIT;  Z : out BIT);
   end component;

begin

   O1 <= I1;
   O2 <= I2;
   O4 <= I2;
   O5 <= I2;

   U1 : INVH  port map( A => I2, Z => O3);

end STRUCTURAL_VIEW;
```

Figure 12-3 Fanout Test Circuit Schematic



In this example, the library under test has two wire load models defined: SMALL_CHIP_WIRE and BIG_CHIP_WIRE. Assume that the test circuit inverter tree is also in a VHDL netlist named FANOUT_TEST.NET.

First, collect Design Compiler timing reports on the test circuit netlist, by using both of the defined wire load models:

```
dc_shell> read -f vhdl FANOUT_TEST.NET
dc_shell> set_wire_load SMALL_CHIP_WIRE
dc_shell> report_timing > fanout_test.small_chip
dc_shell> set_wire_load BIG_CHIP_WIRE
dc_shell> report_timing > fanout_test.big_chip
```

Then run the timing simulator on the fanout test circuit netlist and compare the results with those generated by Design Compiler.

Testing Environmental Scaling

Use a design with a single inverter to test the environmental scaling performed under all defined conditions. Generate a Design Compiler timing report for each `operating_conditions` group in the library.

In the following example, the three operating condition groups, BCMIL, NOM, and WCMIL, are tested by use of the test circuit netlist SINGLE_INV.NET, which contains one inverter:

```
dc_shell> read -f vhdl SINGLE_INV.NET
dc_shell> set_operating_conditions BCMIL
dc_shell> report_timing > single_inv.bcmil
dc_shell> set_operating_conditions NOM
dc_shell> report_timing > single_inv.nom
dc_shell> set_operating_conditions WCMIL
dc_shell> report_timing > single_inv.wcmil
```

Compare the results with those from simple 2-input vector simulations, where the simulation delay values were computed with the values from the corresponding delay environments. This test also performs an implicit check on the environmental scaling factors in the technology library.

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Z