Liberty User Guide, Vol. 1

Version 2008.09

1. Sample Library Description

2. Building a Technology Library

3. Building Environments

4. Defining Core Cells

9. Modeling Power and Electromigration

10. Timing Arcs

12. Advanced Composite Current Source Modeling

13. Composite Current Source Signal Integrity Modeling

14. Composite Current Source Power Modeling

15. Modeling Noise

1. Sample Library Description

This chapter familiarizes you with the basic format and syntax of a library description. It starts with an example that shows the general syntax of the library. It also discusses the syntax used to describe a library and how the library description is structured. These topics are covered in the following sections:

- General Syntax
- Statements

For a complete list of all the groups and attributes in a technology library, see Appendix A.

1.1 General Syntax

Example 1-1 shows the general syntax of a library description. The first statement names the library. The statements that follow are library-level attributes that apply to the library as a whole. These statements define library features such as technology type, and definitions and defaults that apply to the library in general. Every cell in the library has a separate cell description. For more information on cell descriptions, see Chapter , “.”

Example 1-1 General Syntax of the Library Description

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

```
  piece_define ("list") ;
  define_cell_area(area_name,resource_type);

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

1.2 Statements

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

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

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

1.2.1 Group Statements

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

Syntax

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

name

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

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

Example 1-2 defines pin group A.

Example 1-2 Group Statement Specification

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

1.2.2 Attribute Statements

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

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

Simple Attributes

This is the syntax of a simple attribute:

attribute_name : *attribute_value* ;

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

Example 1-3 adds a direction attribute to the pin group shown in Example 1-2.

Example 1-3 Simple Attribute Specification

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

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

attribute_name : "*attribute_value*" ;

Example 1-4 adds the function X + Y to the pin example.

Example 1-4 Defining the Function of a Pin

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

Complex Attributes

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

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

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

line (1, 2, 6, 8);

1.2.3 Define Statements

You can create new simple attributes with the define statement.

Syntax

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

attribute_name

The name of the new attribute you are creating.

group_name

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

attribute_type

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

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

define (bork, pin, string) ;

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

bork : "nimo"

1.3 Reducing Library File Size

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

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

Syntax

include_file(*file_name*$_{id}$) ;

Example

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

where memory_file contains the `memory` group statements.

Limitations

The `include_file` attribute has these requirements:

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

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

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

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

  where `source_file` is the following:

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

2. Building a Technology Library

The library description identifies the characteristics of a technology library and the cells it contains. Creating a library description for a CMOS technology library involves the following concepts and tasks explained in this chapter:

- Creating Library Groups
- Using General Library Attributes
- Delay and Slew Attributes
- Defining Units
- Using Piecewise Linear Attributes

2.1 Creating Library Groups

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

2.1.1 library Group

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

Example

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

2.2 Using General Library Attributes

These attributes apply generally to the technology library:

- technology
- delay_model
- bus_naming_style
- routing_layers

2.2.1 technology Attribute

This attribute identifies the following technology tools used in the library:

- CMOS (default)
- FPGA

The `technology` attribute must be the first attribute defined and is placed at the top of the listing. If no `technology` attribute is entered, the default is assumed.

Syntax

technology (*name*$_{enum}$) ;

name

Valid values are CMOS or FPGA. If you specify FPGA, you must also specify the `fpga_technology` attribute at the library level. The default is CMOS.

Example

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

2.2.2 delay_model Attribute

This attribute indicates which delay model to use in the delay calculations. The six models are

- generic_cmos (default)
- table_lookup (nonlinear delay model)
- piecewise_cmos (optional)
- dcm (Delay Calculation Module)
- polynomial

The `delay_model` attribute must follow the technology attribute; or, if a `technology` attribute is not present, the `delay_model` attribute must be the first attribute in the library. The default value for the `delay_model` attribute, when it is the first attribute in the library, is generic_cmos.

Example

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

2.2.3 bus_naming_style Attribute

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

Example

```
bus_naming_style : "Bus%sPin%d";
```

2.2.4 routing_layers Attribute

This attribute declares the routing layers available for place and route for the library. The declaration is a string that represents the symbolic name used later in a library to describe routability information associated with each layer.

The routing_layers attribute must be defined in the library before other routability information in a cell. Otherwise, cell routability information in the library is considered an error. Each different library can have only one `routing_layers` attribute.

Syntax

routing_layers("*routing_layer_1_name*",...,
"*routing_layer_n_name*") ;

Example

```
routing_layers ("routing_layer_one, routing_layer_two");
```

You can display routing_layers information in a library with the `report_lib` command. The report looks similar to the following:

```
Porosity information:
Routing_layers: "metal2" "metal3"
default_min_porosity: 15.0
```

If there is no porosity information in the library, `report_lib` displays the following line:

```
No porosity information specified.
```

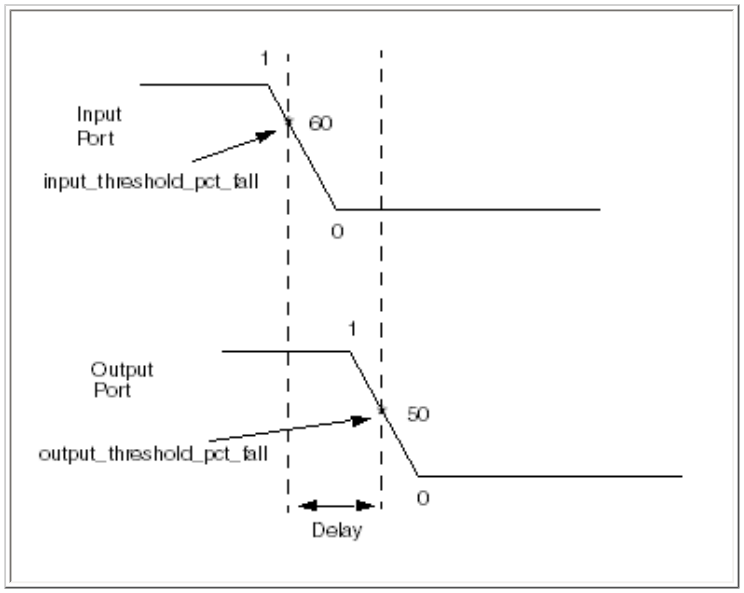
2.3 Delay and Slew Attributes

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

Delay is the time it takes for the output signal voltage, which is falling from 1 to 0, to fall to the threshold point set with the `output_threshold_pct_fall` attribute after the input signal voltage, which is falling from 1 to 0, has fallen to the threshold point set with the `input_threshold_pct_fall` attribute (see Figure 2-1).

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

Figure 2-1 Delay Modeling for Falling Signal



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

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

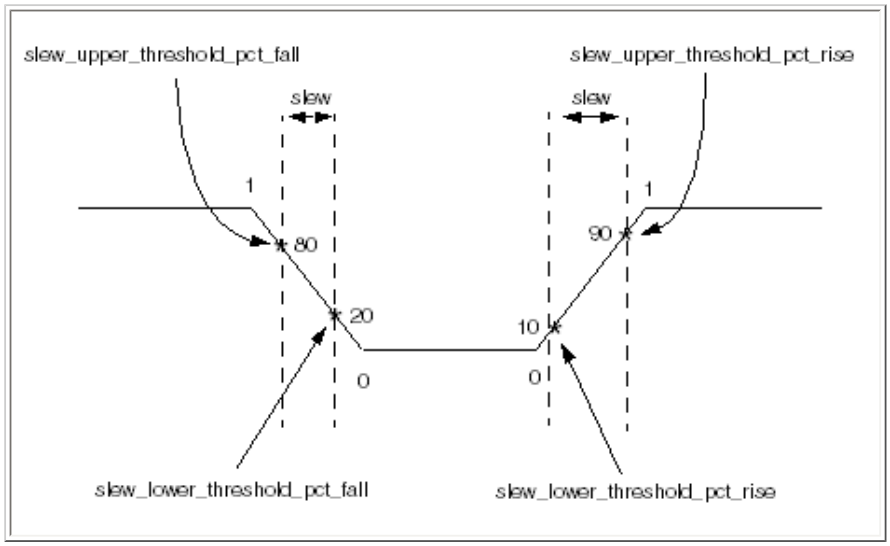
- slew_lower_threshold_pct_fall
- slew_upper_threshold_pct_fall

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

- slew_lower_threshold_pct_rise
- slew_upper_threshold_pct_rise

Figure 2-2 shows an example of slew modeling.

Figure 2-2 Slew Modeling



2.3.1 input_threshold_pct_fall Simple Attribute

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

Syntax

input_threshold_pct_fall : *trip_point*$_{value}$;

trip_point

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

Example

```
input_threshold_pct_fall : 60.0 ;
```

2.3.2 input_threshold_pct_rise Simple Attribute

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

Syntax

input_threshold_pct_rise : *trip_point*$_{value}$;

trip_point

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

Example

```
input_threshold_pct_rise : 40.0 ;
```

2.3.3 output_threshold_pct_fall Simple Attribute

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

Syntax

output_threshold_pct_fall : *trip_point*$_{value}$;

trip_point

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

Example

```
output_threshold_pct_fall : 40.0 ;
```

2.3.4 output_threshold_pct_rise Simple Attribute

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

Syntax

output_threshold_pct_rise : *trip_point*$_{value}$;

trip_point

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

Example

```
output_threshold_pct_rise : 40.0 ;
```

2.3.5 slew_derate_from_library Simple Attribute

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

Syntax

slew_derate_from_library : *derate*$_{value}$;

derate

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

Example

```
slew_derate_from_library : 0.5 ;
```

2.3.6 slew_lower_threshold_pct_fall Simple Attribute

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

Syntax

slew_lower_threshold_pct_fall : *trip_point value* ;

trip_point

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

Example

```
slew_lower_threshold_pct_fall : 30.0 ;
```

2.3.7 slew_lower_threshold_pct_rise Simple Attribute

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

Syntax

slew_lower_threshold_pct_rise : $trip\_point_{value}$;

trip_point

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

Example

```
slew_lower_threshold_pct_rise : 30.0 ;
```

2.3.8 slew_upper_threshold_pct_fall Simple Attribute

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

Syntax

slew_upper_threshold_pct_fall : $trip\_point_{value}$;

trip_point

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

Example

```
slew_upper_threshold_pct_fall : 70.0 ;
```

2.3.9 slew_upper_threshold_pct_rise Simple Attribute

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

Syntax

slew_upper_threshold_pct_rise : *trip_point*$_{value}$;

trip_point

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

Example

```
slew_upper_threshold_pct_rise : 70.0 ;
```

2.4 Defining Units

Use these six library-level attributes to define units:

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

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

2.4.1 time_unit Attribute

The VHDL library generator uses this attribute to identify the physical time unit used in the generated library.

Syntax

time_unit : *unit* ;

unit

Valid values are 1ps, 10ps, 100ps, and 1ns. The `time_unit` attribute default value is 1ns.

Example

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

2.4.2 voltage_unit Attribute

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

Syntax

voltage_unit : *unit* ;

unit

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

Example

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

2.4.3 current_unit Attribute

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

Syntax

current_unit : $value_{enum}$;

value

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

Example

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

2.4.4 pulling_resistance_unit Attribute

Use this attribute to define pulling resistance values for pull-up and pull-down devices.

Syntax

pulling_resistance_unit : "*unit*" ;

unit

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

Example

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

2.4.5 capacitive_load_unit Attribute

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

Syntax

capacitive_load_unit ($value_{float}$,$unit_{enum}$) ;

value

A floating-point number.

unit

Valid values are ff and pf.

The first line in the following example sets the capacitive load unit to 1 pf. The second line represents capacitance in terms of the standard unit load of an inverter.

Example

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

2.4.6 leakage_power_unit Attribute

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

Syntax

leakage_power_unit : $value_{enum}$;

value

Valid values are 1mW, 100uW (for 100mW), 10uW (for 10mW), 1uW (for 1mW), 100nW, 10nW, 1nW, 100pW, 10pW, and 1pW.

Example

```
leakage_power_unit : 100uW;
```

2.5 Using Piecewise Linear Attributes

Use these library-level attributes for libraries with piecewise linear delay models:

- piece_type
- piece_define

2.5.1 piece_type Attribute

This attribute lets you use capacitance to define the piecewise linear model.

Syntax

piece_type : $value_{enum}$;

value

Valid values are `piece_length`, `piece_wire_cap`, `piece_pin_cap`, and `piece_total_cap`.

Example

```
piece_type : piece_length;
```

The *piece_wire_cap* , *piece_pin_cap* , and *piece_total_cap* values represent the piecewise linear model extensions that cause modeling to use capacitance instead of length. These values are used to indicate wire capacitance alone, total pin capacitance, or the total wire and pin capacitance. If the `piece_type` attribute is not defined, modeling defaults to the `piece_length` model.

2.5.2 piece_define Attribute

This attribute defines the pieces used in the piecewise linear delay model. With this attribute, you can define the ranges of length or capacitance for indexed variables, such as `rise_pin_resistance`, used in the delay equations.

You must include in this statement all ranges of wire length or capacitance for which you want to enter a unique attribute value.

Example

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

Each capacitance is a positive floating-point number defining the lower limit of the respective range. A piece of wire whose capacitance is between *range0* and *range1* is identified as *piece0* , a capacitance between *range1* and *range2* is piece 1, and so on.

You must include in the `piece_define` statement all ranges of wire length or capacitance for which you want to enter a unique attribute value.

3. Building Environments

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

The environment attributes include various attributes and tasks, covered in the following sections:

- Library-Level Default Attributes
- Defining Operating Conditions
- Defining Power Supply Cells
- Defining Wire Load Groups
- Specifying Delay Scaling Attributes

3.1 Library-Level Default Attributes

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

3.1.1 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;
```

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;
```

3.1.2 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

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 3-1 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 3-2 sets the default timing attributes for a generic delay model.

Example 3-2 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 3-3 shows the default timing attributes setting for a piecewise linear timing delay model.

Example 3-3 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;
  ...
}
```

3.1.3 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.

Syntax

default_wire_load : *wire_load_name*;

Example

default_wire_load : WL1;

default_wire_load_capacitance Attribute

Specifies a value for the default wire load capacitance.

Syntax

default_wire_load_capacitance : *value*;

Example

default_wire_load_capacitance : .05;

default_wire_load_resistance Attribute

Specifies a value for the default wire load resistance.

Syntax

default_wire_load_resistance : *value*;

Example

default_wire_load_resistance : .067;

default_wire_load_area Attribute

Specifies a value for the default wire load area.

Syntax

```
default_wire_load_resistance : value;
```

Example

```
default_wire_load_area : 0.33;
```

3.1.4 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.

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".

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 ...];
```

3.1.5 Examples of Library-Level Default Attributes

Example 3-4 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 3-4 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;
...
}
```

3.2 Defining Operating Conditions

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

3.2.1 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.

tree_type : model ;

The definition for the environment interconnect model.. The model is one of the following three models:

- *best_case_tree*

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

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

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

3.2.2 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 3-5 describes the SLOW_RANGE and FAST_RANGE timing ranges.

Example 3-5 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.

3.3 Defining Power Supply Cells

Use the `power_supply` group to model multiple power supply cells.

3.3.1 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) ;
}
```

3.4 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.

3.4.1 wire_load Group

The `wire_load` group has an extended `fanout_length` complex attribute.

Define the `wire_load` group at the library level.

Syntax

```
wire_load(name){
   resistance : value ;
   capacitance : value ;
```

```
  area : value ;
  slope : value ;
  fanout_length(fanout_int, length_float,\      average_capacitance_float,standard_deviation_float,
\       number_of_nets_int);
}
```

In a `wire_load` group, you define the estimated wire length as a function of fanout. You can also define scaling factors to derive wire resistance, capacitance, and area from a given length of wire.

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

You can define the following simple attributes in a `wire_load` group:

`resistance : value ;`

Specifies a floating-point number representing wire resistance per unit length of interconnect wire.

`capacitance : value ;`

Specifies a floating-point number representing capacitance per unit length of interconnect wire.

`area : value ;`

Specifies a floating-point number representing the area per unit length of interconnect wire.

`slope : value ;`

Specifies a floating-point number representing slope. This attribute characterizes linear fanout length behavior beyond the scope of the longest length described by the `fanout_length` attributes.

You can define the following complex attribute in a `wire_load` group:

*`fanout_length ( fanout`$_{int}$ *, length*$_{float}$ *, average_capacitance*$_{float}$ *\ standard_deviation*$_{float}$ *, number_of_nets*$_{int}$ *);*

`fanout_length` is a complex attribute that defines values that represent fanout and length. The *fanout* value is an integer; *length* is a floating-point number.

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

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

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

Specify the `interconnect_delay` values.

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

3.4.2 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.

3.5 Specifying Delay Scaling Attributes

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

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

- k-factors that apply to the entire library and are defined at the library level.
- User-selected operating conditions that override the values in the library for an individual cell (see "Scaling Factors for Individual Cells").

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

3.5.1 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.

3.5.2 Slope Sensitivity Factors

You can define slope sensitivity factors only in a library using a CMOS linear delay model.

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.

3.5.3 Drive Capability Factors

You can define drive capability factors only in a library using a CMOS linear delay model.

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.

3.5.4 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.

3.5.5 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.

3.5.6 Pin Resistance Factors

You can define pin resistance factors only in libraries that use a CMOS piecewise linear delay model.

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.

3.5.7 Intercept Delay Factors

You can define intercept delay factors only in libraries that use a CMOS piecewise linear delay model.

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.

3.5.8 Power Scaling Factors

You can define power factors only in libraries that use a CMOS technology. Power scaling factors scale the power computation according to 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.

3.5.9 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

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.

3.5.10 Delay Scaling Factors Example

Example 3-6 shows delay scaling factors for a CMOS generic delay model.

Example 3-6 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 3-6, 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.

3.5.11 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 3-7 uses the special scaling factors.

Example 3-7 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 3-7 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.

3.5.12 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

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.

4. Defining Core Cells

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

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

4.1 Defining cell Groups

A `cell` group defines a single cell in the technology library. This section discusses the attributes in a `cell` group.

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

- “Defining Bused Pins”
- “Defining Signal Bundles”
- “Defining scaled_cell Groups ”

See Example 4-2 for a sample cell description.

4.1.1 cell Group

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

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

Use a *name* that corresponds to the name the ASIC vendor uses for the cell.

When naming cells, remember that names are case-sensitive; cell names AND2, and2, and And2 are all different. Cell names beginning with a number must be enclosed in quotation marks.

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

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

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

- `area`
- `bundle` ()
- `bus` ()
- `cell_footprint`
- `clock_gating_integrated_cell`
- `contention_condition`
- `handle_negative_constraint`
- `is_clock_gating_cell`
- `map_only`
- `pad_cell`
- `pad_type`
- `pin_equal`
- `pin_opposite`
- `preferred`
- `scaling_factors`

4.1.2 area Attribute

This attribute specifies the cell area.

Syntax

area : *float* ;

float

A floating-point number. No units are explicitly given for the value, but you should use the same unit for the area of all cells in a library. Typical area units include gate equivalents, square microns, and transistors.

Example

```
area : 2.0;
```

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

4.1.3 cell_footprint Attribute

This attribute assigns a footprint class to a cell.

Syntax

cell_footprint : *class_name*$_{id}$;

class_name

A character string that represents a footprint class. The string is case-sensitive: And4 is different from and4.

Example

```
cell_footprint : 5MIL ;
```

Characters in the string are case-sensitive.

Use this attribute to assign the same footprint class to all cells that have the same geometric layout characteristics.

Cells with the same footprint class are considered interchangeable and can be swapped during in-place optimization.

If the `in_place_swap_mode` attribute is set to match_footprint, a cell can have only one footprint. Cells without `cell_footprint` attributes are not swapped during in-place optimization.

4.1.4 clock_gating_integrated_cell Attribute

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

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

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

Syntax

clock_gating_integrated_cell : generic|*value*$_{id}$;

generic

When you specify an attribute value of generic, the actual type of clock gating integrated cell structure is determined byaccessing the state_function specified on the library pin.

Note:

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

value

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

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

Example

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

Table 4-1 lists some example values for the `clock_gating_integrated_cell` attribute:

Table 4-1 Values for the clock_gating_integrated_cell Attribute

When the Value is:	The integrated cell must contain
`latch_negedge`	Latch-based gating logic.Logic appropriate for falling-edge-triggered registers.
`latch_posedge_postcontrol`	Latch-based gating logic.Logic appropriate for rising-edge-triggered registers.Test-control logic located after the latch.
`latch_negedge_precontrol`	Latch-based gating logic.Logic appropriate for falling-edge-triggered registers.Test-control logic located before the latch.
`none_posedge_control_obs`	Latch-free gating logic. Logic appropriate for rising-edge-triggered registers.Test-control logic (no latch).Observability logic.

Setting Pin Attributes for an Integrated Cell

The clock-gating software requires that you set the pins of your integrated cells by using the attributes listed in Table 4-2. Setting some of the pin attributes, such as those for test and observability, is optional.

Table 4-2 Pin Attributes for Integrated Clock-Gating Cells

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

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

- "clock_gate_clock_pin Attribute"
- "clock_gate_enable_pin Attribute"
- "clock_gate_obs_pin Attribute"
- "clock_gate_out_pin Attribute"
- "clock_gate_test_pin Attribute"

You set both the setup and hold arcs on the enable pin by setting the `clock_gate_enable_pin` attribute for the integrated cell to true. The setup and hold arcs for the cell are determined by the edge values you enter for the `clock_gating_integrated_cell` attribute. Table 4-3 lists the edge values and the corresponding setup and hold arcs.

Table 4-3 Values of the clock_gating_integrated_cell Attributes

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

4.1.5 contention_condition Attribute

Specifies the contention conditions for a cell.Contention is a clash of 0 and 1 signals. In certain cells, it can be a forbidden condition and cause circuits to short.

Syntax

contention_condition : "*Boolean expression*" ;

Example

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

4.1.6 handle_negative_constraint Attribute

Specifies whether the cell needs negative constraint handling. It is an optional Boolean attribute for timing constraints in a `cell` group.

Syntax

handle_negative_constraint : true | false ;

Example

```
handle_negative_constraint : true ;
```

If you omit this attribute, the VITAL generator does not write out the negative constraint handling structure.

4.1.7 pad_cell Attribute

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

Syntax

pad_cell : true | false ;

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

Example

```
pad_cell : true ;
```

4.1.8 pin_equal Attribute

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

Syntax

pin_equal ("*name_list*") ;

name_list

A list of input or output pins whose values must be equal.

4.1.9 pin_opposite Attribute

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

Syntax

pin_opposite ("*name_list1*", "*name_list2*") ;

name_list1 , name_list2

A `name_list` of output pins requires the supplied output values to be opposite. A `name_list` of input pins requires the supplied input values to be opposite.

Example

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

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

Note:

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

4.1.10 scaling_factors Attribute

This attribute applies the scaling factors defined in the `scaling_factors` group.

Syntax

scaling_factors : $group\_name_{id}$;

group_name

Name of the set of special scaling factors in a `scaling_factors` statement at the library level.

Example

```
scaling_factors : IO_PAD_SCALING ;
```

You can define a special set of scaling factors in the library-level group called `scaling_factors` and apply these scaling factors to selected cells, using the `scaling_factors` cell attribute. You can apply library-level scaling factors to the majority of cells in your library while using these constructs to provide more-accurate scaling factors for special cells.

By default, all cells without a `scaling_factors` attribute continue to use the library-level scaling factors.

Example 4-1 shows one of these special scaling factors in the library description and cell description.

Example 4-1 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 4-1 defines a scaling factor group called IO_PAD_SCALING that contains scaling factors different from the library-level scaling factors. The `scaling_factors` attribute in the INPAD_WITH_HYSTERESIS cell is set to IO_PAD_SCALING, so all scaling factors set in the IO_PAD_SCALING group are applied to this cell.

4.1.11 vhdl_name Attribute

This attribute defines valid VHDL object names.

Syntax

vhdl_name : "*name*$_{id}$" ;

Example

```
vhdl_name : "INb" ;
```

4.1.12 type Group

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

Example

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

4.1.13 cell Group Example

Example 4-2 shows cell definitions that include some of the CMOS cell attributes described in this section.

Example 4-2 cell Group Example

```
library (cell_example){
  date : "August 14, 2002";
  revision : 2000.03;
  scaling_factors(IO_PAD_SCALING) {
    k_volt_intrinsic_rise : 0.846;
  }
  cell (inout){
    pad_cell : true;
    dont_use : true;
    dont_fault : sa0;
    dont_touch : true;
    vhdl_name : "inpad";
    area : 0;    /* pads do not normally consume internal
           core area */
    cell_footprint : 5MIL;
    scaling_factors : IO_PAD_SCALING;
    pin (A) {
      direction : input;
      capacitance : 0;
    }
    pin (Z) {
      direction : output;
      function : "A";
      timing () {
      ...
      }
    }
  }
  cell(inverter_med){
    area : 3;
    preferred : true;
    pin (A) {
      direction : input;
      capacitance : 1.0;
    }
    pin (Z) {
      direction : output;
      function : "A' ";
      timing () {
      ...
      }
    }
  }
  cell(nand){
    area : 4;
    pin(A) {
      direction : input;
      capacitance : 1;
      fanout_load : 1.0;
    }
    pin(B) {
      direction : input;
      capacitance : 1;
      fanout_load : 1.0;
    }

    pin (Y) {
      direction : output;
      function : "(A * B)' ";
      timing() {
      ...
      }
    }
  }
  cell(buff1){
    area : 3;
```

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

4.2 Defining Cell Routability

To add routability information for the cell, define a `routing_track` group at the `cell` group level.

4.2.1 routing_track Group

A `routing_track` group is defined at the `cell` group or `model` group level.

Syntax

library (*name*$_{string}$){
  cell (*name*$_{string}$){
    routing_track (*routing_layer_name*$_{string}$){
     ... *routing track description* ...
    }
  }
}

A `routing_track` group contains the following attributes:

- `tracks`
- `total_track_area`

Names must be unique for each `routing_track` group and must be declared in the `routing_layers` attribute of the `library` group.

tracks Attribute

This attribute indicates the number of tracks available for routing on any particular layer. Use an integer larger than or equal to 0. This attribute is not currently used for optimization reporting.

Syntax

tracks : *value*$_{int}$;

value

A number larger than or equal to 0.

Example

```
tracks : 2 ;
```

total_track_area Attribute

This attribute specifies the total routing area of the routing tracks.

Syntax

total_track_area : *value*$_{float}$;

value

A floating-point number larger than or equal to 0.0 and less than or equal to the area on the cell.

Example

```
total_track_area : 0.2;
```

Each routing layer declared in the `routing_layers` attribute must have a corresponding `routing_track` group in a cell. If it does not, a warning is issued when the library is compiled. Do not use two `routing_track` groups for the same routing layer in the same cell.

Example 4-3 shows a library that contains routability information.

Example 4-3 A Library With Routability

```
library(lib_with_routability) {
 default_min_porosity : 15.0;
 routing_layers("metal2", "metal3");
 cell("ND2") {
  area :1;
  ...
 }
 cell("ND2P") {
  area : 2;
  routing_track(metal2) {
   tracks : 2;
   total_track_area : 0.2;
  }
  routing_track(metal3) {
   tracks : 4;
   total_track_area : 0.4;
  }
  ...
 }
 ...
}
```

Note:

For a scaled cell or test cell, routability information is not allowed. For pad cells, routability information is optional.

4.3 Defining pin Groups

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

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

Example 4-9 shows a `pin` group specification.

4.3.1 pin Group

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

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

See "Defining Bused Pins" for descriptions of bus groups. See "Defining Signal Bundles" for descriptions of bundle groups.

The pin groups are also valid within test_cell groups. They have different requirements from pin groups in cell, bus, or bundle groups. See "Pins in the test_cell Group" for specific information and restrictions on describing test pins.

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

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

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

```
cell (AND) {
  area : 3 ;
  vhdl_name : "AND2" ;
  pin (A) {
    direction : input ;
    capacitance : 1 ;
  }
  pin (B) {
    direction : input ;
    capacitance : 1 ;
  }
}
```

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

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

4.3.2 General pin Group Attributes

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

- capacitance
- clock_gate_clock_pin
- clock_gate_enable_pin
- clock_gate_obs_pin
- clock_gate_out_pin
- clock_gate_test_pin
- complementary_pin
- connection_class
- direction
- dont_fault
- driver_type
- fall_capacitance
- fault_model
- inverted_output
- pin_func_type
- rise_capacitance
- steady_state_resistance
- test_output_only

capacitance Attribute

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

Syntax

capacitance : $value_{float}$;

value

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

The following example shows a bundle group that defines a capacitance attribute value of 1 for input pins D0, D1, D2, and D3 in bundle D:

Example

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

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

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

clock_gate_clock_pin Attribute

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

Valid values for this attribute are true and false.

Example

```
clock_gate_clock_pin : true;
```

clock_gate_enable_pin Attribute

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

Valid values for this attribute are true and false.

Example

```
clock_gate_enable_pin : true;
```

clock_gate_obs_pin Attribute

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

Valid values for this attribute are true and false.

Example

```
clock_gate_obs_pin : true;
```

clock_gate_out_pin Attribute

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

Valid values for this attribute are true and false.

Example

```
clock_gate_out_pin : true;
```

clock_gate_test_pin Attribute

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

Valid values for this attribute are true and false.

Example

```
clock_gate_test_pin : true;
```

complementary_pin Simple Attribute

The `complementary_pin` attribute supports differential I/O. Differential I/O assumes the following:

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

Syntax

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

string

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

For details on the `fault_model` attribute used to define the value when both the complementary pin and the pin that it complements are driven to the same value, see “fault_model Simple Attribute ”.

Example

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

connection_class Simple Attribute

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

Example

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

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

Example 4-4 uses connection classes. The `output_pad` cell can be driven only by the `pad_driver` cell. The pad driver’s input can be connected to internal core logic, because it has the internal connection class.

Example 4-5 shows the use of multiple connection classes for a single pin. The `high_drive_buffer` cell can drive internal core cells and pad cells, whereas the `low_drive_buffer` cell can drive only internal cells.

Example 4-4 Connection Class Example

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

Example 4-5 Multiple Connection Classes for a Pin

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

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

direction Attribute

Specifies whether the pin being described is an input, output, internal, or bidirectional pin.

Syntax

```
direction : input | output | inout | internal ;
```

Example

```
direction : output;
```

driver_type Attribute

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

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

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

pull_up

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

pull_down

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

bus_hold

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

open_drain

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

open_source

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

resistive

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

resistive_0

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

resistive_1

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

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

driver_type : "*driver_type1 driver_type2*" ;

Example

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

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

Interpretation of Driver Types

The driver type specifies one of these signal modifications:

Resolve the value of Z

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

Transform the signal

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

For output pins, the `driver_type` attribute is applied after the pin's functional evaluation. For input pins, this attribute is applied before the signal is used for functional evaluation. See Table 4-4 for the signal mappings and pin types of the different driver types.

Table 4-4 Driver Types

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

Signal Mapping:

0 and 1 strong values

L weak 0

H weak 1

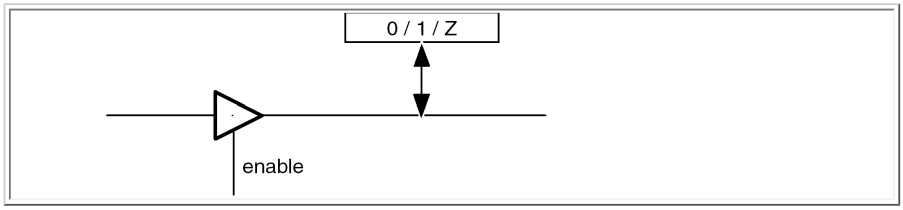
Z high impedance

S previous state

Interpreting Bus Holder Driver Type

Figure 4-1 illustrates the 01Z to 01S signal mapping for bus holders.

Figure 4-1 Interpreting bus_hold Driver Type



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

Modeling Pull-Up and Pull-Down Cells

Figure 4-2 shows a pull-up cell transistor.

Figure 4-2 Pull-Up Cell Transistor

Example 4-6 is the description of the pull-up cell transistor in Figure 4-2.

Example 4-6 Description of a Pull-Up Cell Transistor

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

Example 4-7 describes an output pin with a pull-up resistor and the bidirectional pin on a bus holder cell.

Example 4-7 Pin Driver Type Specifications

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

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

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

Note:

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

fall_capacitance Attribute

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

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

Syntax

fall_capacitance : *float* ;

float

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

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

Example

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

fault_model Simple Attribute

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

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

For details on the `complementary_pin` attribute, see “complementary_pin Simple Attribute” .

Syntax

fault_model : "*two-value string*" ;

two-value string

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

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

Example

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

Table 4-5 shows howtesting interprets the complementary pin values for this example:

Table 4-5 Interpretation of Pin Values

Pin A (noninverting pin)	DiffA (complementary_pin)	Resulting signal pin value

1	0	1
0	1	0
0	0	1
1	1	x

inverted_output Attribute

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

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

Example

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

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

pin_func_type Attribute

This attribute describes the functions of a pin.

Example

```
pin_func_type : clock_enable;
```

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

rise_capacitance Attribute

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

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

Syntax

rise_capacitance : *float* ;

float

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

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

Example

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

steady_state_resistance Attributes

When there are multiple drivers connected to an interconnect network driven by library cells and there is no direct current path between them, the driver resistances could take on different values. Use the following attributes for more-accurate modeling of steady state driver resistances in library cells.

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

Example

```
steady_state_resistance_above_high : 200 ;
```

test_output_only Attribute

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

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

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

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

Example

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

4.3.3 Describing Design Rule Checks

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

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

fanout_load Attribute

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

Syntax

fanout_load : *value*$_{float}$;

value

A floating-point number that represents the internal fanout of the input pin. There are no fixed units for `fanout_load`. Typical units are standard loads or pin count.

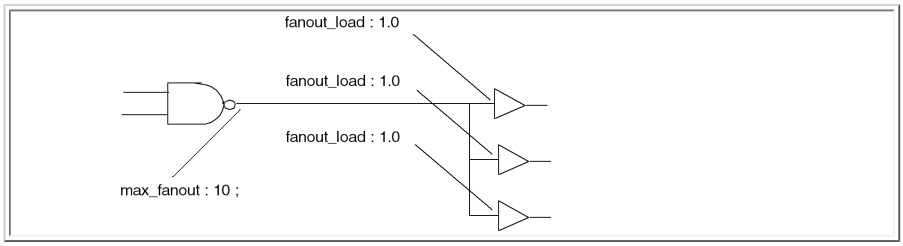
Example

```
fanout_load : 1.0;
```

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

Figure 4-3 illustrates `max_fanout` and `fanout_load` attributes for a cell.

Figure 4-3 Fanout Attributes



max_fanout Attribute

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

Syntax

max_fanout : *value*$_{float}$;

value

A floating-point number that represents the number of fanouts the pin can drive. There are no fixed units for `max_fanout`. Typical units are standard loads or pin count.

Example

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

Some designs have limitations on input load that an output pin can drive regardless of any loading contributed by interconnect metal layers. To limit the number of inputs on the same net driven by an output pin, define a `max_fanout` value for each output pin and a `fanout_load` on each input pin in a cell. (See Figure 4-3.)

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

standard value for the fanout loads of the other cells.

Although you can use capacitance as the unit for your `max_fanout` and `fanout_load` specifications, it should be used to constrain routability requirements. It differs from the capacitance pin attribute in the following ways:

min_fanout Attribute

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

Syntax

min_fanout : *value*$_{float}$;

value

A floating-point number that represents the minimum number of fanouts the pin can drive. There are no fixed units for `min_fanout`. Typical units are standard loads or pin count.

Example

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

max_transition Attribute

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

Syntax

max_transition : *value*$_{float}$;

value

A floating-point number in units consistent with other time values in the library.

Example

```
pin(A) {
 direction : input;
 max_transition : 4.2;
}
```

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

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

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

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

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

max_capacitance Attribute

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

Syntax

max_capacitance : *value*$_{float}$;

value

A floating-point number that represents the capacitive load.

Example

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

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

min_capacitance Attribute

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

Syntax

max_capacitance : *value*$_{float}$;

value

A floating-point number that represents the capacitive load.

Example

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

cell_degradation Group

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

This description is restricted to functionally related input and output pairs. You can determine the degradation value by switching some inputs while keeping other inputs constant. This causes output discharge. The degradation value for a specified input transition rate is the maximum output loading that does not cause cell degradation.

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

There are two ways to model cell degradation,

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

   This is the syntax of the cell degradation template.

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

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

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

   This is an example of a cell degradation template.

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

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

   This example shows the use of the `cell_degradation` group

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

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

   - combinational
   - three_state_enable
   - rising_edge
   - falling_edge
   - preset
   - clear

4.3.4 Describing Clocks

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

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

clock Attribute

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

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

Syntax

clock : true | false ;

Example

```
clock : true ;
```

min_period Attribute

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

Syntax

min_period : $value_{float}$;

value

A floating-point number indicating a time unit.

Example

```
min_period : 26.0;
```

min_pulse_width_high and min_pulse_width_low Attributes

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

Syntax

min_pulse_width_high : $value_{float}$;

value

A floating-point number defined in units consistent with other time values in the library. It gives the minimum length of time the pin must remain at logic 1 (`min_pulse_width_high`) or logic 0 (`min_pulse_width_low`).

Example

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

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

Yield Modeling

An example of modeling yield information is as follows.

```
library ( my_library_name ) {

   faults_lut_template ( my_faults_temp ) {
     variable_1 : fab_name;
     variable_2 : time_range;
     index_1 (  fab1, fab2, fab3 );
     index_2 ( 2005.01, 2005.07, 2006.01, 2006.07 );
   }


   cell ( and2 ) {

      functional_yield_metric () {
      average_number_of_faults ( my_faults_temp ) {
         values (  73.5, 78.8, 85.0, 92 ,\
            74.3, 78.7, 84.8, 92.2 ,\
             72.2, 78.1, 84.3, 91.0  );
         }
      }

   } /* end of cell */
} /* end of library */
```

Describing Clock Pin Functions

To define a clock pin's function, use these `pin` group attributes:

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

function Attribute

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

Syntax

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

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

Table 4-6 lists the Boolean operators that are valid in a `function` statement.

Table 4-6 Valid Boolean Operators

Operator	Description
'	Invert previous expression
!	Invert following expression
^	Logical XOR
*	Logical AND
&	Logical AND
space	Logical AND
+	Logical OR
|	Logical OR
1	Signal tied to logic 1

0	Signal tied to logic 0

Example

```
pin(Q) {
  direction : output ;
  function : "A + B" ;
}
```

Note:

Pin names beginning with a number, and pin names containing special characters, must be enclosed in double quotation marks preceded by a backslash (\), as shown here:

```
function : " \"1A\" + \"1B\" " ;
```

The absence of a backslash causes the quotation marks to terminate the function statement.

The following `function` statements all describe 2-input multiplexers. The parentheses are optional. The operators and operands are separated by spaces.

```
function : "A S + B S′" ;
function : "A & S | B & !S" ;
function : "(A * S) + (B * S′)" ;
```

Grouped Pins in function Statements

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

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

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

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

the function for Z[0] is interpreted as

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

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

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

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

output is

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

The function for Z[0] is interpreted as

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

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

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

three_state Attribute

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

Syntax

three_state : "*Boolean expression*" ;

Boolean expression

An equation defining the condition that causes the pin to go to the high-impedance state. The syntax of this equation is the same as the syntax of the `function` attribute statement described in “”. The `three_state` attribute can be used in both combinational and sequential `pin` groups, with `bus` or `bundle` variables.

Example 4-8 Three-State Cell Description

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

x_function Attribute

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

Syntax

x_function : "*Boolean expression*" ;

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

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

1. x_function
2. three_state
3. function

Example

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

state_function Attribute

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

Syntax

state_function : "*Boolean expression*" ;

Example

```
state_function : QN;
```

internal_node Attribute

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

Syntax

internal_node : pin_name_{id} ;

pin_name

Name of either an internal or output pin.

Example

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

4.3.5 CMOS pin Group Example

Example 4-9 shows pin attributes in a CMOS library.

Example 4-9 CMOS pin Group Example

```
library(example){
  date : "May 14, 2002";
  revision : 2002.05;
  ...
  cell(AN2) {
    area : 2;
    pin(A) {
    direction : input;
    capacitance : 1.3;
    fanout_load : 2;   /* internal fanout load */
    max_transition : 4.2;/* design-rule constraint */
  }
  pin(B) {
    direction : input;
    capacitance : 1.3;
  }
  pin(Z) {
    direction : output;
    function : "A * B";
    max_transition : 5.0;
    timing() {
      intrinsic_rise : 0.58;
      intrinsic_fall : 0.69 ;
      rise_resistance : 0.1378;
      fall_resistance : 0.0465;
      related_pin : "A B";
    }
  }
}
```

4.4 Defining Bused Pins

To define bused pins, use these groups:

- `type` group
- `bus` group

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

4.4.1 type Group

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

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

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

name

Identifies the bus type.

A `type` group can one of the following:

base_type

Only the array base type is supported.

data_type

Only the bit data type is supported.

bit_width

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

bit_from

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

bit_to

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

downto

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

Example 4-10 illustrates a `type` group statement.

Example 4-10 type Group Statement

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

It is not necessary to use all the `type` group attributes. For example, the `type` group statements in Example 4-11 are both valid descriptions of BUS4 in Example 4-10.

Example 4-11 Alternative type Group Statements

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

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

4.4.2 bus Group

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

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

A bus group contains the following elements:

- bus_type attribute
- pin groups

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

4.4.3 bus_type Attribute

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

Syntax

bus_type : *name* ;

4.4.4 Pin Attributes and Groups

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

All pin attributes are valid inside bus and pin groups. See "General pin Group Attributes" for a description of pin attributes. The direction attribute value of all bus members must be the same.

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

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

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

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

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

You can also define pin groups for a range of bus members. A range of bus members is defined by a beginning value and an ending value, separated by a colon. No spaces can appear between the colon and the member numbers. The following example illustrates a pin group that defines a new capacitance attribute value for bus members 0, 1, 2, and 3 in bus A:

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

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

See Table 4-7 for a comparison of bused and single-pin formats.

Table 4-7 Comparison of Bused and Single-Pin Formats

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

4.4.5 Sample Bus Description

Example 4-12 is a complete bus description that includes `type` and `bus` groups. It also shows the use of bus variables in `function`, `related_pin`, `pin_opposite`, and `pin_equal` attributes.

Example 4-12 Bus Description

```
library (ExamBus) {
  date : "May 14, 2002";
   revision : 2002.05;
   bus_naming_style :"%s[%d]";/* Optional; this is the
             default */
  type (bus4) {
    base_type : array;/* Required */
    data_type : bit;/* Required if base_type is array */
    bit_width : 4;/* Optional; default is 1 */
    bit_from : 0;/* Optional MSB; defaults to 0 */
    bit_to : 3;/* Optional LSB; defaults to 0 */
    downto : false;/* Optional; defaults to false */
  }
  cell (bused_cell) {
    area : 10;
    bus (A) {
      bus_type : bus4;
      direction : input;
      capacitance : 3;
      pin (A[0:2]) {
        capacitance : 2;
      }
      pin (A[3]) {
       capacitance : 2.5;
      }
    }
    bus (B) {
      bus_type : bus4;
      direction : input;
      capacitance : 2;
    }
    pin (E) {
      direction : input ;
      capacitance 2 ;
    }
    bus(X) {
      bus_type : bus4;
      direction : output;
      capacitance : 1;
      pin (X[0:3]) {
```

```
    function : "A & B'";
    timing() {
      related_pin : "A B";
      /* A[0] and B[0] are related to X[0],
      A[1] and B[1] are related to X[1], etc. */
    }
  }
}
bus (Y) {
  bus_type : bus4;
  direction : output;
  capacitance : 1;
  pin (Y[0:3]) {
    function : "B";
    three_state : "!E";
    timing () {
      related_pin : "A[0:3] B E";
    }
    internal_power() {
      when: "E" ;
      related_pin : B ;
      power() {
      ...
      }
    }
    internal_power() {
      related_pin : B ;
      power() {
      ...
      }
    }
  }
}
bus (Z) {
  bus_type : bus4;
  direction : output;
  pin (Z[0:1]) {
    function : "!A[0:1]";
    timing () {
      related_pin : "A[0:1]";
    }
    internal_power() {
    related_pin : "A[0:1]";
    power() {
    ...
    }
  }
}
pin (Z[2]) {
  function "A[2]";
  timing () {
    related_pin : "A[2]";
  }
  internal_power() {
    related_pin : "A[0:1]";
    power() {
    ...
    }
  }
}
pin (Z[3]) {
  function : "!A[3]";
  timing () {
    related_pin : "A[3]";
  }
  internal_power() {
    related_pin : "A[0:1]";
    power() {
      ...
    }
  }
```

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

4.5 Defining Signal Bundles

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

4.5.1 bundle Group

You define a `bundle` group in a `cell` group, as illustrated here:

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

A `bundle` group contains the following elements:

members attribute

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

pin attributes

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

4.5.2 members Attribute

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

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

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

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

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

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

4.5.3 pin Attributes

For information about pin attributes, see “General pin Group Attributes” .

Example 4-13 shows a `bundle` group in a multibit latch.

Example 4-13 Multibit Latch With Signal Bundles

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

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

4.6 Defining Layout-Related Multibit Attributes

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

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

Syntax

single_bit_degenerate : *"cell_name*$_{id}$";

cell_name

A character string identifying a single-bit cell.

Example 4-14 shows multibit library cells with the `single_bit_degenerate` attribute.

Example 4-14 Multibit Cells With single_bit_degenerate Attribute

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

```
  }
}

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

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

4.7 Defining scaled_cell Groups

Not all cells scale uniformly. 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. The values you set for a cell under one set of operating conditions do not produce accurate results for the cell when it is scaled for different conditions.

You can use a `scaled_cell` group to set the values explicitly for a cell under certain operating conditions.

Use a scaled cell only when absolutely necessary. The size of the library database increases proportionately to the number of scaled cells you define. This size increase may increase processing and load times.

4.7.1 scaled_cell Group

You can use the `scaled_cell` group to supply an alternative set of values for an existing cell. The choice is based on the set of operating conditions used.

Example

```
library (example) {
  operating_conditions(WCCOM) {
  ...
  }
  cell(INV) {
    pin(A) {
      direction : input ;
      capacitance : 1.0 ;
    }
    pin(Z) {
      direction : output ;
      function : "A'" ;
      timing() {
        intrinsic_rise : 0.36 ;
        intrinsic_fall : 0.16 ;
        rise_resistance : 0.0653 ;
        fall_resistance : 0.0331 ;
        related_pin : "A" ;
      }
    }
  }

  scaled_cell(INV,WCCOM) {
    pin(A) {
      direction : input ;
      capacitance : 0.7 ;
    }
    pin(Z) {
```

```
      direction : output ;
      timing() {
        intrinsic_rise : 0.12 ;
        intrinsic_fall : 0.13 ;
        rise_resistance : 0.605 ;
        fall_resistance : 0.493 ;
        related_pin : "A" ;
      }
    }
  }
}
```

4.8 Defining Multiplexers

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

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

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

4.8.1 Library Requirements

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

- A one-hot MUX cell in the target library should be a single-output cell.
- Its inputs can be divided into two disjoint sets of the same size as follows:
  `C={C_1, C_2, ..., C_n}` and `D={D_1, D_2, ..., D_n}`
  where n is greater than 1 and is the size of the set. Actual names of the inputs can vary.
- The `contention_condition` attribute must be set on the cell. The value of the attribute is a combinational function, f{C}, of inputs in set C that defines prohibited combinations of inputs as shown in the following examples (where size n of the set is 3):
  FC = C_0' & C_1' & C_2' | C_0 & C_1 | C_0 & C_2 | C_1 & C_2

  or

  FC = (C_0 & C_1' & C_2' | C_0' & C_1 & C_2' | C_0' & C_1' & C_2)'
- The cell must have a combinational function FO defined on the output with respect to all its inputs. This function FO must logically define, together with the contention condition, a base function F* that is a sum of n product terms, where the i_{th} term contains all the inputs in C, with C_i high and all others low and exclusively one input in D.
  Examples of the defined function are as follows (for n = 3):
  F* = C_0 & C_1' & C_2' & D_0 | C_0' & C_1 & C_2' & D_1 | C_0' & C_1' & C_2 & D_2

  or

  F* = C_0 & C_1' & C_2' & D_0' + C_0' & C_1 & C_2' & D_1' + C_0' & C_1' &
  C_2 & D_2'

  The function FO itself can take many forms, as long as it satisfies the following condition:
  `FO & FC' == F*`
  That is, when FO is restricted by FC', it should be equivalent to F*. The term FO = F*is acceptable; other examples are as follows (for n = 3):
  FO = (D_0 & C_0) | (D_1 & C_1) | (D_2 & C_2)

  or

  FO = (D_0' & C_0) | (D_1' & C_1) | (D_2' & C_2)

  Note that when FO is restricted by FC, inverting all inputs in D is equivalent to inverting the output; however, inverting

only a subset of D would yield an incompatible function. It is recommended that you use the simple form, such as those described above, or F*.

The following example shows a cell that has been properly specified.

Example

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

4.9 Defining Decoupling Capacitor Cells, Filler Cells, and Tap Cells

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

4.9.1 Syntax

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

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

4.9.2 Cell-Level Attributes

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

is_decap_cell

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

is_filler_cell

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

is_tap_cell

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

5. Defining Sequential Cells

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

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

5.1 Using Sequential Cell Syntax

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

- ff/latch format

  Cells using the ff/latch format are identified by the `ff` group and `latch` group.
- statetable format

  Cells using the statetable format are identified by the `statetable` group. The statetable format supports all the sequential cells supported by the ff/latch format. In addition, the statetable format supports more-complex sequential cells, such as the following:
  - Sequential cells with multiple clock ports, such as a cell with a system clock and a test scan clock
  - Internal-state sequential cells, such as master-slave cells
  - Multistate sequential cells, such as counters and shift registers
  - Sequential cells that also have combinational outputs
  - Sequential cells with complex clocking and complex asynchronous behavior
  - Sequential cells with multiple simultaneous input transitions
  - Sequential cells with illegal input conditions

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

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

5.2 Describing a Flip-Flop

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

5.2.1 Using an ff Group

An `ff` group can describe either a single-stage or a master-slave flip-flop. An `ff_bank` group is used to represent multibit

registers, such as a bank of flip-flops. See “Describing a Multibit Flip-Flop” for more information on the `ff_bank` group.

Syntax

```
library (lib_name) {
   cell (cell_name) {
      ...
      ff ( variable1, variable2 ) {
         clocked_on : "Boolean_expression" ;
         next_state : "Boolean_expression" ;
         clear : "Boolean_expression" ;
         preset : "Boolean_expression" ;
         clear_preset_var1 : value ;
         clear_preset_var2 : value ;
         clocked_on_also : "Boolean_expression" ;
      }
   }
}
```

variable1

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

variable2

The state of the inverting output.

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

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

clocked_on and clocked_on_also Attributes

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

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

Examples

This describes a rising-edge-triggered device:

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

This describes a falling-edge-triggered device:

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

next_state Attribute

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

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

Syntax

```
next_state : "Boolean expression" ;
```

Boolean expression

Identifies the active edge of the clock signal.

Example

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

nextstate_type Attribute

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

Syntax

```
nextstate_type : data | preset
| clear | load | scan_in | scan_enable ;
```

where

data

Identifies the pin as a synchronous data pin. This is the default value.

preset

Identifies the pin as a synchronous preset pin.

clear

Identifies the pin as a synchronous clear pin.

load

Identifies the pin as a synchronous load pin.

scan_in

Identifies the pin as a synchronous scan-in pin.

scan_enable

Identifies the pin as a synchronous scan-enable pin.

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

Note:

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

Example

```
nextstate_type : data;
```

Example 5-5 and Example 5-6 show how to use the `nextstate_type` attribute.

clear Attribute

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

The example defines an active-low clear signal.

Example

```
clear : "CD′" ;
```

For more information about the `clear` attribute, see “Describing a Single-Stage Flip-Flop”.

preset Attribute

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

The example defines an active-high preset signal.

Example

```
preset : "PD′" ;
```

For more information about the preset attribute, see “Describing a Single-Stage Flip-Flop”.

clear_preset_var1 Attribute

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

Example

```
clear_preset_var1 : L;
```

For more information about the `clear_preset_var1` attribute, including its function and values, see “Describing a Single-Stage Flip-Flop”.

clear_preset_var2 Attribute

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

Example

```
clear_preset_var2 : L ;
```

For more information about the `clear_preset_var2` attribute, including its function and values, see “Describing a Single-Stage Flip-Flop”.

ff Group Examples

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

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

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

In some flip-flops, the next state depends on the current state. In this case, the first state variable, *variable1* (IQ in the example), can be used in the `next_state` statement; the second state variable, IQN, cannot.

For example, the `ff` declaration for a JK flip-flop looks like this:

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

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

5.2.2 Describing a Single-Stage Flip-Flop

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

Table 5-1 shows the functions of the attributes in the `ff` group for a single-stage flip-flop.

Table 5-1 Function Table for Single-Stage Flip-Flop

active edge	clear	preset	variable1	variable2
clocked_on	inactive	inactive	next_state	!next_state
--	active	inactive	0	1
--	inactive	active	1	0
--	active	active	clear_preset_var1	clear_preset_var2

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

```
clear : "CD′" ;
```

The `clear_preset_var1` and `clear_preset_var2` attributes give the value that *variable1* and *variable2* have when `clear` and `preset` are both active at the same time. Valid values are shown in Table 5-2.

Table 5-2 Valid Values or the clear_preset_var1 and clear_preset_var2 Attributes

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

1 *Use these values to generate VHDL models.*

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

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

Example 5-1 is an `ff` group for a single-stage D flip-flop with rising-edge sampling, negative clear and preset, and output pins set to 0 when both `clear` and `preset` are active (low).

Example 5-1 Single-Stage D Flip-Flop

```
ff(IQ, IQN) {
  next_state : "D" ;
  clocked_on : "CP" ;
  clear : "CD'" ;
  preset : "PD'" ;
  clear_preset_var1 : L ;
  clear_preset_var2 : L ;
}
```

Example 5-2 is an `ff` group for a single-stage, rising-edge-triggered JK flip-flop with scan input, negative clear and preset, and output pins set to 0 when `clear` and `preset` are both active.

Example 5-2 Single-Stage JK Flip-Flop

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

Example 5-3 is an `ff` group for a D flip-flop with synchronous negative clear.

Example 5-3 D Flip-Flop With Synchronous Negative Clear

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

5.2.3 Describing a Master-Slave Flip-Flop

The syntax for a master-slave flip-flop is the same as for a single-stage device, except that it includes the `clocked_on_also` attribute. Table 5-3 shows the functions of the attributes in the `ff` group for a master-slave flip-flop.

Table 5-3 Function Tables for Master-Slave Flip-Flop

active_edge	clear	preset	internal1	internal2	variable1	variable2
clocked_on	inactive	inactive	next_state	!next_state		
clocked_on_also	inactive	inactive			internal1	internal2
	active	active	clear_ preset_ var1	clear_ preset_ var2	clear_ preset_ var1	clear_ preset_ var2
	active	inactive	0	1	0	1
	inactive	active	1	0	1	0

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

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

Note:

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

Example 5-4 shows an ff group for a master-slave D flip-flop with rising-edge sampling, falling-edge data transfer, negative clear and preset, and output values set high when clear and preset are both active.

Example 5-4 Master-Slave D Flip-Flop

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

5.3 Using the function Attribute

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

Example 5-5 shows a complete functional description of a rising-edge-triggered D flip-flop with active-low clear and preset.

Example 5-5 D Flip-Flop Description

```
cell (dff) {
 area : 1 ;
 pin (CLK) {
  direction : input ;
  capacitance : 0 ;
 }
 pin (D) {
  nextstate_type : data;
  direction : input ;
  capacitance : 0 ;
 }
 pin (CLR) {
  direction : input ;
  capacitance : 0 ;
 }
 pin (PRE) {
  direction : input ;
  capacitance : 0 ;
 }
 ff (IQ, IQN) {
  next_state : "D" ;
  clocked_on : "CLK" ;
  clear : "CLR'" ;
  preset : "PRE'" ;
  clear_preset_var1 : L ;
  clear_preset_var2 : L ;
 }
 pin (Q) {
  direction : output ;
  function : "IQ" ;
 }
 pin (QN) {
  direction : output ;
   function : "IQN" ;
```

```
 }
} /* end of cell dff */
```

5.4 Describing a Multibit Flip-Flop

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

The syntax is similar to that of the ff group; see “Describing a Flip-Flop” .

Syntax

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

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

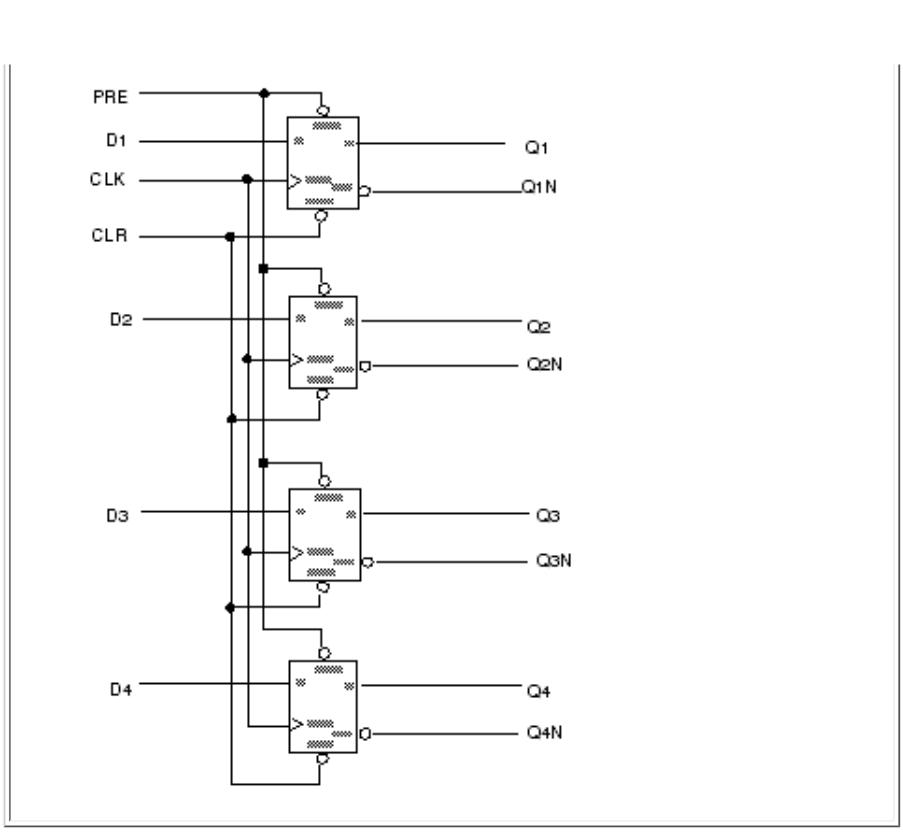
Three-state output pins are allowed; you can add a three_state attribute to an output bus or bundle to specify this function.

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

Figure 5-1 shows a multibit register containing four rising-edge-triggered D flip-flops with clear and preset.

Example 5-6 is the description of the multibit register shown in Figure 5-1.

Figure 5-1 Multibit Flip-Flop



Example 5-6 Multibit D Flip-Flop Register

```
cell (dff4) {
 area : 1 ;
 pin (CLK) {
  direction : input ;
  capacitance : 0 ;
  min_pulse_width_low : 3 ;
  min_pulse_width_high : 3 ;
 }
 bundle (D) {
    members(D1, D2, D3, D4);
    nextstate_type : data;
    direction : input ;
    capacitance : 0 ;
    timing() {
   related_pin   : "CLK" ;
   timing_type   : setup_rising ;
   intrinsic_rise : 1.0 ;
   intrinsic_fall : 1.0 ;
  }
  timing() {
   related_pin   : "CLK" ;
   timing_type   : hold_rising ;
   intrinsic_rise : 1.0 ;
   intrinsic_fall : 1.0 ;
  }
 }
 pin (CLR) {
  direction : input ;
  capacitance : 0 ;
  timing() {
   related_pin   : "CLK" ;
   timing_type   : recovery_rising ;
   intrinsic_rise : 1.0 ;
   intrinsic_fall : 0.0 ;
  }
 }
 pin (PRE) {
  direction : input ;
```

```
 capacitance : 0 ;
 timing() {
  related_pin   : "CLK" ;
  timing_type   : recovery_rising ;
  intrinsic_rise : 1.0 ;
  intrinsic_fall : 0.0 ;
 }
}
ff_bank (IQ, IQN, 4) {
 next_state : "D" ;
 clocked_on : "CLK" ;
 clear : "CLR'" ;
 preset : "PRE'" ;
 clear_preset_var1 : L ;
 clear_preset_var2 : L ;
}
bundle (Q) {
 members(Q1, Q2, Q3, Q4);
 direction : output ;
 function : "(IQ)" ;
 timing() {
  related_pin   : "CLK" ;
  timing_type   : rising_edge ;
  intrinsic_rise : 2.0 ;
  intrinsic_fall : 2.0 ;
 }
 timing() {
  related_pin   : "PRE" ;
  timing_type   : preset ;
  timing_sense   : negative_unate ;
  intrinsic_rise : 1.0 ;
 }
 timing() {
  related_pin   : "CLR" ;
  timing_type   : clear ;
  timing_sense   : positive_unate ;
  intrinsic_fall : 1.0 ;
 }
}
bundle (QN) {
 members(Q1N, Q2N, Q3N, Q4N);
 direction : output ;
 function : "IQN" ;
 timing() {
  related_pin   : "CLK" ;
  timing_type   : rising_edge ;
  intrinsic_rise : 2.0 ;
  intrinsic_fall : 2.0 ;
 }
 timing() {
  related_pin   : "PRE" ;
  timing_type   : clear ;
  timing_sense   : positive_unate ;
  intrinsic_fall : 1.0 ;
  }
  timing() {
  related_pin   : "CLR" ;
  timing_type   : preset ;
  timing_sense   : negative_unate ;
  intrinsic_rise : 1.0 ;
 }
}
} /* end of cell dff4 */
```

5.5 Describing a Latch

To describe a level-sensitive storage device, you include a `latch` group or `statetable` group in the cell definition. This section describes how to define a latch by using the ff/latch format. See “Describing Sequential Cells With the Statetable

Format" for information about defining cells using the `statetable` group.

5.5.1 latch Group

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

Syntax

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

variable1

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

variable2

The state of the inverting output.

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

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

enable and data_in Attributes

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

Example

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

clear Attribute

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

Example

This example defines a active-low clear signal.

```
clear : "CD′" ;
```

preset Attribute

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

Example

This example defines an active-low preset signal.

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

clear_preset_var1 Attribute

The `clear_preset_var1` attribute gives the value that *variable1* has when clear and preset are both active at the same time. Valid values are shown in Table 5-4.

Example

```
clear_preset_var1 : L;
```

Table 5-4 Valid Values for the clear_preset_var1 and clear_preset_var2 Attributes

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

1 *Use these values to generate VHDL models.*

clear_preset_var2 Attribute

The `clear_present_var2` attribute gives the value that *variable2* has when `clear` and `preset` are both active at the same time. Valid values are shown in Table 5-4.

Example

```
clear_preset_var2 : L ;
```

Table 5-5 shows the functions of the attributes in the `latch` group.

Table 5-5 Function Table for latch Group Attributes

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

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

Example 5-7 shows a `latch` group for a D latch with active-high `enable` and negative `clear`.

Example 5-7 D Latch With Active-High enable and Negative clear

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

Example 5-8 shows a `latch` group for an SR latch. The `enable` and `data_in` attributes are not required for an SR latch.

Example 5-8 SR Latch

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

You can view a fully described D latch in the examples section at the end of this chapter.

5.6 Describing a Multibit Latch

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

5.6.1 latch_bank Group

The syntax is similar to that of the `latch` group. See “Describing a Latch ”.

Syntax

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

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

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

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

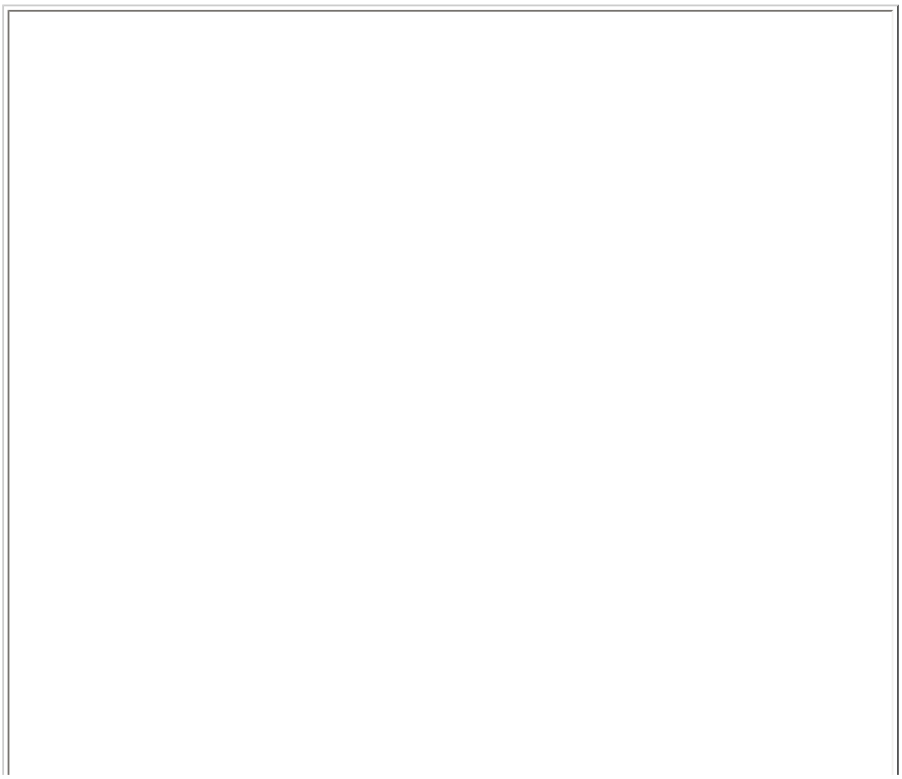
Example 5-9 shows a `latch_bank` group for a multibit register containing four rising-edge-triggered D latches.

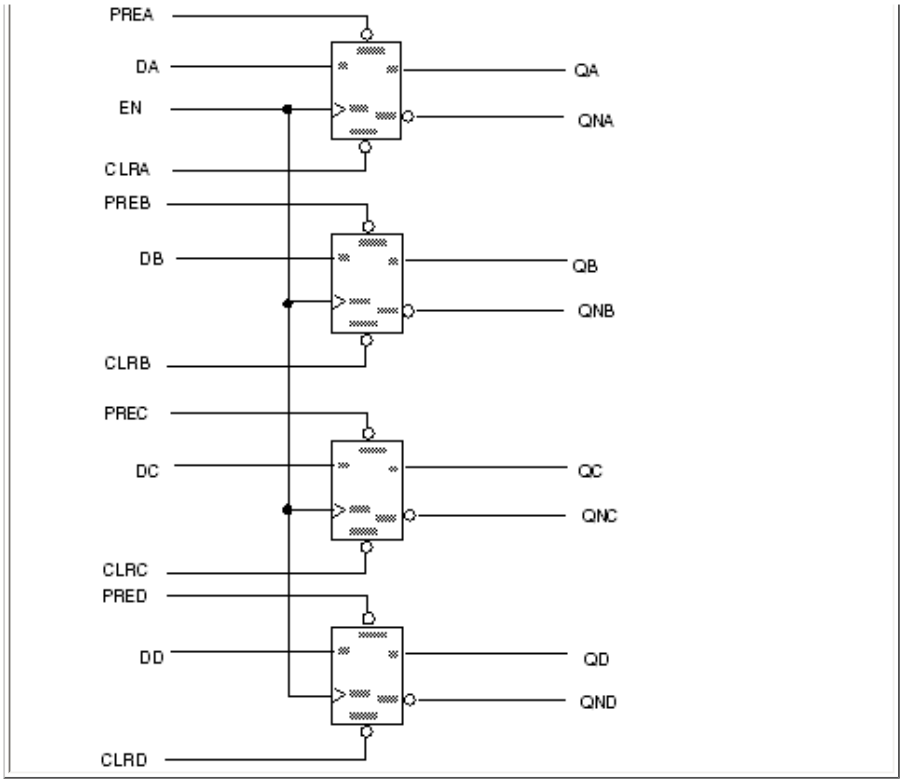
Example 5-9 Multibit D Latch

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

Figure 5-2 shows a multibit register containing four high-enable D latches with `clear`.

Figure 5-2 Multibit Latch





Example 5-10 is the cell description of the multibit register shown in Figure 5-2 that contains four high-enable D latches with `clear`.

Example 5-10 Multibit Latches With clear

```
cell (DLT2) {

/* note: 0 hold time */

 area : 1 ;
 pin (EN) {
  direction : input ;
  capacitance : 0 ;
  min_pulse_width_low : 3 ;
  min_pulse_width_high : 3 ;
 }

 bundle (D) {
  members(DA, DB, DC, DD);
  direction : input ;
  capacitance : 0 ;
  timing() {
   related_pin   : "EN" ;
   timing_type   : setup_falling ;
   intrinsic_rise : 1.0 ;
   intrinsic_fall : 1.0 ;
  }
  timing() {
   related_pin   : "EN" ;
   timing_type   : hold_falling ;
   intrinsic_rise : 0.0 ;
   intrinsic_fall : 0.0 ;
  }
 }

 bundle (CLR) {
  members(CLRA, CLRB, CLRC, CLRD);
  direction : input ;
  capacitance : 0 ;
  timing() {
```

```
  related_pin   : "EN" ;
  timing_type   : recovery_falling ;
  intrinsic_rise : 1.0 ;
  intrinsic_fall : 0.0 ;
 }
}

bundle (PRE) {
 members(PREA, PREB, PREC, PRED);
 direction : input ;
 capacitance : 0 ;
 timing() {
  related_pin   : "EN" ;
  timing_type   : recovery_falling ;
  intrinsic_rise : 1.0 ;
  intrinsic_fall : 0.0 ;
 }
}

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

bundle (Q) {
 members(QA, QB, QC, QD);
 direction : output ;
 function : "IQ" ;
 timing() {
  related_pin   : "D" ;
  intrinsic_rise : 2.0 ;
  intrinsic_fall : 2.0 ;
 }
 timing() {
  related_pin   : "EN" ;
  timing_type   : rising_edge ;
  intrinsic_rise : 2.0 ;
  intrinsic_fall : 2.0 ;
 }
 timing() {
  related_pin   : "CLR" ;
  timing_type   : clear ;
  timing_sense  : positive_unate ;
  intrinsic_fall : 1.0 ;
 }
 timing() {
  related_pin   : "PRE" ;
  timing_type   : preset ;
  timing_sense  : negative_unate ;
  intrinsic_rise : 1.0 ;
 }
}

bundle (QN) {
 members(QNA, QNB, QNC, QND);
 direction : output ;
 function : "IQN" ;
 timing() {
  related_pin   : "D" ;
  intrinsic_rise : 2.0 ;
  intrinsic_fall : 2.0 ;
 }
 timing() {
  related_pin   : "EN" ;
  timing_type   : rising_edge ;
  intrinsic_rise : 2.0 ;
  intrinsic_fall : 2.0 ;
```

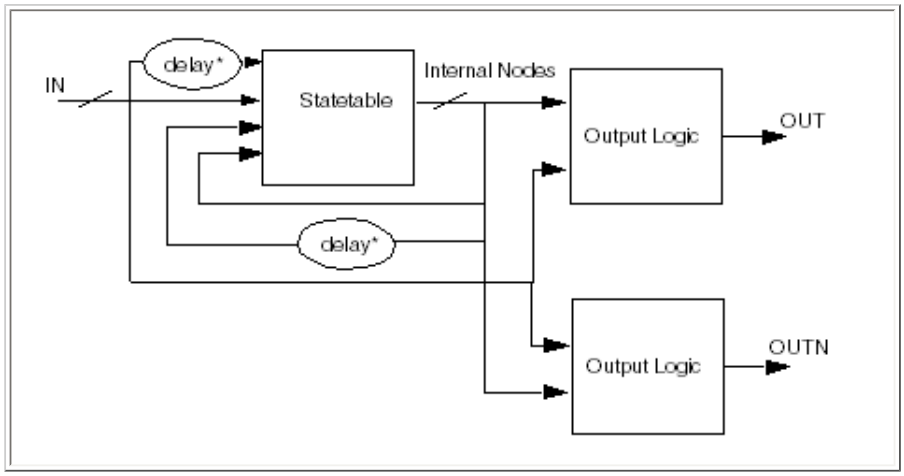
```
  }
  timing() {
   related_pin   : "CLR" ;
   timing_type   : preset ;
   timing_sense  : negative_unate ;
   intrinsic_rise : 1.0 ;
  }
  timing() {
   related_pin   : "PRE" ;
   timing_type   : clear ;
   timing_sense  : positive_unate ;
   intrinsic_fall : 1.0 ;
  }
 }
} /* end of cell DLT2 */
```

5.7 Describing Sequential Cells With the Statetable Format

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

Figure 5-3 shows how you can model each sequential output port (OUT and OUTN) in a sequential library cell.

Figure 5-3 Generic Sequential Library Cell Model



OUT and OUTN

Sequential output ports of the sequential cell.

IN

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

*delay**

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

Statetable

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

Internal Nodes

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

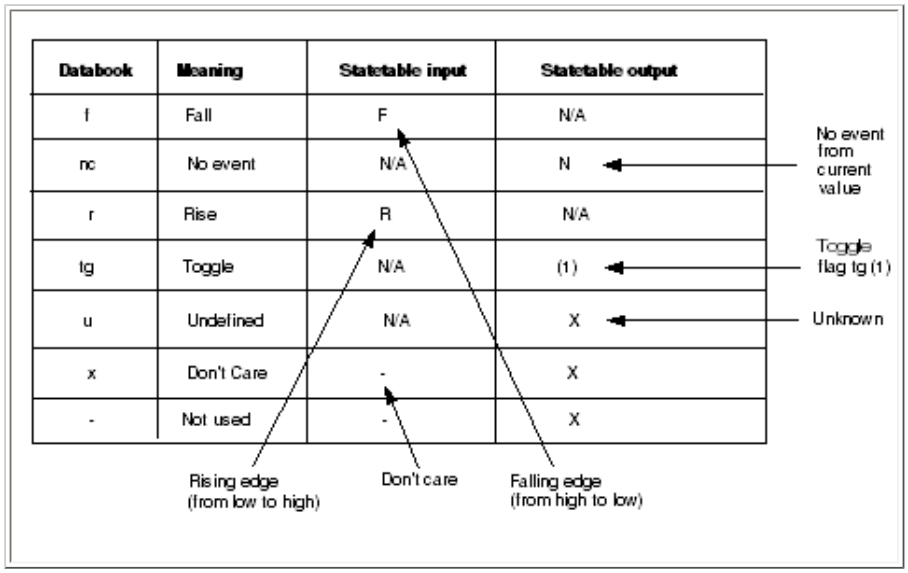
Output Logic

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

To capture the function of complex sequential cells, use the `statetable` group in the `cell` group to define the statetable in Figure 5-3. The `statetable` group syntax maps to the truth tables in databooks.

Figure 5-4 is an example of a table that maps a truth table from an ASIC vendor's databook to the statetable syntax. For table input token values, see "statetable Group" .

Figure 5-4 Mapping Databook Truth Table to Statetable



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

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

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

5.7.1 statetable Group

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

Syntax

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

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

- Give nodes unique names.
- Separate node names with white space.
- Place each rule consisting of a set of input node values, current internal values, and next internal values on a separate line, followed by a comma and the line continuation character (\). To prevent syntax errors, the line

continuation character must be followed immediately by the next line character.

- Separate node values and the colon delimiter (:) with white space.
- Insert comments only where a character space is allowed. For example, you cannot insert a comment within an H/L token or after the line continuation character (\).

Figure 5-5 shows an example of a `statetable` group.

Figure 5-5 statetable Group Example

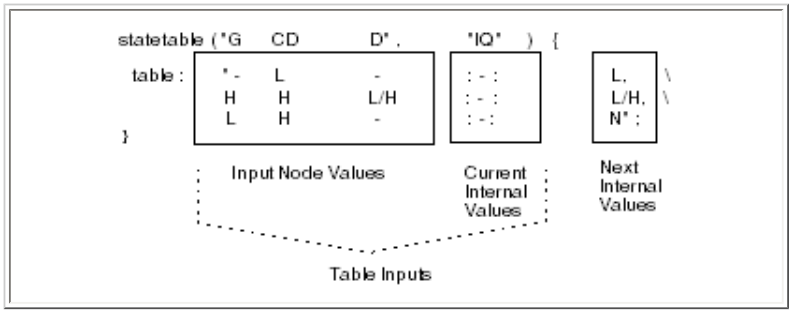


Table 5-6 shows the token values for table inputs.

Table 5-6 Legitimate Values for Table Inputs (Input and Current Internal Nodes)

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

Table 5-7 shows the token values for the next internal node.

Table 5-7 Legitimate Values for Next Internal Node

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

Note:

It is important to use the N token value appropriately. The output is never N when any input

(asynchronous or synchronous) is active. The output should be N only when all the inputs are inactive.

Using Shortcuts

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

Example 5-11 shows a fully expanded `statetable` group for a data latch with active-low `clear`.

Example 5-11 Fully Expanded statetable Group for Data Latch With Active-Low Clear

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

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

don’t care symbol (-)

For the input and current internal node, the don’t care symbol represents all permutations of L and H.

For the next internal node, the don’t care symbol means the rule does not define a value for the next internal node.

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

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

Or it can be written more concisely as follows

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

L/H and H/L

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

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

is a concise version of the following lines.

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

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

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

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

Table 5-8 shows the internal representation of the same cell.

Table 5-8 Internal Representation

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

Priority ordering of outputs

The outputs follow a prioritized order. Two rules can cover the same input combinations, but the rule defined first has priority over subsequent rules.

Note:

Use shortcuts with care. When in doubt, be explicit.

5.7.2 Partitioning the Cell Into a Model

You can partition a structural netlist to match the general sequential output model described in Example 5-12 by performing these tasks:

1. Represent every storage element by an internal node.
2. Separate the output logic from the internal node. The internal node must be a function of only the sequential cell data input when the sequential cell is triggered.

Note:

There are two ways to specify that an output does not change logic values. One way is to use the nonactive N value as the next internal value, and the other way is to have the next internal value remain the same as the current

internal value (see the italic lines in Example 5-12).

Example 5-12 JK Flip-Flop With Active-Low, Direct-Clear, and Negative-Edge Clock

```
statetable ("J K CN CD" , "IQ") {
 table : "      -  -  -  L  : -    :    L,\
    -  -  ~F  H  : -    :    N,\
    L  L  F  H  : L/H    :    L/H,\
      H  L  F  H  : -    :    H,\
      L  H  F  H  : -    :    L,\
      H  H  F  H  : L/H    :    H/L" ;
}
```

In Example 5-12, the value of the next internal node of the second rule is N, because both the clear CD and clock CN are inactive. The value of the next internal node of the third rule is L/H, because the clock is active.

5.7.3 Defining an Output pin Group

Every output pin in the cell has either an `internal_node` attribute, which is the name of an internal node, or a `state_function` attribute, which is a Boolean expression of ports and internal pins.

Every output pin can also have an optional `three_state` expression.

An output pin can have one of the following combinations:

- A `state_function` attribute and an optional `three_state` attribute
- An `internal_node` attribute, an optional `input_map` attribute, and an optional `three_state` attribute

state_function Attribute

The `state_function` attribute defines output logic. Ports in the `state_function` Boolean expression must be either input, inout that can be made three-state, or ports with an `internal_node` attribute.

The `state_function` attribute specifies the purely combinational function of input and internal pins. A port in the `state_function` expression refers only to the non-three-state functional behavior of that port. For example, if E is a port of the cell that enables the three-state, the `state_function` attribute cannot have a Boolean expression that uses pin E.

An inout port in the `state_function` expression is treated only as an input port.

Example

```
pin(Q)
  direction : output;
  state_function : "A"; /*combinational feedthrough*/
```

internal_node Attribute

The `internal_node` attribute is used to resolve node names to real port names.

The statetable is in an independent, isolated name space. The term *node* refers to the identifiers. Input node and internal node names can contain any characters except white space and comments. These node names are resolved to the real port names by each port with an `internal_node` attribute.

Each output defined by the `internal_node` attribute may have an optional `input_map` attribute.

Example

```
internal_node : "IQ";
```

input_map Attribute

The `input_map` attribute maps real port and internal pin names to each table input and internal node specified in the state table. An input map is a listing of port names, separated by white space, that correspond to internal nodes.

Syntax

input_map : *name*$_{id}$;

name

A string representing a name or a list of port names, separated by spaces, that correspond to the input pin names, followed by the internal node names.

Example

```
input_map : "Gx1 CDx1 Dx1 QN";/*QN is internal node*/
```

Mapping port and internal pin names to table input and internal nodes occurs by using a combination of the `input_map` attribute and the `internal_node` attribute. If a node name is not mapped explicitly to a real port name in the input map, it automatically inherits the node name as the real port name. The internal node name specified by the `internal_node` attribute maps implicitly to the output being defined. For example, to map two input nodes, A and B, to real ports D and CP, and to map one internal node, Z, to port Q, set the `input_map` attribute as follows:

```
input_map : "D CP Q"
```

You can use a *don't care* symbol in place of a name to signify that the input is not used for this output. Comments are allowed in the `input_map` attribute.

The delayed nature of outputs specified with the `input_map` attribute is implicit:

Internal node

When an output port is specified for this type of node, the internal node is forced to map to the delayed value of the output port.

Synchronous data input node

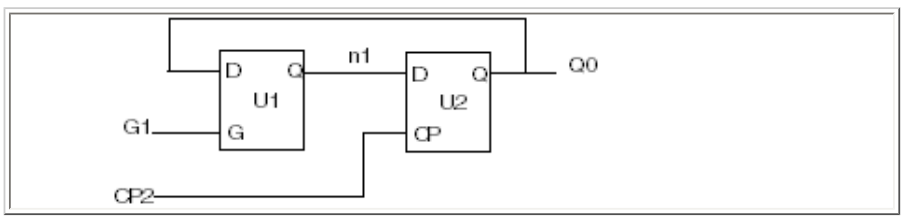
When an output port is specified for this type of node, the input is forced to map to the delayed value of the output port.

Asynchronous or clock input node

When an output port is specified for this type of node, the input is forced to map to the undelayed value of the output port.

For example, Figure 5-6 shows a circuit consisting of latch U1 and flip-flop U2.

Figure 5-6 Circuit With Latch and Flip-Flop



In Figure 5-6, internal pin n1 should map to ports "Q0 G1 n1*" and output Q0 should map to "n1* CP2 Q0*". The subtle significance is relevant during simultaneous input transitions.

With this `input_map` attribute, the functional syntax for ganged cells is identical to that of nonganged cells. You can use the input map to represent the interconnection of any network of sequential cells (for example, shift registers and counters).

The `input_map` attribute can be completely unspecified, completely specified, or can specify only the beginning ports. The default for an incompletely defined input map is the assumption that missing trailing primary input nodes and internal nodes are equal to the node names specified in the statetable. The current state of the internal node should be equal to the output port name.

inverted_output Attribute

You can define output as inverted or noninverted by using an `inverted_output` attribute on the pin. The `inverted_output` attribute is a required Boolean attribute for the statetable format that you can set for any output port. Set this attribute to false for noninverting output. This is variable1 or IQ for flip-flop or latch groups. Set this attribute to true for inverting output. This is variable2 or IQN for flip-flop or latch groups.

Example

```
inverted_output : true;
```

In the statetable format, an output is considered inverted if it has any data input with a negative sense. This algorithm may be inconsistent with a user's intentions. For example, whether a toggle output is considered inverted or noninverted is arbitrary.

5.7.4 Internal Pin Type

In ff/latch format, the `pin`, `bus`, or `bundle` groups can have a `direction` attribute with input, output, or inout values. In the statetable format, the `direction` attribute for any library cell (combinational or sequential) can have the internal value. A pin with the internal value to the `direction` attribute is treated like an output port, except that it is hidden within the library cell. It can take any of the attributes of an output pin, but it cannot have the `three_state` attribute.

An internal pin can have timing arcs and can have timing arcs related to it, allowing a distributed delay model for multistate sequential cells.

Example

```
pin (n1) {
  direction : internal;
  internal_node :"IQ"; /* use default input_map */
  ...
}
pin (QN) {
  direction : output;
  internal_node :"IQN";
  input_map : "Gx1 CDx1 Dx1 QN"; /* QN is internal node */
  three_state : "EN2";
  ...
}
pin (QNZ) {
  direction : output;
  state_function :"QN"; /* ignores QN's three state */
  three_state : "EN";
  ...
}
pin (Y) {
  direction : output;
  state_function : "A"; /* combinational feedthrough */
  ...
}
```

5.8 Critical Area Analysis Modeling

Liberty syntax models critical area analysis data for library cells in order to analyze and optimize for yield in the early implementation stage of design flow. The syntax for modeling critical area analysis data is included in the following section.

5.8.1 Syntax

```
library(my_library) {
distance_unit : enum (um, mm);
dist_conversion_factor : integer;
critical_area_lut_template (<template_name>) {
      variable_1 : defect_size_diameter;
   index_1 ("float…float");
}

device_layer(<string>) {} /* such as diffusion layer OD */
poly_layer(<string>) {} /* such as poly layer */
routing_layer(<string>) {} /* such as M1, M2, … */
cont_layer(<string>){} /* via layer, such as VIA */
cell (my_cell) {
   functional_yield_metric() {
    average_number_of_faults(<fault_tempalte>) {
      values("float…float");
   }

   critical_area_table (<template_name>) {
      defect_type : enum (open, short, open_and_short);
      related_layer : string;
      index_1 ("float…float");
      values ("float…float");
   }
…
   }
 }
}
```

5.8.2 Library-Level Groups and Attributes

This section describes library-level groups and attributes used for modeling critical area analysis data.

distance_unit and dist_conversion_factor Attributes

The `distance_unit` attribute specifies the distance unit and the resolution, or accuracy, of the values in the `critical_area_table` table in the `critical_area_lut_template` group. The distance and area values are represented as floating-point numbers that are rounded in the `critical_area_table`. The distance values are rounded by the `dist_conversion_factor` and the area values are rounded by the `dist_conversion_factor` squared.

critical_area_lut_template Group

The `critical_area_lut_template` group is a critical area lookup table used only for critical area analysis modeling. The `defect_size_diameter` is the only valid variable.

device_layer, poly_layer, routing_layer, and cont_layer Groups

Because yield calculation varies among different types of layers, Liberty syntax supports the following types of layers: device, poly, routing, and contact (via) layers.The `device_layer`, `poly_layer`, `routing_layer`, and `cont_layer` groups define layers that have critical area data modeled on them for cells in the library. The layer definition is specified at the library level. It is recommended that you declare the layers in order, from the bottom up. The layer names specified here must match the actual layer names in the corresponding physical libraries.

5.8.3 Cell-Level Groups and Attributes

This section describes cell-level groups and attributes used for modeling critical area analysis data.

critical_area_table Group

The `critical_area_table` group specifies a critical area table at the cell level that is used for critical area analysis modeling.

The critical_area_table group uses critical_area_lut_template as the template. The critical_area_table group contains the following attributes:

defect_type Attribute

The defect_type attribute value indicates whether the critical area analysis table values are measured against a short or open electrical failure when particles fall on the wafer. The following enumerated values are accepted: short, open and open_and_short. The open_and_short attribute value specifies that the critical area analysis table is modeled for both short and open failure types. If defect_type is not specified, the default value is open_and_short.

related_layer Attribute

The related_layer attribute defines the names of the layers to which the critical area analysis table values apply. All layer names must be predefined in the library's layer definitions.

index_1 Attribute

The index_1 attribute defines the defect size diameter array in the unit of distance_unit. The attribute is optional if the values for this array are the same as that in the critical_area_lut_template.

values Attribute

The values attribute defines critical area values for nonvia layers in the unit of distance_unit squared. For via layers, the values attribute specifies the number of single cuts on the layer.

5.8.4 Example

The following example shows a library with critical area analysis data modeling.

```
library(my_library) {

distance_unit : um;
dist_conversion_factor : 1000;
critical_area_lut_template (caa_template) {
       variable_1 : defect_size_diameter;
    index_1 ("0.05, 0.10, 0.15, 0.20, 0.25, 0.30");
}

device_layer("OD") {}
poly_layer("PO") {}
routing_layer("M1") {}
routing_layer("M2") {}
cont_layer("VIA") {}
…

cell (BUF) {
    functional_yield_metric() {
    critical_area_table (caa_template) {
        defect_type : open;
        related_layer : M1 ;
        index_1 ("0.08, 0.09, 0.1, 0.11, 0.12, 0.13, 0.14, 0.15, 0.16,
 0.17") ;
        values ("0.03, 0.09, 0.15, 0.22, 0.30, 0.39, 0.50, 0.62, 0.74,
0.87") ;
    }
    critical_area_table (caa_template) {
        defect_type : short;
        related_layer : M1 ;
        index_1 ("0.08, 0.09, 0.1, 0.11, 0.12, 0.13, 0.14, 0.15, 0.16,
 0.17") ;
        values ("0.03, 0.08, 0.17, 0.28, 0.40, 0.54, 0.68, 0.81, 0.95,
 1.09") ;
    }

    critical_area_table (scalar) {
        /* If no defect_type is defined, the critical area analysis value
        is used for both short and open. Define defect_type : open_and_short
```

```
        also works.*/
       related_layer : VIA;
       values ("12");
     }
     …
     }
   }
 }
```

5.9 Flip-Flop and Latch Examples

Example 5-13 through Example 5-25 show flip-flops and latches in ff/latch and statetable syntax.

Example 5-13 D Flip-Flop

```
/* ff/latch format */

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

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

Example 5-14 D Flip-Flops With Master-Slave Clock Input Pins

```
/* ff/latch format */

ff (IQ,IQN) {
 next_state : "D" ;
 clocked_on : "CK" ;
 clocked_on_also : "CKN′" ;
}

/* statetable format */
statetable(" D  CK  CKN",  "MQSQ") {
  table : " H/L R  ~F          : -  - : H/L N,\
       -  ~R        F  : H/L - : N  H/L,\
      H/L R  F  : L  - : H/L L,\
      H/L R  F  : H  - : H/L H,\
       -    ~R ~F : -  - : N  N";
}
```

Example 5-15 D Flip-Flop With Gated Clock

```
/* ff/latch format */

ff (IQ,IQN) {
 next_state : "D" ;
 clocked_on : "C1 * C2" ;
}

/* statetable format */

statetable("     D    C1  C2",  "IQ") {
  table : "     H/L    H  R  : - :    H/L,\
      H/L    R  H  : - :    H/L,\
      H/L    R  R  : - :    H/L,\
      -    -  -  : - :    N";
}
```

Example 5-16 D Flip-Flop With Active-Low Direct-Clear and Active-Low Direct-Set

```
/* ff/latch format */

ff (IQ,IQN) {
 next_state : "D" ;
 clocked_on : "CP" ;
 clear : " CD' " ;
 preset : " SD' " ;
 clear_preset_var1 : L ;
 clear_preset_var2 : L ;
}

/* statetable format */

statetable("D  CP CD SD","IQ IQN") {
 table : "H/L R  H  H : -  - : H/L  L/H,\
      -  ~R H  H : -  - : N   N,\
      -  -  L  H : -  - : L   H,\
      -  -  H  L : -  - : H   L,\
      -  -  L  L : -  - : L   L";
}
```

Example 5-17 D Flip-Flop With Active-Low Direct-Clear, Active-Low Direct-Set, and One Output

```
/* ff/latch format */

ff (IQ,IQN) {
 next_state : "D" ;
 clocked_on : "CP" ;
 clear : " CD' " ;
 preset : " SD' " ;
 clear_preset_var1 : L ;
 clear_preset_var2 : L ;
}

/* statetable format */

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

Example 5-18 JK Flip-Flop With Active-Low Direct-Clear and Negative-Edge Clock

```
/* ff/latch format */

ff (IQ, IQN) {
 next_state : " (J' K' IQ) + (J K') + (J K IQ')" ;
 clocked_on : " CN' " ;
 clear : " CD' " ;
}

/* statetable format */

statetable(" J  K  CD  CD", "IQ") {
 table : " -  -  -  L  : - : L,\
      -  -  ~F  H  : - : N,\
      L  L  F   H  : L/H : L/H \
      H  L  F   H  : - : H,\
      L  H  F   H  : - : L,\
      H  H  F   H  : L/H : H/L";
```

Example 5-19 D Flip-Flop With Scan Input Pins

```
/* ff/latch format */

ff (IQ,IQN) {
 next_state : " (D TE′) + (TI TE)" ;
 clocked_on : "CP" ;
}

/* statetable format */

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

Example 5-20 D Flip-Flop With Synchronous Clear

```
/* ff/latch format */

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

/* statetable format */

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

Example 5-21 D Latch

```
/* ff/latch format */

latch (IQ,IQN) {
 data_in : "D" ;
 enable : "G" ;
}

/* statetable format */

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

Example 5-22 SR Latch

```
/* ff/latch format */

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

```
}

/* statetable format */

statetable(" R      S",   "IQ IQN") {
 table : " H  L  : - - : L  H,\
      L  H  : - - : H  L,\
      H  H  : - - : L  L,\
      L  L  : - - : N  N";
}
```

Example 5-23 D Latch With Active-Low Direct-Clear

```
/* ff/latch format */

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

/* statetable format */

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

Example 5-24 Multibit D Latch With Active-Low Direct-Clear

```
/* ff/latch format */

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

/* statetable format */

statetable(" D  EN CL PRE","IQ IQN") {
 table : "H/L H  H  H  : -  - : H/L L/H,\
      -  L  H  H  : -  - : N  N,\
      -  -  L  H  : -  - : L  H,\
      -  -  H  L  : -  - : H  L,\
      -  -  L  L  : -  - : H  H";
}
```

Example 5-25 D Flip-Flop With Scan Clock

```
statetable(" D  S  CD SC CP", "IQ") {
 table : " H/L -  ~R R  : - : H/L,\
      -  H/L R  ~R : - : H/L,\
      -  -  ~R ~R : - : N,\
      -  -  R  R  : - : X";
}
```

5.10 Cell Description Examples

Example 5-26 through Example 5-28 illustrate of some of the concepts this chapter discusses.

Example 5-26 D Latches With Master-Slave Enable Input Pins

```
cell(ms_latch) {
 area : 16;
 pin(D) {
  direction : input;
  capacitance : 1;
 }
 pin(G1) {
  direction : input;
  capacitance : 2;
 }
 pin(G2) {
  direction : input;
  capacitance : 2;
 }
 pin(mq) {
  internal_node : "Q";
  direction : internal;
  input_map : "D G1";
  timing() {
   intrinsic_rise : 0.99;
   intrinsic_fall : 0.96;
   rise_resistance : 0.1458;
   fall_resistance : 0.0653;
   related_pin : "G1";
  }
 }
 pin(Q) {
  direction : output;
  function : "IQ";
  internal_node : "Q";
  input_map : "mq G2";
  timing() {
   intrinsic_rise : 0.99;
   intrinsic_fall : 0.96;
   rise_resistance : 0.1458;
   fall_resistance : 0.0653;
   related_pin : "G2";
  }
 }
 pin(QN) {
  direction : output;
  function : "IQN";
  internal_node : "QN";
  input_map : "mq G2";
  timing() {
   intrinsic_rise : 0.99;
   intrinsic_fall : 0.96;
   rise_resistance : 0.1458;
   fall_resistance : 0.0653;
   related_pin : "G2";
  }
 }
 ff(IQ, IQN) {
  clocked_on : "G1";
  clocked_on_also : "G2";
  next_state : "D";
 }
  statetable ( "D G", "Q QN" ) {
  table : "L/H H : - - : L/H H/L,\
      -  L : - - : N  N";
 }
}
```

Example 5-27 FF Shift Register With Timing Removed

```
cell(shift_reg_ff) {
 area : 16;
 pin(D) {
  direction : input;
  capacitance : 1;
 }
 pin(CP) {
  direction : input;
  capacitance : 2;
 }
 pin (Q0) {
  direction : output;
  internal_node : "Q";
  input_map : "D CP";
 }
 pin (Q1) {
  direction : output;
  internal_node : "Q";
  input_map : "Q0 CP";
 }
 pin (Q2) {
  direction : output;
  internal_node : "Q";
  input_map : "Q1 CP";
 }
 pin (Q3) {
  direction : output;
  internal_node : "Q";
  input_map : "Q2 CP";
 }
 statetable( "D CP", "Q QN" ) {
  table : "-  ~R : - - : N  N,\
      H/L R : - - : H/L L/H";
 }
}
```

Example 5-28 FF Counter With Timing Removed

```
cell(counter_ff) {
 area : 16;
 pin(reset) {
  direction : input;
  capacitance : 1;
 }
 pin(CP) {
  direction : input;
  capacitance : 2;
 }
 pin (Q0) {
  direction : output;
  internal_node : "Q0";
  input_map : "CP reset Q0 Q1";
 }
 pin (Q1) {
  direction : output;
  internal_node : "Q1";
  input_map : "CP reset Q0 Q1";
 }
 statetable( "CP reset", "Q0 Q1" ) {
  table : "-  L : - - : L H,\
      ~R H : - - : N N,\
      R H : L L : H L,\
      R H : H L : L H,\
      R H : L H : H H,\
      R H : H H : L L";
 }
```

```
}
```

6. Defining I/O Pads

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

To model I/O pads, you must understand the following concepts covered in this chapter:

6.1 Special Characteristics of I/O Pads

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

Pads typically have longer delays and higher drive capabilities than the cells in an integrated circuit's core. Because of their higher drive, CMOS pads sometimes exhibit noise problems. Slew-rate control is available on output pads to help alleviate this problem.

One distinguishing feature of pad cells is the voltage level at which input pads transfer logic 0 or logic 1 signals to the core or at which output pad drivers communicate logic values from the core.

Integrated circuits that communicate with one another must have compatible voltage levels at their pads. Because pads communicate with the world outside the integrated circuit, you must describe the pertinent units of peripheral library properties, such as external load, drive capability, delay, current, power, and resistance. This description makes it easier to design chips from multiple technologies.

You must capture all these properties in the library to make it possible for the integrated circuit designer to insert the correct pads during synthesis.

6.2 Identifying Pad Cells

Use the attributes described in the following sections to specify I/O pads and pad pin behaviors.

pad_cell Simple Attribute

In a `cell` group or a `model` group, the `pad_cell` attribute identifies a cell as a pad cell.

Syntax

```
pad_cell : true | false ;
```

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

Example

```
pad_cell : true ;
```

If more than one pad cell can be used to build a logical pad, use the `auxiliary_pad_cell` attribute in the cell definitions of all the component pad cells.

Syntax

auxiliary_pad_cell : true | false ;

Example

```
auxiliary_pad_cell : true ;
```

If the `pad_cell` or `auxiliary_pad_cell` attribute is omitted, the cell is treated as an internal core cell rather than as a pad cell.

Note:

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

pad_type Simple Attribute

Use the `pad_type` attribute to identify a type of `pad_cell` or `auxiliary_pad_cell` that requires special treatment.

Syntax

pad_type : *value* ;

Example

```
pad_type : clock;
```

6.2.1 is_pad Attribute

After you identify a cell as a pad cell, you must indicate which pin represents the pad. The `is_pad` simple attribute must be used on at least one pin of a cell with a `pad_cell` attribute.

The `direction` attribute indicates whether the pad is an input, output, or bidirectional pad.

Syntax

is_pad : true | false ;

Example

```
cell(INBUF){
   ...
   pin(PAD){
      direction : input ;
      is_pad : true ;
   }
}
```

6.2.2 driver_type Attribute

A `driver_type` attribute defines two types of signal modifications: transformation and resolution.

- Transformation specifies an actual signal transition from 0/1 to L/H/Z. This signal transition performs a function on an input signal and requires only a straightforward mapping.
- Resolution resolves the value Z on an existing circuit node without actually performing a function and implies a constant

(0/1) signal source as part of the resolution.

Syntax

```
driver_type : pull_up | pull_down |
   open_drain | open_source | bus_hold |
   resistive | resistive_0 | resistive_1 ;
```

pull_up

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

pull_down

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

open_drain

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

Note:

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

open_source

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

bus_hold

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

resistive

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

resistive_0

The pin is an output pin connected to power through a pull-up resistor that has a control port EN. When EN is disabled, the pull-up resistor is turned off and has no effect on the pin. When EN is enabled, a functional value of 1 evaluated at the pin turns into a weak 1, but a functional value of 0 or Z is not affected.

resistive_1

The pin is an output pin connected to ground through a pull-down resistor that has a control port EN. When EN is disabled, the pull-down resistor is turned off and has no effect on the pin. When EN is enabled, a functional value of 0 evaluated at the pin turns into a weak 0, but a functional value of 1 or Z is not affected.

Table 6-1 lists the driver types, their signal mappings, and the applicable pin types.

Table 6-1 Pin Driver Types

Driver type	Signal mapping	Pin
pull_up	01Z->01H	in, out
pull_down	01Z->01L	in, out
open_drain	01Z->0ZZ	out
open_source	01Z->Z1Z	out
bus_hold	01Z->01Z	inout
resistive	01Z->LHZ	out
resistive_0	01Z->0HZ	out
resistive_1	01Z->L1Z	out

In Table 6-1, the `pull_up`, `pull_down`, and `bus_hold` driver types define a resolution scheme. The remaining driver types define transformations.

The following example describes an output pin with a pull-up resistor and the bidirectional pin on a `bus_hold` cell.

Example

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

6.3 Defining Units for Pad Cells

To process pads for full-chip synthesis, specify the units of time, capacitance, resistance, voltage, current, and power:

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

All these attributes are defined at the library level, as described in “Defining Units”. These values are required.

6.3.1 Capacitance

The `capacitive_load_unit` attribute defines the capacitance associated with a standard load. If you already represent

capacitance values in terms of picofarads or femtofarads, use this attribute to define your base unit. If you represent capacitance in terms of the standard load of an inverter, define the exact capacitance for that inverter—for example, 0.101 pF.

Example

```
capacitive_load_unit( 0.1,ff );
```

6.3.2 Resistance

In timing groups, you can define a `rise_resistance` and a `fall_resistance` value. These values are used expressly in timing calculations. The values indicate how many time units it takes to drive a capacitive load of one capacitance unit to either 1 or 0. You do not need to provide units of measure for `rise_resistance` or `fall_resistance`.

You must supply a `pulling_resistance` attribute for pull-up and pull-down devices on pads and identify the unit to use with the `pulling_resistance_unit` attribute.

Example

```
pulling_resistance_unit : "1kohm";
```

6.3.3 Voltage

You can use the `input_voltage` and `output_voltage` groups to define a set of input or output voltage ranges for your pads. To define the units of voltage you use for these groups, use the `voltage_unit` attribute. All the attributes defined inside `input_voltage` and `output_voltage` groups are scaled by the value defined for `voltage_unit`. In addition, the `voltage` attribute in the `operating_conditions` groups also represents its values in these units.

Example

```
voltage_unit : "1V";
```

6.3.4 Current

You can define the drive current that can be generated by an output pad and also define the pulling current for a pull-up or pull-down transistor under nominal voltage conditions. Define all current values with the library-level `current_unit` attribute.

Example

```
current_unit : "1uA";
```

6.4 Describing Input Pads

To represent input pads in your technology library, you must describe the input voltage characteristics and indicate whether hysteresis applies.

The input pad properties are described in the next section. Examples at the end of this chapter describe a standard input buffer, an input buffer with hysteresis, and an input clock buffer.

input_voltage Group

An `input_voltage` group is defined in the `library` group to designate a set of input voltage ranges for your 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.

After you define an `input_voltage` group, you can use its name with the `input_voltage` simple attribute in a `pin` group 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 and use the `pin` group to assign those ranges to the cell pin, as shown here.

Example

```
pin() {
  ...
  input_voltage : my_input_voltages ;
  ...
}
```

The value of each attribute is expressed as a floating-point number, an expression, or both. Table 6-2 lists the predefined variables that can be used in an expression.

Table 6-2 Voltage-Level Variables for the input_voltage Group

CMOS/BiCMOS variable	Default value
VDD	5V
VSS	0V
VCC	5V

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

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

Example 6-1 shows a collection of `input_voltage` groups.

Example 6-1 input_voltage Groups

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

```
input_voltage(TTL_5V) {
  vil : 0.8 ;
  vih : 2.0 ;
  vimin : -0.5 ;
  vimax : VDD + 0.5 ;
}
```

6.4.1 hysteresis Attribute

Use the `hysteresis` attribute on an input pad when you anticipate a long transition time or when you expect the pad to be driven by a particularly noisy line.

You can indicate an input pad with hysteresis, using the `hysteresis` attribute. The default for this attribute is false. When it is true, the `vil` and `vol` voltage ratings are actual transition points.

Example

```
hysteresis : true;
```

Pads with hysteresis sometimes have derating factors that are different from those of cells in the core. As a result, you need to describe the timing of cells that have hysteresis with a `scaled_cell` group. This construct provides derating capabilities and minimum, typical, or maximum timing for cells.

6.5 Describing Output Pads

To represent output pads in your technology library, you must describe the output voltage characteristics and the drive-current rating of output and bidirectional pads. Additionally, you must include information about the slew rate of the pad. These output pad properties are described in the sections that follow.

Examples at the end of this chapter show a standard output buffer and a bidirectional pad.

output_voltage Group

You define an `output_voltage` group in the `library` group to designate a set of output voltage level ranges to drive output cells.

Syntax

```
library (name_string) {
  output_voltage(name_string) {
    vol : float | expression ;
    voh : float | expression ;
    vomin : float | expression ;
    vomax : float | expression ;
  }
   output_voltage (name_string) {
  ... output_voltage description ... ;
   }
}
```

The value for `vol`, `voh`, `vomin`, and `vomax` is a floating-point number or an expression. An expression allows you to define voltage levels as a percentage of VSS or VDD.

vol

The maximum output voltage generated to represent a logic 0.

voh

The minimum output voltage generated to represent a logic 1.

vomin

The minimum output voltage the pad can generate.

vomax

The maximum output voltage the pad can generate.

Table 6-3 lists the predefined variables you can use in an `output_voltage` expression attribute. Separate variables are defined for CMOS and BiCMOS.

Table 6-3 Voltage-Level Variables for the output_voltage Group

CMOS/BiCMOS variable	Default value
VDD	5V
VSS	0V
VCC	5V

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 `voltage_unit` attribute within the `library` group.

Example 6-2 shows an example of an `output_voltage` group.

Example 6-2 output_voltage Group

```
output_voltage(GENERAL) {
  vol : 0.4 ;
  voh : 2.4 ;
  vomin : -0.3 ;
  vomax : VDD + 0.3 ;
}
```

6.5.1 Drive Current

Output and bidirectional pads in a technology can have different drive-current capabilities. To define the drive current supplied by the pad buffer, use the `drive_current` attribute on an output or bidirectional pad or auxiliary pad pin. The value is in units consistent with the `current_unit` attribute you defined.

Example

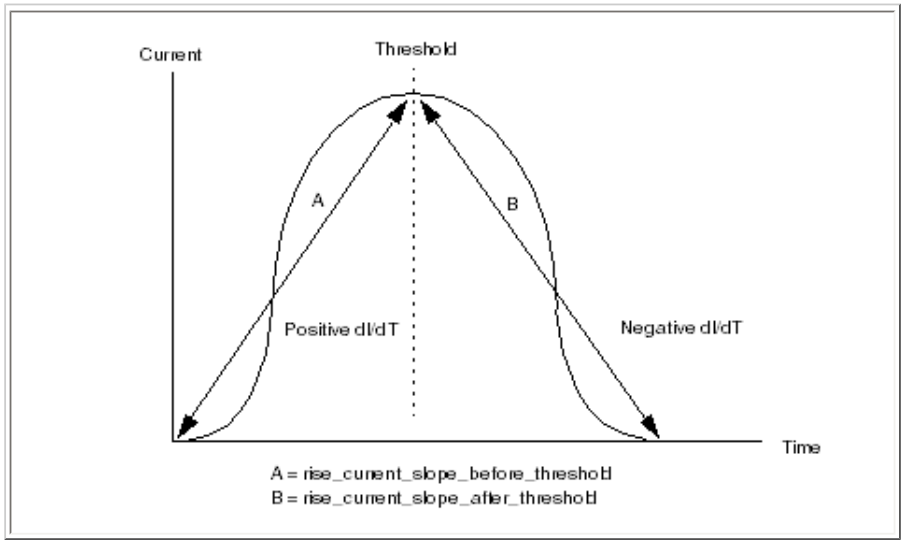
```
pin(PAD) {
  direction : output;
  is_pad : true;
  drive_current : 1.0;
}
```

6.5.2 Slew-Rate Control

The `slew_control` attribute accepts one of four possible enumerations: none, low, medium, and high; the default is none. Increasing the slew control level slows down the transition rate. This method is the coarsest way to measure the level of slew-rate control associated with an output pad.

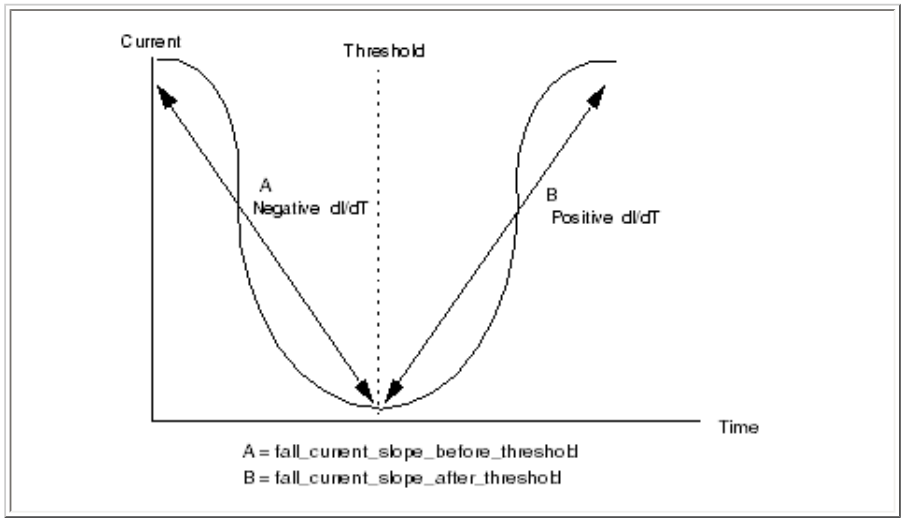
Figure 6-1 depicts the `rise_current_slope` attributes. The A value is a positive number representing a linear approximation of the change of current as a function of time from the beginning of the rising transition to the threshold point. The B value is a negative number representing a linear approximation of the current change over time from the point at which the rising transition reaches the threshold to the end of the transition.

Figure 6-1 Slew-Rate Attributes—Rising Transitions



For falling transitions, the graph is reversed: A is negative and B is positive, as shown in Figure 6-2.

Figure 6-2 Slew-Rate Attributes—Falling Transitions



Example 6-3 shows the slew-rate control attributes:

Example 6-3 Slew-Rate Control Attributes

```
pin(PAD) {
  is_pad : true;
  direction : output;
  output_voltage : GENERAL;
  slew_control : high;
  rise_current_slope_before_threshold : 0.18;
  rise_time_before_threshold : 0.8;
  rise_current_slope_after_threshold : -0.09;
  rise_time_after_threshold : 2.4;
  fall_current_slope_before_threshold : -0.14;
  fall_time_before_threshold : 0.55;
  fall_current_slope_after_threshold : 0.07;
  fall_time_after_threshold : 1.8;
  ...
}
```

6.6 Modeling Wire Load for Pads

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

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

You can define the wire_load group, which you want to use for wire load estimation, on the pad ring. Add a level of hierarchy by placing the pads in the top level and by placing all the core circuitry at a lower level.

A different model is defined for the core hierarchical level.

6.7 Programmable Driver Type Support in I/O Pad Cell Models

To support pull-up and pull-down circuit structures, the Liberty models for I/O pad cells support pull-up and pull-down driver information using the driver_type attribute with the values pull_up or pull_down.

Liberty syntax also supports conditional (programmable) pull-up and pull-down driver information for I/O pad cells. The programmable pin syntax has also been extended to other driver_type attribute values, such as bus_hold, open_drain, open_source, resistive, resistive_0, and resistive_1.

6.7.1 Syntax

The following syntax supports programmable driver types in I/O pad cell models. Unlike the nonprogrammable driver type support, the programmable driver type support allows you to specify more than one driver type within a pin.

```
pin (<pin_name>) { /* programmable driver type pin */
  …
  pull_up_function : "<function string>";
  pull_down_function : "<function string>";
  bus_hold_function : "<function string>";
  open_drain_function : "<function string>";
  open_source_function : "<function string>";
  resistive_function : "<function string>";
  resistive_0_function : "<function string>";
  resistive_1_function : "<function string>";
  …
}
```

6.7.2 Programmable Driver Type Functions

The functions in Table 6-4 have been introduced on top of (as an extension of) the existing driver_type attribute to support programmable pins. These driver type functions help model the programmable driver types. The same rules that apply to nonprogrammable driver types also apply to these functions.

Table 6-4 Programmable Driver Type Functions

Programmable Driver Type	Applicable on Pin Types
pull_up_function	Input, output and inout
pull_down_function	Input, output and inout
bus_hold_function	Inout
open_drain_function	Output and inout
open_source_function	Output and inout
resistive_function	Output and inout
resistive_0_function	Output and inout
resistive_1_function	Output and inout

With the exception of `pull_up_function` and `pull_down_function`, if any of the driver type functions in Table 6-4 is specified on an inout pin, it is used only for output pins.

The following rules apply to programmable driver type functions (as well as nonprogrammable driver types in I/O pad cell models):

- The attribute can be applied to pad cell only.
- Only the input and inout pin can be specified in the function string.
- The function string is a Boolean function of input pins.

The following rules apply to an inout pin:

- If `pull_up_function` or `pull_down_function` and `open_drain_function` are specified within the same inout pin, `pull_up_function` or `pull_down_function` is used for the input pins.
- If `bus_hold_function` is specified on an inout pin, it is used for input and output pins.

Example

The following example models a programmable driver type in an I/O pad cell.

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

time_unit : 1ns;
voltage_unit : 1V;
capacitive_load_unit (1.0, pf);
current_unit : 1mA;

cell(conditional_PU_PD) {
  dont_touch : true ;
  dont_use : true ;
  pad_cell : true ;
  pin(IO) {
    drive_current  : 1 ;
    min_capacitance : 0.001 ;
    min_transition : 0.0008 ;
    is_pad     : true ;
    direction    : inout ;
    max_capacitance : 30 ;
    max_fanout   : 2644 ;
    function    : "(A*ETM')+(TA*ETM)" ;
    three_state  : "(TEN*ETM')+(EN*ETM)" ;
         pull_up_function  : "(!P1 * !P2)" ;
    pull_down_function : "( P1 * P2)" ;
    capacitance   : 2.06649 ;
    timing() {
      related_pin : "IO A ETM TEN TA" ;
      cell_rise(scalar) {
        values("0" ) ;
      }
      rise_transition(scalar) {
        values("0" ) ;
      }
      cell_fall(scalar) {
        values("0" ) ;
      }
      fall_transition(scalar) {
        values("0" ) ;
      }
     }
    timing() {

      timing_type : three_state_disbale;
      related_pin : "EN ETM TEN" ;
      cell_rise(scalar) {
        values("0" ) ;
      }
```

```
      rise_transition(scalar) {
        values("0") ;
      }
      cell_fall(scalar) {
        values("0") ;
      }
      fall_transition(scalar) {
        values("0") ;
      }
    }

  pin(ZI) {
   direction : output;
     function    : "IO" ;
     timing() {
       related_pin : "IO" ;
       cell_rise(scalar) {
         values("0") ;
       }
       rise_transition(scalar) {
         values("0") ;
       }
       cell_fall(scalar) {
         values("0") ;
       }
       fall_transition(scalar) {
         values("0") ;
       }
     }
  }
  pin(A) {
   direction : input;
   capacitance : 1.0;
  }
  pin(EN) {
   direction : input;
   capacitance : 1.0;
  }
  pin(TA) {
   direction : input;
   capacitance : 1.0;
  }
  pin(TEN) {
   direction : input;
   capacitance : 1.0;
 }
  pin(ETM) {
   direction : input;
   capacitance : 1.0;
  }
  pin(P1) {
   direction : input;
   capacitance : 1.0;
  }
  pin(P2) {
   direction : input;
   capacitance : 1.0;
  }
 } /* End cell conditional_PU_PD */
 } /* End Library */
```

6.8 Pad Cell Examples

These are examples of input, clock, output, and bidirectional pad cells.

6.8.1 Input Pads

The input pad definition in Example 6-4 represents a standard input buffer, and Example 6-5 lists an input buffer with hysteresis.

Example 6-6 shows an input clock buffer.

Example 6-4 Input Buffer

```
library (example1) {
  date : "August 14, 2005" ;
  revision : 2005.05;
  ...
  time_unit : "1ns";
  voltage_unit : "1V";
  current_unit : "1uA";
  pulling_resistance_unit : "1kohm";
  capacitive_load_unit( 0.1,ff );
  ...
  define_cell_area(bond_pads,pad_slots);
  define_cell_area(driver_sites,pad_driver_sites);
  ...
  input_voltage(CMOS) {
    vil : 1.5;
    vih : 3.5;
    vimin : -0.3;
    vimax : VDD + 0.3;
  }
  ...
  /***** INPUT PAD*****/
  cell(INBUF) {
    area : 0.000000;
    pad_cell : true;
    bond_pads : 1;
    driver_sites : 1;
    pin(PAD ) {
      direction : input;
      is_pad : true;
      input_voltage : CMOS;
      capacitance : 2.500000;
      fanout_load : 0.000000;
    }
    pin(Y ) {
      direction : output;
      function : "PAD";
      timing() {
        intrinsic_fall : 2.952000
        intrinsic_rise : 3.075000
        fall_resistance : 0.500000
        rise_resistance : 0.500000
        related_pin :"PAD";
      }
    }
  }
```

Example 6-5 Input Buffer With Hysteresis

```
library (example1) {
  date : "August 14, 2005" ;
  revision : 2005.05;
  ...
  time_unit : "1ns";
  voltage_unit : "1V";
  current_unit : "1uA";
  pulling_resistance_unit : "1kohm";
  capacitive_load_unit( 0.1,ff );
  ...
  input_voltage(CMOS_SCHMITT) {
    vil : 1.0;
    vih : 4.0;
    vimin : -0.3;
    vimax : VDD + 0.3;
  }
  ...
```

```
  /*INPUT PAD WITH HYSTERESIS*/
  cell(INBUFH) {
    area : 0.000000;
    pad_cell : true;
    pin(PAD ) {
      direction : input;
      is_pad : true;
      hysteresis : true;
      input_voltage : CMOS_SCHMITT;
      capacitance : 2.500000;
      fanout_load : 0.000000;
    }
    pin(Y ) {
      direction : output;
      function : "PAD";
      timing() {
        intrinsic_fall : 2.952000
        intrinsic_rise : 3.075000
        fall_resistance : 0.500000
        rise_resistance : 0.500000
        related_pin :"PAD";
      }
    }
  }
```

Example 6-6 Input Clock Buffer

```
library (example1) {
  date : "August 12, 2005" ;
  revision : 2005.05;
  ...
  time_unit : "1ns";
  voltage_unit : "1V";
  current_unit : "1uA";
  pulling_resistance_unit : "1kohm";
  capacitive_load_unit( 0.1,ff );
  ...
  input_voltage(CMOS) {
    vil : 1.5;
    vih : 3.5;
    vimin : -0.3;
    vimax : VDD + 0.3;
  }
  ...
  /***** CLOCK INPUT BUFFER *****/
  cell(CLKBUF) {
    area : 0.000000;
    pad_cell : true;
    pad_type : clock;
    pin(PAD ) {
      direction : input;
      is_pad : true;
      input_voltage : CMOS;
      capacitance : 2.500000;
      fanout_load : 0.000000;
    }
    pin(Y ) {
      direction : output;
      function : "PAD";
      max_fanout : 2000.000000;
      timing() {
        intrinsic_fall : 6.900000
        intrinsic_rise : 5.700000
        fall_resistance : 0.010238
        rise_resistance : 0.009921
        related_pin :"PAD";
      }
    }
```

```
}
```

6.8.2 Output Pads

The output pad definition in Example 6-7 represents a standard output buffer.

Example 6-7 Output Buffer

```
library (example1) {
  date : "August 12, 2005" ;
  revision : 2005.05;
  ...
  time_unit : "1ns";
  voltage_unit : "1V";
  current_unit : "1uA";
  pulling_resistance_unit : "1kohm";
  capacitive_load_unit( 0.1,ff );
  ...
  output_voltage(GENERAL) {
    vol : 0.4;
    voh : 2.4;
    vomin : -0.3;
    vomax : VDD + 0.3;
  }
  /***** OUTPUT PAD *****/
  cell(OUTBUF) {
    area : 0.000000;
    pad_cell : true;
    pin(D ) {
      direction : input;
      capacitance : 1.800000;
    }
    pin(PAD ) {
      direction : output;
      is_pad : true;
      drive_current : 2.0;
      output_voltage : GENERAL;
      function : "D";
      timing() {
        intrinsic_fall : 9.348001
        intrinsic_rise : 8.487000
        fall_resistance : 0.186960
        rise_resistance : 0.169740
        related_pin :"D";
      }
    }
  }
```

6.8.3 Bidirectional Pad

Example 6-8 shows a bidirectional pad cell with three-state enable.

Example 6-8 Bidirectional Pad

```
library (example1) {
  date : "August 12, 2005" ;
  revision : 2005.05;
  ...
  time_unit : "1ns";
  voltage_unit : "1V";
  current_unit : "1uA";
  pulling_resistance_unit : "1kohm";
  capacitive_load_unit( 0.1,ff );
  ...
  output_voltage(GENERAL) {
    vol : 0.4;
```

```
  voh : 2.4;
  vomin : -0.3;
  vomax : VDD + 0.3;
}
/***** BIDIRECTIONAL PAD *****/
cell(BIBUF) {
  area : 0.000000;
  pad_cell : true;
  pin(E D ) {
    direction : input;
    capacitance : 1.800000;
  }
  pin(Y ) {
    direction : output;
    function : "PAD";
    driver_type : "open_source pull_up";
    pulling_resistance : 10000;
    timing() {
      intrinsic_fall : 2.952000
      intrinsic_rise : 3.075000
      fall_resistance : 0.500000
      rise_resistance : 0.500000
      related_pin :"PAD";
    }
  }
  pin(PAD ) {
    direction : inout;
    is_pad : true;
    drive_current : 2.0;
    output_voltage : GENERAL;
    input_voltage : CMOS;
    function : "D";
    three_state : "E";
    timing() {
      intrinsic_fall : 19.065001
      intrinsic_rise : 17.466000
      fall_resistance : 0.381300
      rise_resistance : 0.346860
      related_pin :"E";
    }
    timing() {
      intrinsic_fall : 9.348001
      intrinsic_rise : 8.487000
      fall_resistance : 0.186960
      rise_resistance : 0.169740
      related_pin :"D";
    }
  }
}
```

6.8.4 Cell with contention_condition and x_function

Example 6-9 shows a cell with the `contention_condition` attribute, which specifies contention-causing conditions and the `x_function` attribute, which describes the X behavior of the pin. (See "contention_condition Attribute" and the "x_function Attribute" description in "Describing Clock Pin Functions " for more information about these attributes.)

Example 6-9 Cell With contention_condition and x_function Attributes

```
default_intrinsic_fall : 0.1;
default_inout_pin_fall_res : 0.1;
default_fanout_load : 0.1;
default_intrinsic_rise : 0.1;
default_slope_rise : 0.1;
default_output_pin_fall_res : 0.1;
default_inout_pin_cap : 0.1;
default_input_pin_cap : 0.1;
default_slope_fall : 0.1;
default_inout_pin_rise_res : 0.1;
```

```
 default_output_pin_cap : 0.1;
 default_output_pin_rise_res : 0.1;

 capacitive_load_unit(1, pf);

pulling_resistance_unit : 1ohm;

voltage_unit : 1V;
current_unit : 1mA;
time_unit : 1ps;

cell (cell_a) {
 area : 1;
 contention_condition : "!ap & an";

   pin (ap, an) {
  direction : input;
  capacitance : 1;
 }
 pin (io) {
  direction : output;
  function : "!ap & !an";
  three_state : "ap & !an";
  x_function : "!ap & an";
  timing() {
   related_pin : "ap an";
   timing_type : three_state_disable;
   intrinsic_rise : 0.1;
   intrinsic_fall : 0.1;
  }
  timing() {
   related_pin : "ap an";
   timing_type : three_state_enable;
   intrinsic_rise : 0.1;
   intrinsic_fall : 0.1;
  }
 }
 pin (z) {
  direction : output;
  function : "!ap & !an";
  x_function : "!ap & an | ap & !an";
 }
 }
}
```

7. Defining Test Cells

A test cell contains information that supports full-scan or a partial-scan methodology.

You must add test-specific details of scannable cells to your technology libraries. For example, you must identify scannable flip-flops and latches and select the types of unscannable cells they replace for a given scan methodology. To do this, you must understand the following concepts described in this chapter:

- Describing a Scan Cell
- Describing a Multibit Scan Cell
- Scan Cell Modeling Examples

7.1 Describing a Scan Cell

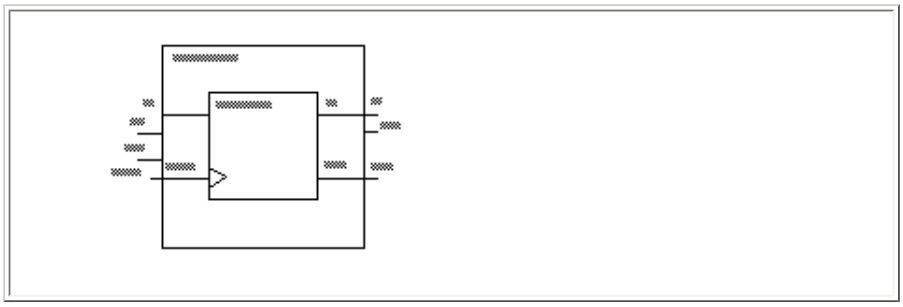
To specify a cell as a scan cell, add the `test_cell` group to the cell description.

Only the nontest mode function of a scan cell is modeled in the `test_cell` group. The nontest operation is described by its `ff`, `ff_bank`, `latch`, or `latch_bank` declaration and `pin` function attributes.

7.1.1 test_cell Group

The `test_cell` group defines only the nontest mode function of a scan cell. Figure 7-1 illustrates the relationship between a `test_cell` group and the scan cell in which it is defined.

Figure 7-1 Scan Cell With test_cell Group



There are two important points to remember when defining a scan cell such as the one Figure 7-1 shows:

- Pin names in the scan cell and the test cell must match.
- The scan cell and the test cell must contain the same functional outputs.

Following is the syntax of a `test_cell` group that contains pins:

Syntax

```
library (lib_name) {    cell (cell_name) {
        test_cell () {
            ... test cell description ...
            pin ( name ) {
                ... pin description ...
            }
            pin ( name ) {
                ... pin description ...
        }
    }
}
```

You do not need to give the `test_cell` group a name, because the test cell takes the cell name of the cell being defined. The `test_cell` group can contain `ff`, `ff_bank`, `latch`, or `latch_bank` group statements and `pin` groups.

Pins in the test_cell Group

Both test pins and nontest pins can appear in `pin` groups within a `test_cell` group. These groups are like `pin` groups in a `cell` group but must adhere to the following rules:

- Each pin defined in the `cell` group must have a corresponding pin defined at the `test_cell` group level with the same name.
- The `pin` group can contain only `direction`, `function`, `test_output_only`, and `signal_type` attribute statements. The `pin` group cannot contain timing, capacitance, fanout, or load information.
- The `function` attributes can reflect only the nontest behavior of the cell.
- Input pins must be referenced in an `ff`, `ff_bank`, `latch`, or `latch_bank` statement or have a `signal_type` attribute assigned to them.
- An output pin must have either a `function` attribute, a `signal_type` attribute, or both.

7.1.2 test_output_only Attribute

This attribute indicates that an output port is set for test only (as opposed to function only, or function and test).

Syntax

```
test_output_only : true | false ;
```

When you use statetable format to describe the functionality of a scan cell, you must declare the output pin as set for test only by setting the `test_output_only` attribute to `true`.

Example

```
test_output_only : true ;
```

signal_type Simple Attribute

In a `test_cell` group, `signal_type` identifies the type of test pin.

Syntax

```
signal_type : test_scan_in | test_scan_in_inverted
|
        test_scan_out | test_scan_out_inverted |
        test_scan_enable |
        test_scan_enable_inverted |
        test_scan_clock | test_scan_clock_a |
        test_scan_clock_b | test_clock ;
```

test_scan_in

Identifies the scan-in pin of a scan cell. The scanned value is the same as the value present on the scan-in pin. All scan cells must have a pin with either the `test_scan_in` or the `test_scan_in_inverted` attribute.

test_scan_in_inverted

Identifies the scan-in pin of a scan cell as having inverted polarity. The scanned value is the inverse of the value present on the scan-in pin.

For multiplexed flip-flop scan cells, the polarity of the scan-in pin is inferred from the latch or ff declaration of the cell itself. For other types of scan cells, clocked-scan, LSSD, and multiplexed flip-flop latches, it is not possible to give the ff or latch declaration of the entire scan cell. For these cases, you can use the `test_scan_in_inverted` attribute in the cell where the scan-in pin appears in the latch or ff declarations for the entire cell.

test_scan_out

Identifies the scan-out pin of a scan cell. The value present on the scan-out pin is the same as the scanned value. All scan cells must have a pin with either a `test_scan_out` or a `test_scan_out_inverted` attribute.

The scan-out pin corresponds to the output of the slave latch in the LSSD methodologies.

test_scan_out_inverted

Identifies the scan-out pin of a test cell as having inverted polarity. The value on this pin is the inverse of the scanned value.

test_scan_enable

Identifies the pin of a scan cell that, when high, indicates that the cell is configured in scan-shift mode. In this mode, the clock transfers data from the scan-in input to the scan-out input.

test_scan_enable_inverted

Identifies the pin of a scan cell that, when low, indicates that the cell is configured in scan-shift mode. In this mode, the clock transfers data from the scan-in input to the scan-out input.

test_scan_clock

Identifies the test scan clock for the clocked-scan methodology. The signal is assumed to be edge-sensitive. The active edge transfers data from the scan-in pin to the scan-out pin of a cell. The sense of this clock is determined by the sense of the associated timing arcs.

test_scan_clock_a

Identifies the a clock pin in a cell that supports a single-latch LSSD, double-latch LSSD, clocked LSSD, or auxiliary clock LSSD methodology. When the a clock is at the active level, the master latch of the scan cell can accept scan-in data. The sense of this clock is determined by the sense of the associated timing arcs.

test_scan_clock_b

Identifies the b clock pin in a cell that supports the single-latch LSSD, clocked LSSD, or auxiliary clock LSSD methodology. When the b clock is at the active level, the slave latch of the scan-cell can accept the value of the master latch. The sense of this clock is determined by the sense of the associated timing arcs.

test_clock

Identifies an edge-sensitive clock pin that controls the capturing of data to fill scan-in test mode in the auxiliary clock LSSD methodology.

If an input pin is used in both test and nontest modes (such as the clock input in the multiplexed flip-flop methodology), do not include a `signal_type` statement for that pin in the `test_cell` pin definition.

If an input pin is used only in test mode and does not exist on the cell that it will scan and replace, you must include a `signal_type` statement for that pin in the `test_cell` pin definition.

If an output pin is used in nontest mode, it needs a `function` statement. The `signal_type` statement is used to identify an output pin as a scan-out pin. In a `test_cell` group, the `pin` group for an output pin can contain a `function` statement, a `signal_type` attribute, or both.

Note:

You do not have to define a `function` or `signal_type` attribute in the `pin` group if the pin is defined in a previous `test_cell` group for the same cell.

Example

```
signal_type : test_scan_in ;
```

7.2 Describing a Multibit Scan Cell

You can model a multibit scan cell, such as a 4-bit flip-flop, using the `ff_bank` or `latch_bank` group in the `test_cell` description. Figure 7-2 illustrates a 2-bit piece of a 4-bit D flip-flop with scan and enable.

Figure 7-2 Multibit Scan Cell With test_cell Group

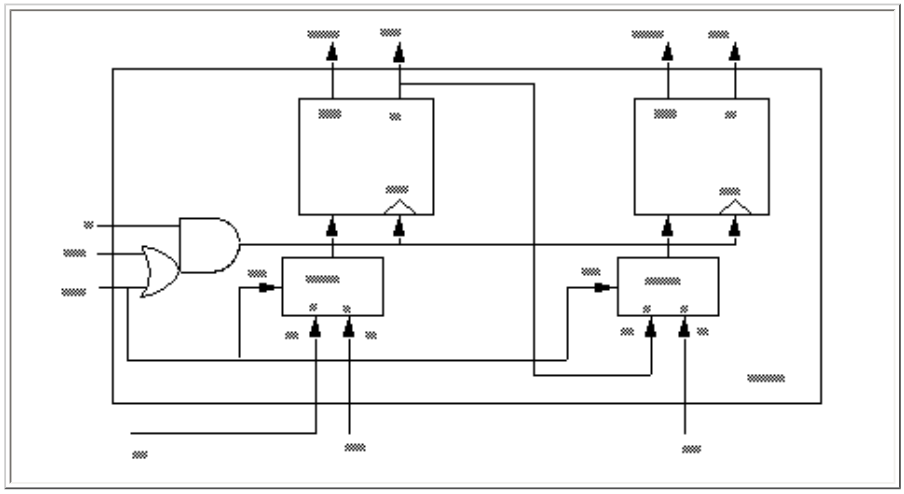
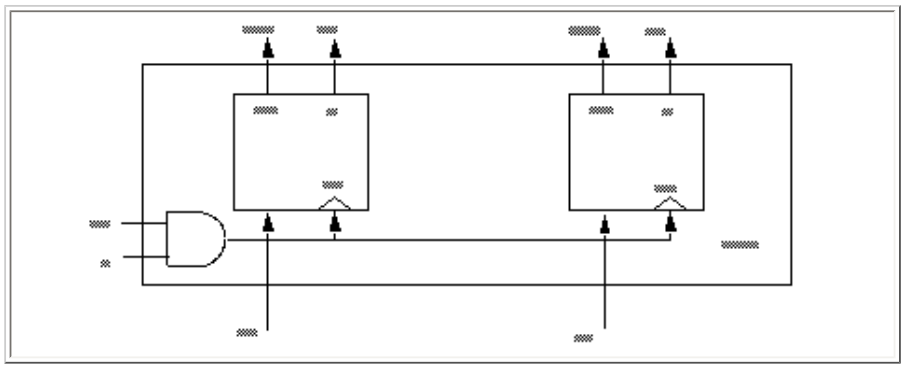


Figure 7-3 shows the `test_cell` group (the nontest mode) of the cell in Figure 7-2.

Figure 7-3 test_cell Group of Multibit Scan Cell



To create a multibit scan cell, use `statetable` in the full cell description, as shown in Example 7-1.

Example 7-1 Statetable in Full-Cell Description

```
cell (FSX2) { ...
 bundle (D) {
    members (D0, D1);
    ...
  }
 bundle (Q) {
    members (Q0, Q1);
    ...
    pin (Q0) {
      input_map : "D0 T SI SM";
      ...
    }
    pin (Q1) {
      input_map : "D1 T Q0 SM";
      ...
    }
 }
 pin (SI) {.
    ..
  }
  pin (SM) {
    ...
  }
  pin (T) {
    ...
  }pin (EN) {
    ...
```

```
  }
 statetable ( "D CP TI TE EN", "Q QN") {
       /*D  CP TI  TE EN Q QN Q+  QN+ */
    table : "-  ~R -  -  -: - -: N  N,\
        -  -  -  L  L: - -: N  N,\
        -  R  H/L H  -: - -: H/L L/H,\
       H/L R  -  L  H: - -: H/L L/H "
 }
}
```

In Example 7-1, `statetable` describes the behavior of a single slice of the cell. The `input_map` statements on pin Q0 and pin Q1 indicate the interconnection of the different slices of the cell—for example, the fact that Q of bit 0 goes to TI of bit 1.

Example 7-2 shows the test_cell description for multibit pins, using the `ff_bank` and `bundle` group attributes.

Example 7-2 test_cell Description for Multibit Scan Cells

```
cell (FSX2) {...
  bundle (D) {
    members (D0, D1);
    ...
  }
  bundle (Q) {
    members (Q0, Q1);
      ...
  }
  pin(SI){
    ...
  }
  pin(SM){
    ...
  }
  pin(T){
    ...
  }
  pin(EN) {
    ...
  }
  statetable (...) {
    ...
  }
  test_cell {
    ff_bank(IQ,IQN,2){
      next_state : D ;
      clocked_on: "T & EN" ;
    }
    bundle (D) {
      members (D0, D1) ;
      ...
    }
    bundle (Q) {
      members (Q0, Q1) ;
      function : IQ ;
      signal_type : test_scan_out ;
    }
    bundle(QC) {
      members (QC0, QC1);
       function : IQN ;
      signal_type : test_scan_out_inverted ;
    }
    pin (SI) {
      signal_type : test_scan_in;
      ...
    }
    pin (SM) {
      signal_type : test_scan_enable ;
      ...
    }
```

```
    pin (T) {
      ...
    }
  }
}
```

For a complete description of multibit scan cells, see Example 7-4.

7.3 Scan Cell Modeling Examples

This section contains modeling examples for these test cells:

- Simple multiplexed D flip-flop
- Multibit cells with multiplexed D flip-flop and enable
- LSSD (level-sensitive scan design) scan cell
- Clocked-scan test cell
- Scan D flip-flop with auxiliary clock

Each example contains a complete cell description.

7.3.1 Simple Multiplexed D Flip-Flop

Example 7-3 shows how to model a simple multiplexed D flip-flop test cell.

Example 7-3 Simple Multiplexed D Flip-Flop Scan Cell

```
cell(SDFF1) {
  area : 9;
  pin(D) {
    direction : input;
    capacitance : 1;
    timing() {...}
  }
  pin(CP) {
    direction : input;
    capacitance : 1;
    timing() {...}
  }
  pin(TI) {
    direction : input;
    capacitance : 1;
    timing() {...}
  }
  pin(TE) {
  direction : input;
  capacitance : 2;
  timing() {...}
  }
  ff(IQ,IQN) {
  /* model full behavior (if possible): */
  next_state : "D TE' + TI TE ";
  clocked_on : "CP";
  }
  pin(Q) {
  direction : output;
  function : "IQ";
  timing() {...}
  }
  pin(QN) {
  direction : output;
  function : "IQN";
  timing() {...}
  }
  test_cell() {
```

```
  pin(D) {
      direction : input
    }
    pin(CP){
      direction : input
    }
    pin(TI) {
      direction : input;
      signal_type : test_scan_in;
    }
    pin(TE) {
      direction : input;
      signal_type : test_scan_enable;
    }
    ff(IQ,IQN) {
      /* just model non-test operation behavior */
      next_state : "D";
      clocked_on : "CP";
  }
  pin(Q) {
      direction : output;
      function : "IQ";
      signal_type : test_scan_out;
  }
    pin(QN) {
      direction : output;
      function : "IQN";
      signal_type : test_scan_out_inverted;
    }
  }
}
```

7.3.2 Multibit Cells With Multiplexed D Flip-Flop and Enable

Example 7-4 contains a complete description of multibit scan cells.

Example 7-4 Multibit Scan Cells With Multiplexed D Flip-Flop and Enable

```
library(banktest) {
...
default_inout_pin_cap    : 1.0;
default_inout_pin_fall_res  : 0.0;
default_inout_pin_rise_res  : 0.0;
default_input_pin_cap    : 1.0;
default_intrinsic_fall    : 1.0;
default_intrinsic_rise    : 1.0;
default_output_pin_cap    : 0.0;
default_output_pin_fall_res : 0.0;
default_output_pin_rise_res : 0.0;
default_slope_fall     : 0.0;
default_slope_rise     : 0.0;
default_fanout_load     : 1.0;
time_unit : "1ns";
voltage_unit : "1V";
current_unit : "1uA";
pulling_resistance_unit : "1kohm";
capacitive_load_unit (0.1,ff);
type (bus4) {
 base_type : array;
 data_type : bit;
 bit_width : 4;
 bit_from : 0;
 bit_to  : 3;
}
cell(FDSX4) {
 area : 36;
 bus(D) {
  bus_type : bus4;
  direction : input;
```

```
 capacitance : 1;
 timing() {
  timing_type : setup_rising;
  intrinsic_rise : 1.3; intrinsic_fall : 1.3;
  related_pin : "CP";
 }
  timing() {
  timing_type : hold_rising;
  intrinsic_rise : 0.3; intrinsic_fall : 0.3;
  related_pin : "CP";
 }
}
pin(CP) {
 direction : input;
 capacitance : 1;
}
pin(TI) {
 direction : input;
 capacitance : 1;
 timing() {
  timing_type : setup_rising;
  intrinsic_rise : 1.3; intrinsic_fall : 1.3;
  related_pin : "CP";
 }
 timing() {
  timing_type : hold_rising;
  intrinsic_rise : 0.3; intrinsic_fall : 0.3;
  related_pin : "CP";
 }
}
pin(TE) {
 direction : input;
 capacitance : 2;
 timing() {
  timing_type : setup_rising;
  intrinsic_rise : 1.3; intrinsic_fall : 1.3;
  related_pin : "CP";
 }
 timing() {
  timing_type : hold_rising;
  intrinsic_rise : 0.3; intrinsic_fall : 0.3;
  related_pin : "CP";
 }
}
statetable ( " D  CP TI  TE ", " Q  QN") {
 table : " - ~R  -  -   : -  - : N  N,  \
       -  R H/L  H   : -  - : H/L L/H, \
      H/L R  -  L   : -  - : H/L L/H" ;
}
 bus(Q) {
 bus_type : bus4;
 direction : output;
 inverted_output : false;
 internal_node : "Q";
 timing() {
  timing_type : rising_edge;
  intrinsic_rise : 1.09; intrinsic_fall : 1.37;
  rise_resistance : 0.1458; fall_resistance : 0.0523;
  related_pin : "CP";
 }
 pin(Q[0]) {
  input_map : "D[0] CP TI  TE";
 }
 pin(Q[1]) {
  input_map : "D[1] CP Q[0] TE";
 }
 pin(Q[2]) {
  input_map : "D[2] CP Q[1] TE";
 }
 pin(Q[3]) {
  input_map : "D[3] CP Q[2] TE";
```

```
 }
}
bus(QN) {
 bus_type : bus4;
 direction : output;
 inverted_output : true;
 internal_node : "QN";
 timing() {
  timing_type : rising_edge;
  intrinsic_rise : 1.59; intrinsic_fall : 1.57;
  rise_resistance : 0.1458; fall_resistance : 0.0523;
  related_pin : "CP";
 }
 pin(QN[0]) {
  input_map : "D[0] CP TI  TE";
 }
 pin(QN[1]) {
  input_map : "D[1] CP Q[0] TE";
 }
 pin(QN[2]) {
  input_map : "D[2] CP Q[1] TE";
 }
 pin(QN[3]) {
  input_map : "D[3] CP Q[2] TE";
 }
}
test_cell() {
 bus (D) {
  bus_type : bus4;
  direction : input;
 }
 pin(CP) {
  direction : input;
 }
 pin(TI) {
  direction : input;
  signal_type : "test_scan_in";
 }
 pin(TE) {
  direction : input;
  signal_type : "test_scan_enable";
 }
 ff_bank("IQ","IQN", 4) {
  next_state : "D";
  clocked_on : "CP";
 }
 bus(Q) {
  bus_type : bus4;
  direction : output;
  function : "IQ";
  signal_type : "test_scan_out";
 }
 bus(QN) {
  bus_type : bus4;
  direction : output;
  function : "IQN";
  signal_type : "test_scan_out_inverted";
 }
}
}
cell(SCAN2) {
 area : 18;
 bundle(D) {
  members(D0, D1);
  direction : input;
  capacitance : 1;
  timing() {
   timing_type : setup_rising;
   intrinsic_rise : 1.3; intrinsic_fall : 1.3;
   related_pin : "T";
  }
```

```
 timing() {
  timing_type : hold_rising;
  intrinsic_rise : 0.3; intrinsic_fall : 0.3;
  related_pin : "T";
 }
}
pin(T) {
 direction : input;
 capacitance : 1;
}
pin(EN) {
 direction : input;
 capacitance : 2;
 timing() {
  timing_type : setup_rising;
  intrinsic_rise : 1.3; intrinsic_fall : 1.3;
  related_pin : "T";
 }
 timing() {
  timing_type : hold_rising;
  intrinsic_rise : 0.3; intrinsic_fall : 0.3;
  related_pin : "T";
 }
}
pin(SI) {
 direction : input;
 capacitance : 1;
 timing() {
  timing_type : setup_rising;
  intrinsic_rise : 1.3; intrinsic_fall : 1.3;
  related_pin : "T";
 }
 timing() {
  timing_type : hold_rising;
  intrinsic_rise : 0.3; intrinsic_fall : 0.3;
  related_pin : "T";
 }
}
pin(SM) {
 direction : input;
 capacitance : 2;
 timing() {
  timing_type : setup_rising;
  intrinsic_rise : 1.3; intrinsic_fall : 1.3;
  related_pin : "T";
 }
 timing() {
  timing_type : hold_rising;
  intrinsic_rise : 0.3; intrinsic_fall : 0.3;
  related_pin : "T";
 }
}
statetable ( "T  D  EN  SI SM",    "Q  QN") {
 table : "~R  -  -   -  -  : -  - : N   N , \
       -  -  L   -  L  : -  - : N   N , \
       R H/L H   -  L  : -  - : H/L L/H, \
       R  -  -  H/L H  : -  - : H/L L/H";
}
bundle(Q) {
 members(Q0, Q1);
 direction : output;
 inverted_output : false;
 internal_node : "Q";
 timing() {
  timing_type : rising_edge;
  intrinsic_rise : 1.09; intrinsic_fall : 1.37;
  rise_resistance : 0.1458; fall_resistance : 0.0523;
  related_pin : "T";
 }
 pin(Q0) {
  input_map : "T D0 EN SI SM";
```

```
  }
  pin(Q1) {
   input_map : "T D1 EN Q0 SM";
  }
 }
 bundle(QN) {
  members(Q0N, Q1N);
  direction : output;
  inverted_output : true;
  internal_node : "QN";
  timing() {
   timing_type : rising_edge;
   intrinsic_rise : 1.59; intrinsic_fall : 1.57;
   rise_resistance : 0.1458; fall_resistance : 0.0523;
   related_pin : "T";
  }
  pin(Q0N) {
   input_map : "T D0 EN SI SM";
  }
  pin(Q1N) {
   input_map : "T D1 EN Q0 SM";
  }
 }
  test_cell() {
  bundle (D) {
   members(D0, D1);
   direction : input;
  }
  pin(T) {
   direction : input;
  }
  pin(EN) {
   direction : input;
  }
  pin(SI) {
   direction : input;
   signal_type : "test_scan_in";
  }
  pin(SM) {
   direction : input;
   signal_type : "test_scan_enable";
  }
  ff_bank("IQ","IQN", 2) {
   next_state : "D";
   clocked_on : "T EN";
  }
  bundle(Q) {
   members(Q0, Q1);
   direction : output;
   function : "IQ";
   signal_type : "test_scan_out";
  }
  bundle(QN) {
   members(Q0N, Q1N);
   direction : output;
   function : "IQN";
   signal_type : "test_scan_out_inverted";
  }
 }
}
```

7.3.3 LSSD Scan Cell

Example 7-5 shows how to model an LSSD element. For latch-based designs, this form of scan cell has two `test_cell` groups so that it can be used in either single-latch or double-latch mode.

Example 7-5 LSSD Scan Cell

```
cell(LSSD) {
 area : 12;
 pin(D) {
  direction : input;
  capacitance : 1;
  timing() {
   timing_type : setup_falling;
   intrinsic_rise : 1.0;
   intrinsic_fall : 1.0;
   related_pin : "MCLK";
  }
  timing() {
   timing_type : hold_falling;
   intrinsic_rise : 1.0;
   intrinsic_fall : 1.0;
   related_pin : "MCLK";
  }
 }
 pin(SI) {
  direction : input;
  capacitance : 1;
  prefer_tied : "0";
  timing() {
   timing_type : setup_falling;
   intrinsic_rise : 1.0;
   intrinsic_fall : 1.0;
   related_pin : "ACLK";
  }
  timing() {
   timing_type : hold_falling;
   intrinsic_rise : 1.0;
   intrinsic_fall : 1.0;
   related_pin : "ACLK";
  }
 }
 pin(MCLK,ACLK,SCLK) {
  direction : input;
  capacitance : 1;
 }
 pin(Q1) {
  direction : output;
  internal_node : "Q1";
  timing() {
   timing_type : rising_edge;
   intrinsic_rise : 1.0;
   intrinsic_fall : 1.0;
   rise_resistance : 0.1;
   fall_resistance : 0.1;
   related_pin : "MCLK";
  }
  timing() {
   timing_type : rising_edge;
   intrinsic_rise : 1.0;
   intrinsic_fall : 1.0;
   rise_resistance : 0.1;
   fall_resistance : 0.1;
   related_pin : "ACLK";
  }
  timing() {
   intrinsic_rise : 1.0;
   intrinsic_fall : 1.0;
   rise_resistance : 0.1;
   fall_resistance : 0.1;
   related_pin : "D";
  }
  timing() {
   intrinsic_rise : 1.0;
   intrinsic_fall : 1.0;
   rise_resistance : 0.1;
   fall_resistance : 0.1;
   related_pin : "SI";
```

```
 }
}
pin(Q1N) {
 direction : output;
 state_function : "Q1′";
 timing() {
  timing_type : rising_edge;
  intrinsic_rise : 1.0;
  intrinsic_fall : 1.0;
  rise_resistance : 0.1;
  fall_resistance : 0.1;
  related_pin : "MCLK";
 }
 timing() {
  timing_type : rising_edge;
  intrinsic_rise : 1.0;
  intrinsic_fall : 1.0;
  rise_resistance : 0.1;
  fall_resistance : 0.1;
  related_pin : "ACLK";
 }
 timing() {
  intrinsic_rise : 1.0;
  intrinsic_fall : 1.0;
 rise_resistance : 0.1;
  fall_resistance : 0.1;
  related_pin : "D";
 }
 timing() {
  intrinsic_rise : 1.0;
  intrinsic_fall : 1.0;
  rise_resistance : 0.1;
  fall_resistance : 0.1;
  related_pin : "SI";
 }
}
pin(Q2) {
 direction : output;
 internal_node : "Q2";
 timing() {
  timing_type : rising_edge;
  intrinsic_rise : 1.0;
  intrinsic_fall : 1.0;
  rise_resistance : 0.1;
  fall_resistance : 0.1;
  related_pin : "SCLK";
 }
}
pin(Q2N) {
 direction : output;
 state_function : "Q2′";
 timing() {
  timing_type : rising_edge;
  intrinsic_rise : 1.0;
  intrinsic_fall : 1.0;
  rise_resistance : 0.1;
  fall_resistance : 0.1;
  related_pin : "SCLK";
 }
}
statetable("MCLK D  ACLK SCLK SI",    "Q1  Q2") {
 table :  " L -  L  -   - : -  - : N  - , \
      H L/H L  -   - : -  - : L/H  - , \
      L -  H  -  L/H : -  - : L/H  - , \
      H -  H  -   - : -  - : X  - , \
      - -  - L   - : -  - : -  N , \
      - -  - H   - : L/H - : -  L/H";
 }
test_cell() { /* for DLATCH */
 pin(D,MCLK) {
  direction : input;
```

```
 }
 pin(SI) {
  direction : input;
   signal_type : "test_scan_in";
 }
 pin(ACLK) {
  direction : input;
  signal_type : "test_scan_clock_a";
 }
 pin(SCLK) {
  direction : input;
  signal_type : "test_scan_clock_b";
 }
 latch ("IQ","IQN") {
  data_in : "D";
  enable : "MCLK";
 }
  pin(Q1) {
   direction : output;
   function : "IQ";
  }
  pin(Q1N) {
   direction : output;
   function : "IQN";
  }
  pin(Q2) {
   direction : output;
   signal_type : "test_scan_out";
  }
  pin(Q2N) {
   direction : output;
   signal_type : "test_scan_out_inverted";
  }
 }
 test_cell() { /* for MSFF1 */
  pin(D,MCLK,SCLK) {
   direction : input;
  }
  pin(SI) {
   direction : input;
   signal_type : "test_scan_in";
  }
  pin(ACLK) {
   direction : input;
   signal_type : "test_scan_clock_a";
  }
  ff ("IQ","IQN") {
   next_state : "D";
   clocked_on : "MCLK";
   clocked_on_also : "SCLK";
  }
  pin(Q1,Q1N) {
   direction : output;
  }
  pin(Q2) {
   direction : output;
   function : "IQ";
   signal_type : "test_scan_out";
  }
  pin(Q2N) {
   direction : output;
   function : "IQN";
   signal_type : "test_scan_out_inverted";
  }
 }
}
```

7.3.4 Clocked-Scan Test Cell

Example 7-6 shows how to model a level-sensitive latch with separate scan clocking. This example shows the scan cell used in clocked-scan implementation.

Example 7-6 Clocked-Scan Test Cell

```
cell(SC_DLATCH) {
 area : 12;
 pin(D) {
  direction : input;
  capacitance : 1;
  timing() {
   timing_type : setup_falling;
   intrinsic_rise : 1.0;
   intrinsic_fall : 1.0;
   related_pin : "G";
  }
  timing() {
   timing_type : hold_falling;
   intrinsic_rise : 1.0;
   intrinsic_fall : 1.0;
   related_pin : "G";
  }
 }
 pin(SI) {
  direction : input;
  capacitance : 1;
  prefer_tied : "0";
  timing() {
   timing_type : setup_rising;
   intrinsic_rise : 1.0;
   intrinsic_fall : 1.0;
   related_pin : "ScanClock";
  }
  timing() {
   timing_type : hold_rising;
   intrinsic_rise : 1.0;
   intrinsic_fall : 1.0;
   related_pin : "ScanClock";
  }
 }
 pin(G,ScanClock) {
  direction : input;
  capacitance : 1;
 }
 statetable( "D  SI  G ScanClock",   " Q  QN" ) {
  table :  "L/H -  H   L  : -  - : L/H  H/L,\
       -  L/H L   R  : -  - : L/H  H/L,\
       -  -   L  ~R  : -  - : N    N";
 }
 pin(Q) {
  direction : output;
  internal_node : "Q";
  timing() {
   timing_type : rising_edge;
   intrinsic_rise : 1.0;
   intrinsic_fall : 1.0;
   rise_resistance : 0.1;
   fall_resistance : 0.1;
   related_pin : "G";
  }
  timing() {
   intrinsic_rise : 1.0;
   intrinsic_fall : 1.0;
   rise_resistance : 0.1;
   fall_resistance : 0.1;
   related_pin : "D";
  }
  timing() {
   timing_type : rising_edge;
   intrinsic_rise : 1.0;
```

```
  intrinsic_fall : 1.0;
  rise_resistance : 0.1;
  fall_resistance : 0.1;
  related_pin : "ScanClock";
 }
}
pin(QN) {
 direction : output;
 internal_node : "QN";
 timing() {
  timing_type : rising_edge;
  intrinsic_rise : 1.0;
  intrinsic_fall : 1.0;
  rise_resistance : 0.1;
  fall_resistance : 0.1;
  related_pin : "G";
 }
 timing() {
  intrinsic_rise : 1.0;
  intrinsic_fall : 1.0;
  rise_resistance : 0.1;
  fall_resistance : 0.1;
  related_pin : "D";
 }
 timing() {
  timing_type : rising_edge;
  intrinsic_rise : 1.0;
  intrinsic_fall : 1.0;
  rise_resistance : 0.1;
  fall_resistance : 0.1;
  related_pin : "ScanClock";
 timing() {
  timing_type : rising_edge;
  intrinsic_rise : 1.0;
  intrinsic_fall : 1.0;
  rise_resistance : 0.1;
  fall_resistance : 0.1;
  related_pin : "ScanClock";
 }
}
test_cell() {
 pin(D,G) {
  direction : input;
 }
 pin(SI) {
  direction : input;
  signal_type : "test_scan_in";
 }
 pin(ScanClock) {
  direction : input;
  signal_type : "test_scan_clock";
 }
 pin(SE) {
  direction : input;
  signal_type : "test_scan_enable";
 }
 latch ("IQ","IQN") {
  data_in : "D";
  enable : "G";
 }
 pin(Q) {
  direction : output;
  function : "IQ";
  signal_type : "test_scan_out";
 }
 pin(QN) {
  direction : output;
  function : "IQN";
  signal_type : "test_scan_out_inverted";
 }
}
```

```
}
```

7.3.5 Scan D Flip-Flop With Auxiliary Clock

Example 7-7 shows how to model a scan D flip-flop with one input and an auxiliary clock.

Example 7-7 Scan D Flip-Flop With Auxiliary Clock

```
cell(AUX_DFF1) {
 area : 12;
 pin(D) {
  direction : input;
  capacitance : 1;
  timing() {
   timing_type : setup_rising;
   intrinsic_rise : 1.0;
   intrinsic_fall : 1.0;
   related_pin : "CK";
  }
  timing() {
   timing_type : hold_rising;
   intrinsic_rise : 1.0;
   intrinsic_fall : 1.0;
   related_pin : "CK";
  }
 timing() {
   timing_type : setup_rising;
   intrinsic_rise : 1.0;
   intrinsic_fall : 1.0;
   related_pin : "IH";
  }
  timing() {
   timing_type : hold_rising;
   intrinsic_rise : 1.0;
   intrinsic_fall : 1.0;
   related_pin : "IH";
  }
 }
 pin(CK,IH,A,B) {
  direction : input;
  capacitance : 1;
 }
 pin(SI) {
  direction : input;
  capacitance : 1;
  prefer_tied : "0";
  timing() {
   timing_type : setup_falling;
   intrinsic_rise : 1.0;
   intrinsic_fall : 1.0;
   related_pin : "A";
  }
  timing() {
   timing_type : hold_falling;
   intrinsic_rise : 1.0;
   intrinsic_fall : 1.0;
   related_pin : "A";
  }
 }
pin(Q) {
  direction : output;
  timing() {
   timing_type : rising_edge;
   intrinsic_rise : 1.0;
   intrinsic_fall : 1.0;
   rise_resistance : 0.1;
   fall_resistance : 0.1;
   related_pin : "CK IH";
  }
```

```
  timing() {
   timing_type : rising_edge;
   intrinsic_rise : 1.0;
   intrinsic_fall : 1.0;
   rise_resistance : 0.1;
   fall_resistance : 0.1;
   related_pin : "B";
  }
 }
 pin(QN) {
  direction : output;
  timing() {
   timing_type : rising_edge;
   intrinsic_rise : 1.0;
   intrinsic_fall : 1.0;
   rise_resistance : 0.1;
   fall_resistance : 0.1;
   related_pin : "CK IH";
  }
  timing() {
   timing_type : rising_edge;
   intrinsic_rise : 1.0;
   intrinsic_fall : 1.0;
   rise_resistance : 0.1;
   fall_resistance : 0.1;
   related_pin : "B";
  }
 }
 statetable( "C TC D A B SI", "IQ1 IQ2" ) {
    table :  "\
/* C TC D    A B  SI     IQ1 IQ2  */ \
  H - -  L -  - : -  - : N  -, /* mast hold */\
  - H -  L -  - : -  - : N  -, /* mast hold */\
  H - -  H - L/H : -  - : L/H -, /* scan mast */\
  - H -  H - L/H : -  - : L/H -, /* scan mast */\
   L  L L/H  L   -  - : -  - : L/H -, /*  D in mast */\
  L L -  H -  - : -  - : X  -, /* both active */\
  H - -  - L  - : L/H - : - L/H, /* slave loads */\
  - H -  - L  - : L/H - : - L/H, /* slave loads */\
  L L -  - -  - : - - : -  N, /* slave loads */\
  - - -  - H  - : - - : -  N"; /* slave loads */
 }
 test_cell(){
  pin(D,CK){
   direction : input
  }
   pin(IH){
    direction : input;
    signal_type : "test_clock";
   }
   pin(SI){
    direction : input;
    signal_type : "test_scan_in";
   }
   pin(A){
    direction : input;
    signal_type : "test_scan_clock_a";
   }
   pin(B){
    direction : input;
    signal_type : "test_scan_clock_b";
   }
   ff ("IQ","IQN"){
    next_state : "D";
    clocked_on : "CK";
   }
   pin(Q){
    direction : output;
    function : "IQ";
    signal_type : "test_scan_out";
   pin(QB){
```

```
    direction : output;
    function : "IQN";
    signal_type : "test_scan_out_inverted";
   }
  }
 }
```

8. 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. Liberty syntax is continually being enhanced 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 retention cells (a sequential cell block that is used to store the state of the sequential element during the power down state). 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:

8.1 Power and Ground (PG) Pins

Liberty provides support for power and ground library pins. 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 power and ground pins.

Important:

All future Liberty syntax updates will be based on the power and ground pin Liberty syntax version Y-2006.06 and later.

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

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.

8.1.1 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*/
```

8.1.2 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.

8.1.3 Cell-Level Attributes

This section describes cell-level attributes for pg_pin groups.

pg_pin Group

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

voltage_name 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 back-bias modeling. *Back 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 back-bias modeling. The pwell and nwell values specify regular wells, and deeppwell and deepnwell specify isolation wells.

Table 8-1 describes the pg_type values.

Table 8-1 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 back-bias modeling.
nwell	Specifies regular n-wells for back-bias modeling.
deepnwell	Specifies isolation n-wells for back-bias modeling.
deeppwell	Specifies isolation p-wells for back-bias modeling.

physical_connection Simple Attribute

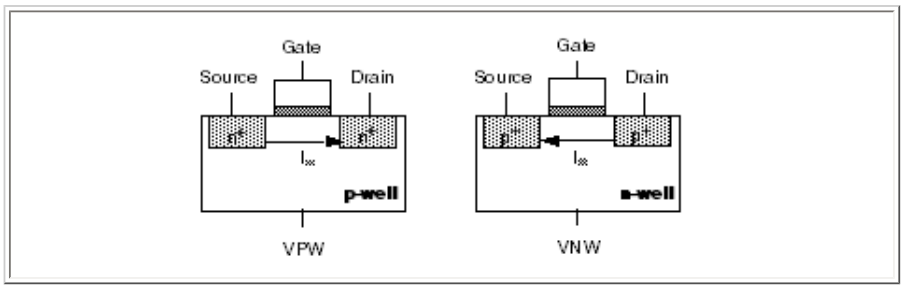
The physical_connection attribute can have the following values: device_layer and routing_pin. The device_layer value specifies that the bias connection is physically external to the cell. In this case, the library provides biasing tap cells that connect through the device layers. The routing_pin value 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 back-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 8-1 shows transistors with p-well and n-well back-bias pins.

Figure 8-1 Transistors With p-well and n-well Back-Bias Pins



user_pg_type Simple Attribute

The user_pg_type optional attribute allows you to customize the type of power and ground pin. It accepts any string value. The following example shows a pg_pin library with the user_pg_type attribute specified. 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;
}
```

8.1.4 Pin-Level Attributes

This section describes pin-level attributes.

power_down_function Attribute

The power_down_function string attribute is used to identify the condition when an output pin is switched off by pg_pin and to specify the Boolean condition under which the cell's output pin is switched off (when the cell is in off mode due to the external power pin states). If power_down_function is set to 1, then X is assumed on the pin.

related_power_pin and related_ground_pin Attributes

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

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

- The 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 8-2 lists the power and ground pin attributes and groups in the old syntax and maps them to the attributes and groups in the new syntax, introduced in the Y-2006.06 release.

Table 8-2 Power and Ground Pin Syntax Changes

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

8.1.5 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) {
...
}
```

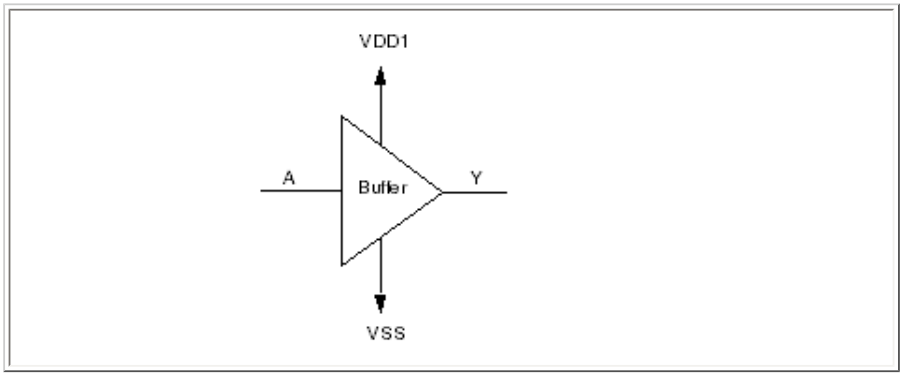
and

```
pg_pin(VSS) {
…
}
```

8.1.6 Standard Cell With One Power and Ground Pin Example

Figure 8-2 shows a standard cell with a power and ground pin.

Figure 8-2 Standard Cell Buffer Schematic



Example

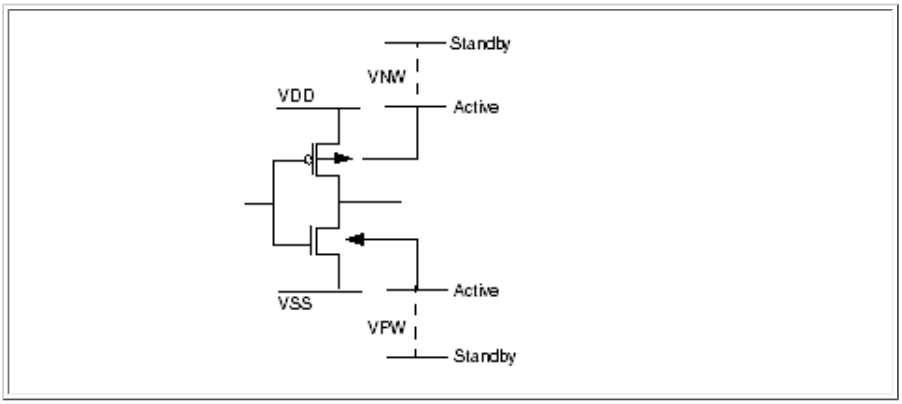
```
library(Standard_cell_library_example) {

 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*/
```

8.1.7 Inverter With Back-Bias Pins Example

Figure 8-3 shows an inverter with back-bias pins.

Figure 8-3 Inverter With Back-Bias Pins



Example

```
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, 0.0);  /* bias power */
voltage_map(VPW, 0.8);  /* 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;
  }
}
}
```

8.2 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.

8.2.1 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.

8.2.2 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

8.2.3 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 */
```

8.2.4 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.

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.

8.2.5 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.

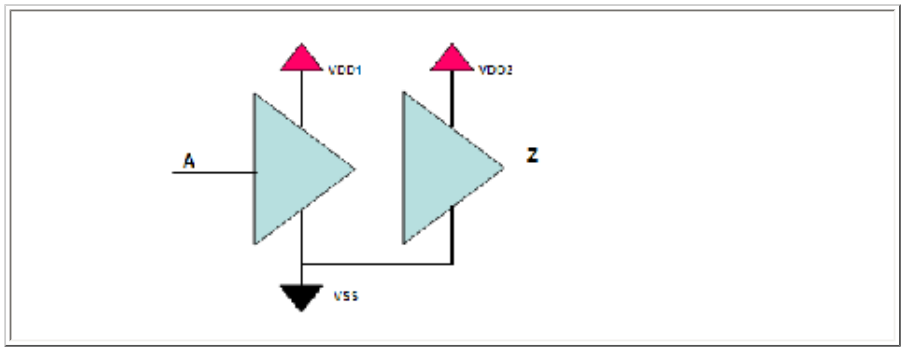
8.2.6 Level-Shifter Modeling Examples

The following sections provide examples for a simple buffer type low-to-high level-shifter cell, a simple buffer type 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 8-4 shows a simple low-to-high level-shifter cell modeled using the power and ground pin syntax and level-shifter attributes.

Figure 8-4 Buffer Type Low-to-High Level-Shifter Cell



Example

```
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;
   ...
  }
```

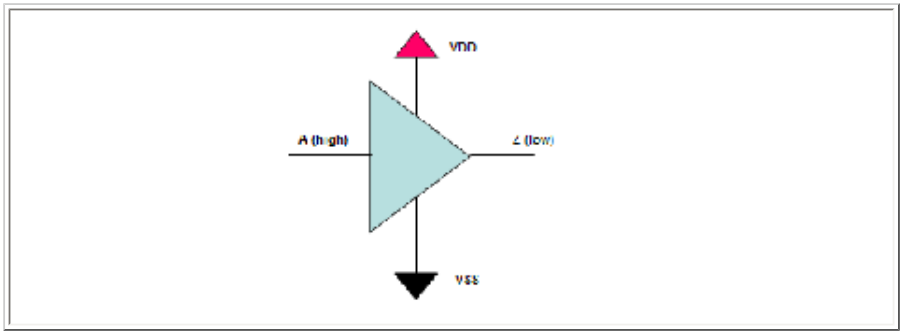
```
 } /* end pin group*/
 ...
 } /*end cell group*/
 ...
} /*end library group*/
```

Simple Buffer Type High-to-Low Level-Shifter

Figure 8-5 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.

The cell in Figure 8-5 is also 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.

Figure 8-5 Buffer Type High-to-Low Level-Shifter



Example

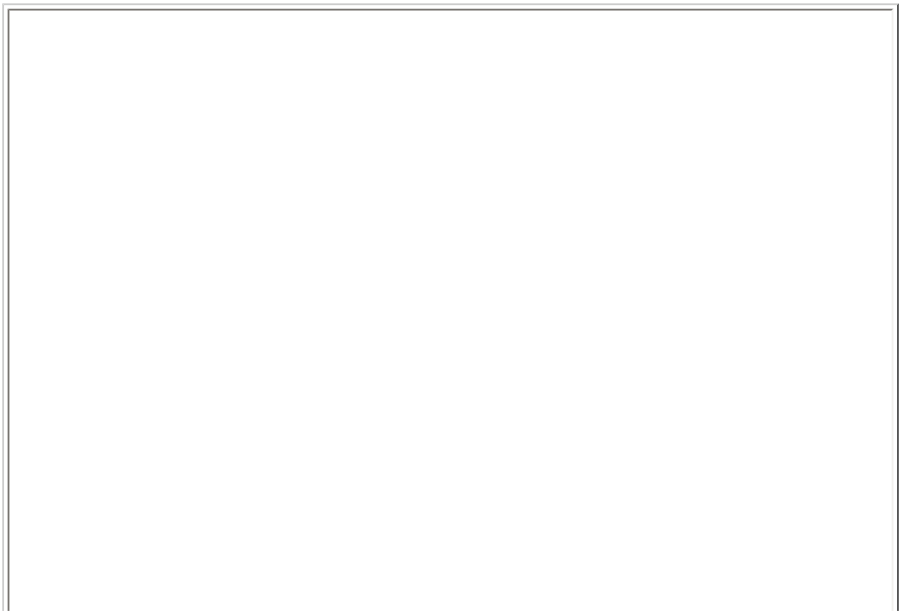
```
library(Level_Shifter_cell_Library_example) {
 voltage_map(VDD1, 1.2);
 voltage_map(VDD, 0.8);
 voltage_map(VSS, 0.0);
 operating_conditions(XYZ) {
  process : 1.0;
  voltage : 3.0;
  temperature : 25.0;
 }
 default_operating_conditions : XYZ;

 cell(Buffer_Type_LH_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;
```

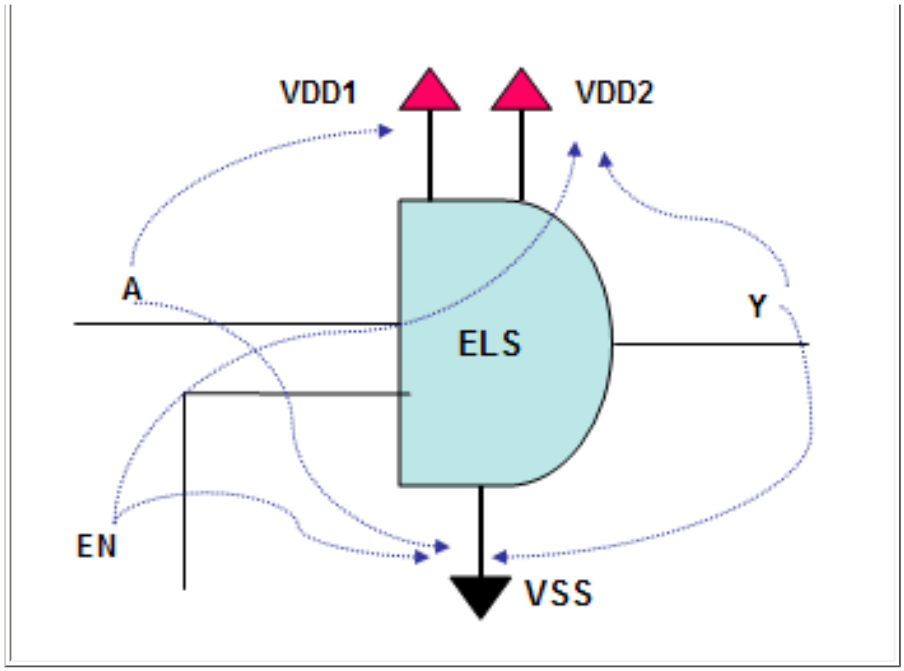
```
    related_ground_pin : VSS;
    input_voltage_range ( 1.1 , 1.3);
   }

   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 Example

This section provides an example of a library containing an enable level shifter. The enable level shifter shifts voltage levels between and locks the output to a logic value. Figure 8-6 shows an enable level-shifter cell schematic. The blue arrows highlight the signal pins that are associated with the power and ground pin pairs in the schematic. Note that the enable pin is related to VDD2.

Figure 8-6 Enable Level-Shifter Cell Schematic





Example

```
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 ( 0.7 , 0.9);
   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 back-bias modeling.

Example

```
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 */
```

8.3 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.

Syntax

```
cell(isolation_cell) {
 is_isolation_cell : true ;
...
 pg_pin(<pg_pin_name_P>) {
  pg_type : primary_power;
  ...
 }
 pg_pin(<pg_pin_name_G>) {
  pg_type : primary_ground;
  ...
 }

 pin (data) {
  direction : input;
  isolation_cell_data_pin : true ;
  ...
 }/* End pin group */

 pin (enable) {
  isolation_cell_enable_pin : true ;
```

```
    ...
    }/* End pin group */
  ...
    pin (output) {
     direction : output;
     power_down_function : (!pg_pin_name_P + pg_pin_name_G);
     ...
    }/* End pin group */

  }/* End Cell group */
```

8.3.1 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.

8.3.2 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

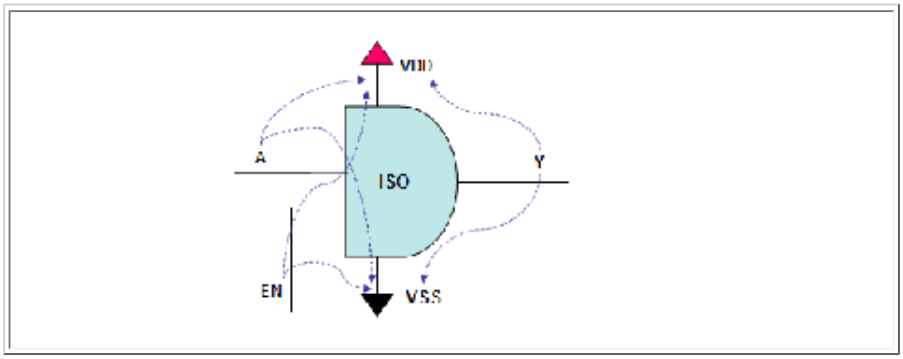
The `power_down_function` string attribute identifies the Boolean condition under which the cell's output pin is switched off (when the cell is in off mode due to the external power pin states). If the `power_down_function` is set to 1, then X is assumed on the pin.

8.3.3 Isolation Cell Example

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

Figure 8-7 shows an isolation cell schematic. The library in this example describes only the portion related to the power and ground pin syntax. The blue arrows highlight the signal pins that are associated with the power and ground pin pairs in the schematic.

Figure 8-7 Isolation Cell Schematic



Example

```
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*/
```

8.4 Switch Cell Modeling

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

- Coarse grain

  There are two types of coarse-grain switch cells: header switch cells, 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, the Liberty syntax supports only fine-grain switch cells for macro cells.

8.4.1 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 Z.

Syntax

The following syntax is a portion of the coarse-grain switch cell syntax:

```
library(<coarse_grain_library_name>) {
…

lu_table _template ( template_name )
 variable_1 : input_voltage;
 variable_2 : output_voltage;
 index_1 ( <float>, … );
 index_2 ( <float>, … );
}
…
cell(<cell_name>) {
 switch_cell_type : coarse_grain;
 …
 pg_pin ( <VDD/VSS pin name> ) {
  pg_type : primary_power | 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 */
```

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` table used for steady state current modeling must be defined at the cell level.

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 the switch cell type does not need to be inferred 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 a Z to) the controlled design partitions, including the

traditional signal input pins only with no related power pins to this output.

switch_pin Attribute

The `switch_pin` attribute is a pin-level Boolean attribute. When it is set to true, it is used to identify the pin as the switch pin of a coarse-grain switch cell.

function Attribute

The `function` attribute in a pin group defines the value of an output pin or inout pin in terms of the input pins or inout pins in the cell group or model group. The `function` attribute describes the Boolean function of only nonsleep input signal pins.

pg_function Attribute

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

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.

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.

8.4.2 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.

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.

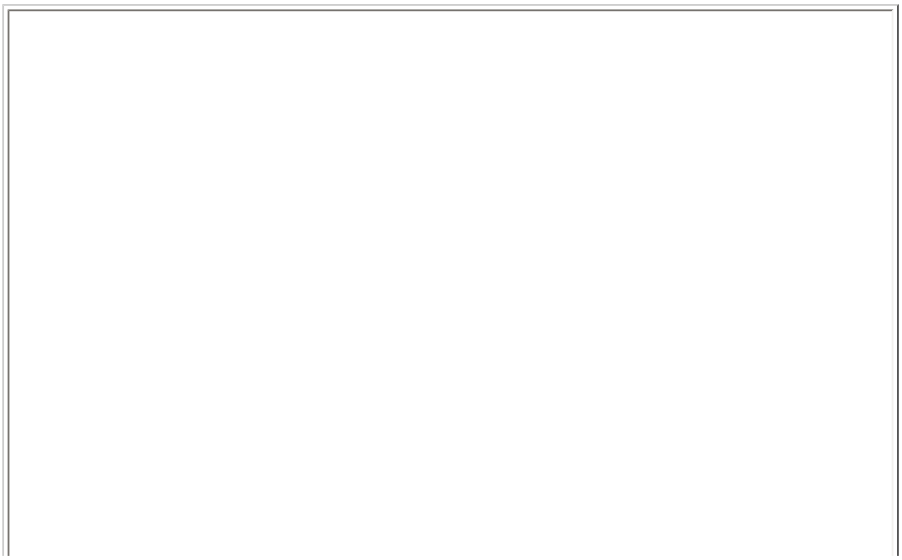
8.4.3 Switch-Cell Modeling Examples

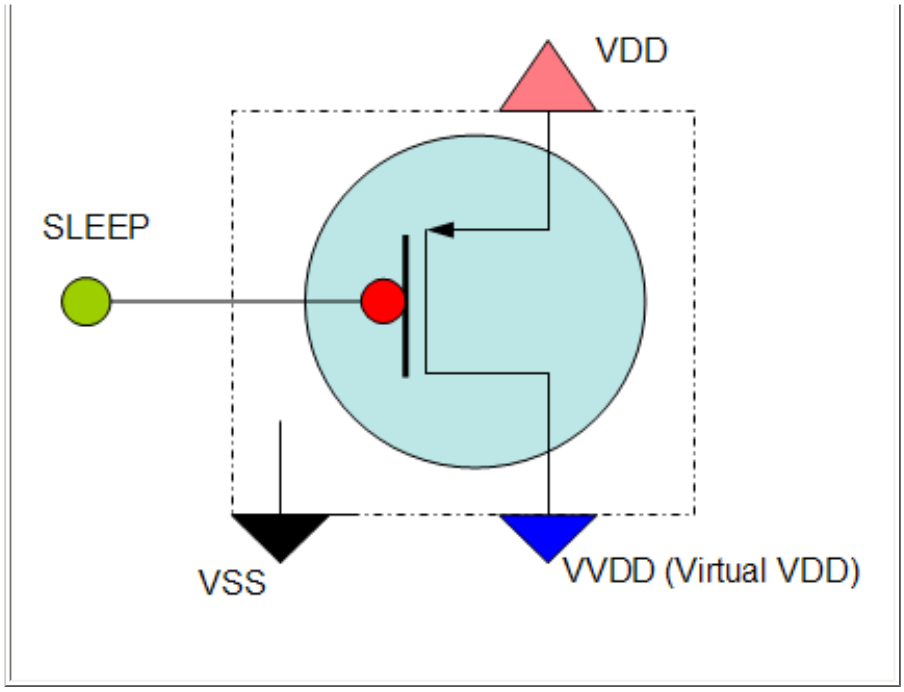
The following sections provide examples for a simple coarse-grain header switch cell and a complex coarse-grain header switch cell.

Simple Coarse-Grain Header Switch Cell

Figure 8-8 shows a simple coarse-grain header switch cell.

Figure 8-8 Simple Coarse-Grain Header Switch Cell





Example

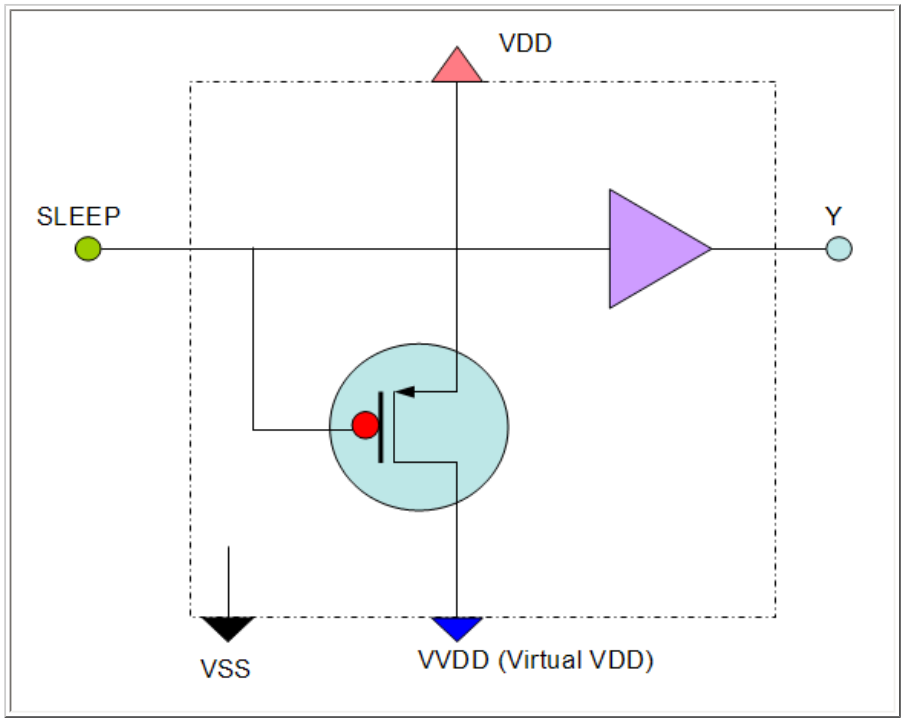
```
library (simple_coarse_grain_lib) {

 …
 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 ) {
/* index_1 specifies the input_voltage value specified at the switch pin with
 respect to the ground */
/* index_2 specifies the voltage value specified at the related_internal_pg_pin
 with respect to the ground */
  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 8-9 shows a complex coarse-grain header switch cell.

Figure 8-9 Complex Coarse-Grain Header Switch Cell



Example

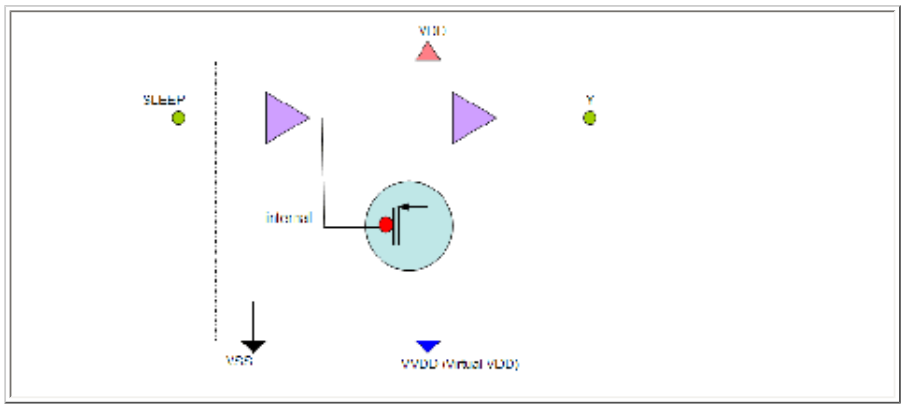
```
library (Complex_CG_lib) {
 …
 current_unit : 1mA;
 …
 voltage_map(VDD, 1.0);
 voltage_map(VVDD, 0.8);
 voltage_map(VSS, 0.0);

 operating_conditions(XYZ) {
  process : 1.0;
  voltage : 1.0;
  temperature : 25.0;
 }
 default_operating_conditions : XYZ;

 lu_table_template ( ivt1 ) {
 /* index_1 specifies the input_voltage value specified at the switch pin with
respect to the ground */
 /* index_2 specifies the voltage value specified at the related_internal_pg_pin
with respect to the ground */
 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 8-10 shows a complex coarse-grain switch cell with an internal switch pin.

Figure 8-10 Complex Coarse-Grain Switch Cell With an Internal Switch Pin



Example

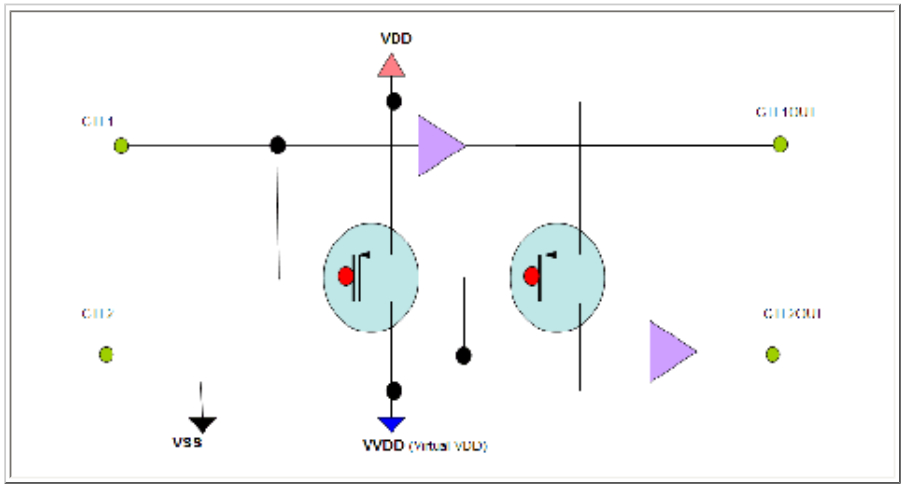
```
library (Complex_CG_lib) {
 …
 current_unit : 1mA;
 …

 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 ) {
 /* index_1 specifies the input_voltage value specified at the switch pin with
respect to the ground */
 /* index_2 specifies the voltage value specified at the related_internal_pg_pin
with respect to the ground */
 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 8-11 shows a complex coarse-grain switch cell with two parallel switches.

Figure 8-11 Complex Coarse-Grain Switch Cell With Two Parallel Switches



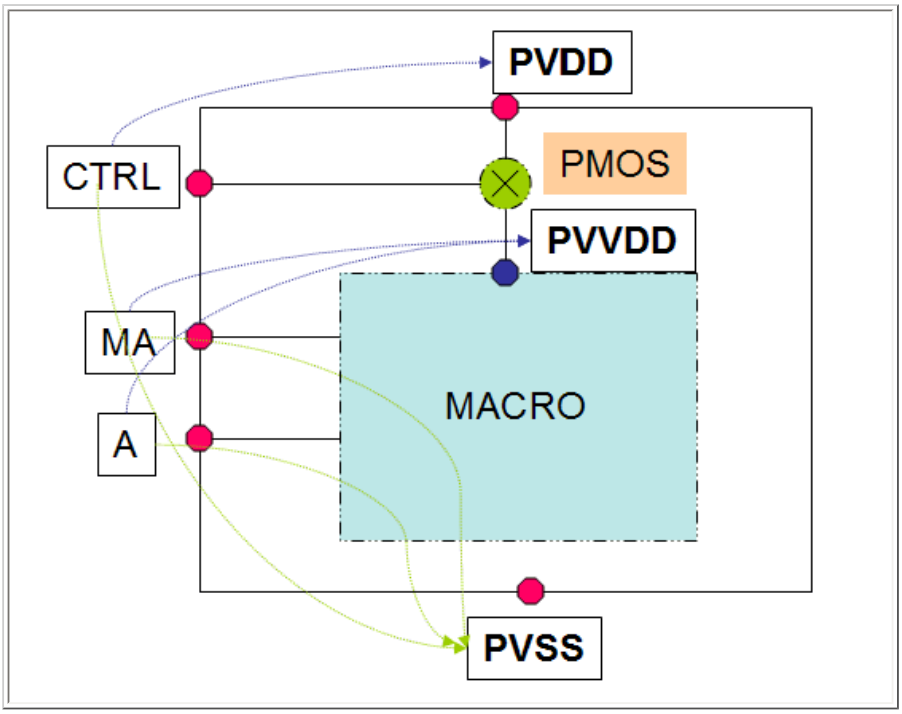

```
library (Complex_CG_lib) {
 …
 current_unit : 1mA;
 …

 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 ) {
 /* index_1 specifies the input_voltage value specified at the switch pin with
respect to the ground */
 /* index_2 specifies the voltage value specified at the related_internal_pg_pin
with respect to the ground */
 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 8-12 shows a macro cell with fine-grained internal power switches. The arrows highlight the signal pins that are associated with the power and ground pin pairs in the schematic.

Figure 8-12 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;
 …
 }
}
```

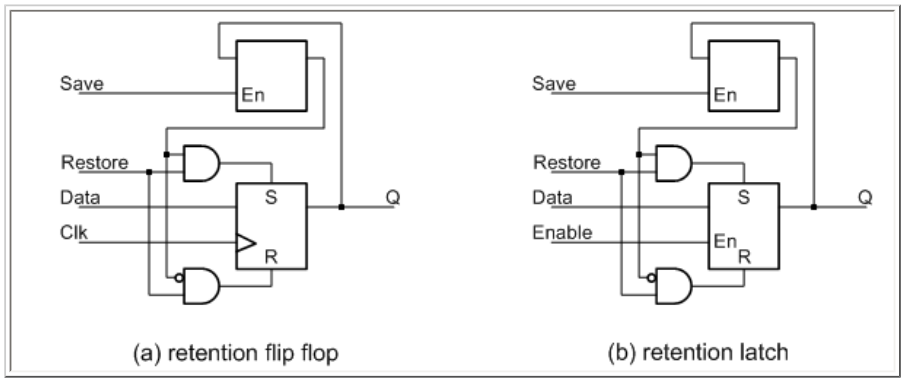
8.5 Retention Cell Modeling

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 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.

Figure 8-13 shows a simple retention flip-flop and retention latch structure.

Figure 8-13 Simple Retention Flip-Flop and Retention Latch Structure



8.5.1 Modeling Retention Flip-Flops

Retention registers consist of two separate elements: a regular flip-flop and a retention latch. The retention latch consists of a balloon circuit modeled with high-threshold transistors. The retention part of the logic is generally powered by a backup power supply.

In addition to the separate power supplies, additional signals such as sleep and wake are required to enable the data transfer from the regular flip-flop to the retention latch and back again based on the mode of operation.

During active mode, the regular flip-flop operates at speed, and the retention latch does not add to the load at the output. During sleep mode, the Q data is transferred to the retention latch, and the power supply to the flip-flop is shut off, thus reducing the leakage standby power. When the circuit is activated with the wake-up signal, the data in the retention 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 is also preserved in a retention latch. Other types of retention registers fix the clock at high or low states.

8.5.2 Modeling Retention Latches

Similar to their register counterparts, retention latches consist of a regular latch that can be a low-threshold latch used during normal (fast) operations and a retention latch. The retention latch attributes are the same as those specified for retention registers. In order to be recognized as retention-related pins during logic optimization, the functional description of the latch must include the retention pins. Otherwise, they are considered `dont_use` pins.

Example 8-1 shows the functional description of a normal latch; Example 8-2 shows the retention latch counterpart.

Example 8-1 Normal Latch

```
latch (IQ, IQN) {
  data_in: D ;
  enable : CLK ;
 ...
}
```

Example 8-2 Retention Latch

```
latch (IQ, IQN) {
  data_in: D & (SAVE & RESTORE) ;
  enable : CLK ;
 ...
}
```

If the sleep and wake retention pins are disabled when the value is 0, the functional descriptions of the normal latch and the retention latch in active mode are equivalent.

8.5.3 Syntax

The syntax for retention cells is as follows:

```
cell(<cell_name>) {
 retention_cell : <rention_cell_style>;
 pin(<pin_name>) {
  retention_pin(<pin_class>, <disable_value>);
  ...
 }
 ...
}
```

8.5.4 Cell-Level Attribute

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

retention_cell Simple Attribute

The `retention_cell` attribute identifies a type of retention register with a string as its name.

Note:

Beginning with the Z-2007.03 release, the `power_gating_cell attribute has been replaced by the retention_cell` attribute. However, libraries with the old syntax are supported for backward compatibility.

There can be multiple types of retention registers for a given register cell that provide the same functionality in normal mode but have different sleep and wake signals, or clocking schemes. For example, if a standard D flip-flop cell supports two power gating 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 cell names must differ, such as DFF_type1 and DFF_type2.

8.5.5 Pin-Level Attribute

The following attribute is a pin-level attribute for retention cells.

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.
- Disable value
  Defines the value of the retention pin when the cell works in normal mode. The valid values are 0 and 1.

Table 8-3 lists the retention modeling attributes in the old syntax and maps them to the attributes in the new syntax.

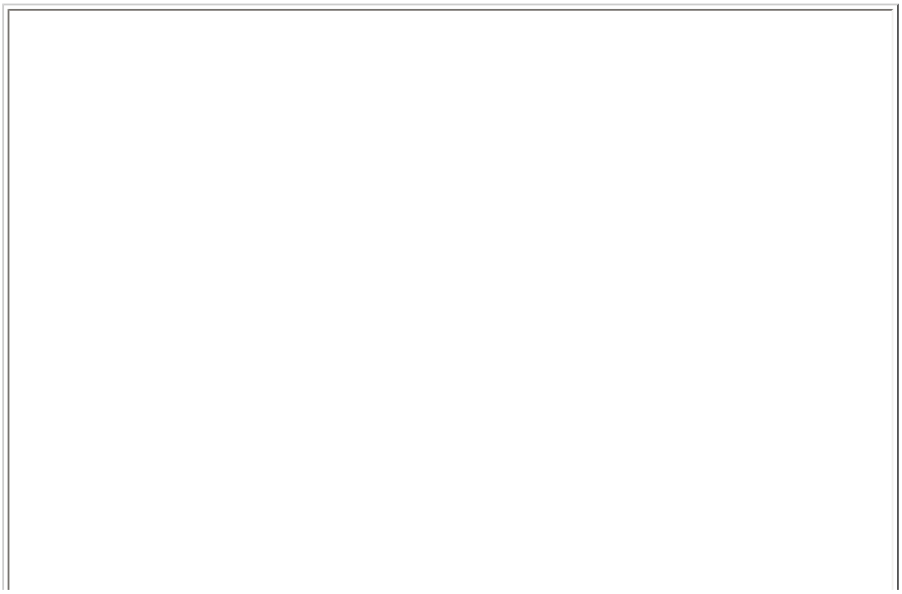
Table 8-3 Retention Modeling Syntax Changes

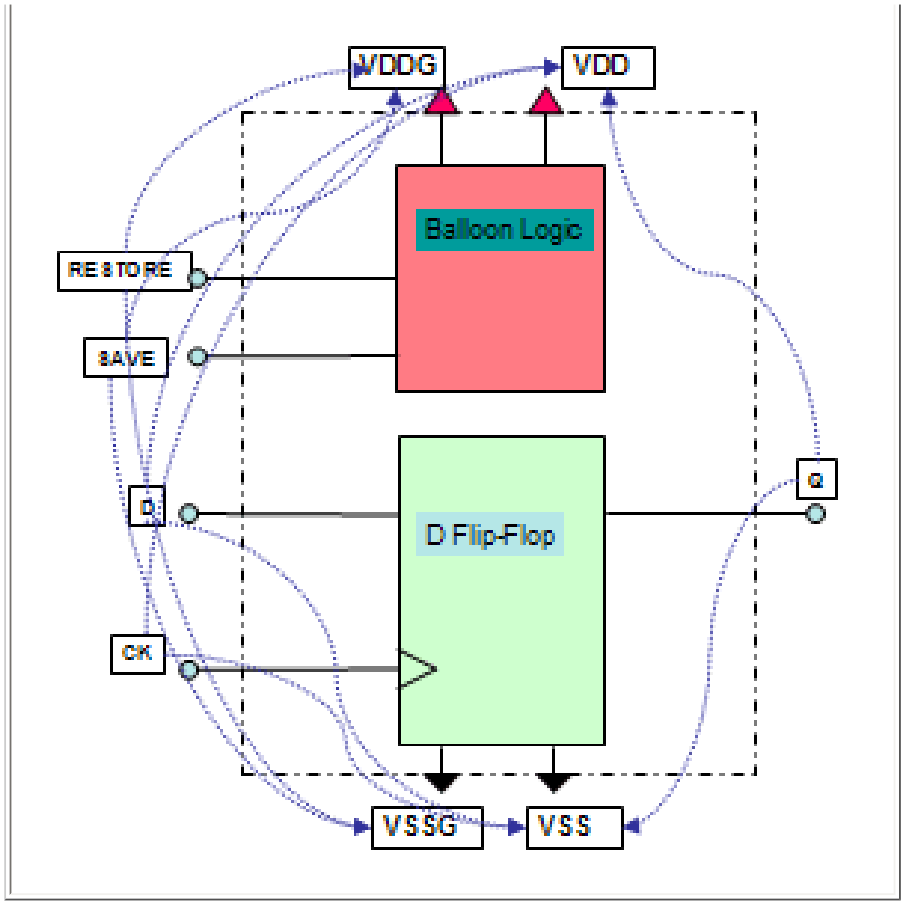
Old syntax	New syntax
`power_gating_cell : <cell type>;` Simple attribute	`retention_cell : <cell type>;` Simple attribute
`power_gating_pin (power_pin_[1-5], " 0 " | " 1 ");` Complex attribute	`retention_pin (<pin_class>, <disable_value>);` Complex attribute

8.5.6 Retention Cell Model Example

Figure 8-14 shows a schematic of a basic retention cell. The blue arrows highlight the signal pins that are associated with the power and ground pin pairs in the schematic.

Figure 8-14 Retention Cell Model Example






```
library(retention_lib_example) {

 delay_model : "table_lookup" ;
…

 voltage_map(VDDG, 0.9);
 voltage_map(VDD, 1.0);
 voltage_map(VSSG, 0.0);
 voltage_map(VSS, 0.0);

 operating_conditions(XYZ) {
  process   : 1;
  temperature : 125;
  voltage   : 1.0;
  tree_type  : balanced_tree;
 }
 default_operating_conditions : XYZ;
 …

cell(RETENTION_DFF) {
 retention_cell : "my_retention_dff" ;
 /* Other cell level information */
 …

 pg_pin(VDDG) {
  voltage_name : VDDG;
  pg_type : backup_power;
 }
 pg_pin(VDD) {
  voltage_name : VDD;
  pg_type : primary_power;
 }
 pg_pin(VSSG) {
  voltage_name : VSSG;
  pg_type : backup_ground;
```

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

pin(D) {
 direction : input;
 capacitance : 1;
 nextstate_type : data ;
 related_power_pin : VDD;
 related_ground_pin : VSS;
 …
}

pin(CP) {
 direction : input;
 capacitance : 1;
 related_power_pin : VDD;
 related_ground_pin : VSS;
  …
  }

pin(SAVE) {
 direction : input;
 capacitance : 1.0;
 nextstate_type : data ;
 related_power_pin : VDDG;
 related_ground_pin : VSSG;
 retention_pin (save, "0") ;
 …
}

pin(RESTORE) {
 direction : input;
 capacitance : 1.0;
 nextstate_type : data ;
 related_power_pin : VDDG;
 related_ground_pin : VSSG;
 retention_pin (restore, "0") ;
 …
}

ff("IQ", "IQN") {
 next_state : "D & (!SAVE & !RESTORE)" ;
 clocked_on : "CP" ;
}

pin(Q) {
 direction : output;
 function : "IQ";
 power_down_function : "!VDD + !VDDG + VSS + VSSG";
 related_power_pin : VDD;
 related_ground_pin : VSS;
  timing() {
   related_pin : CP;
   timing_type : rising_edge;
   cell_rise(template) {
   ...
   }
   cell_fall(template) {
   ...
   }
   rise_transition(template) {
   ...
   }
   fall_transition(template) {
   ...
   }
   …
  } /* End pin */
```

```
 cell_leakage_power : 0.5 ;
 leakage_power() {
  when : "!SAVE & !RESTORE" ;
  value : 0.7 ;
 }
 leakage_power() {
  when : "SAVE" ;
  value : 0.02 ;
 }
 } /* End Cell */
} /* End Library */
```

8.6 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 attributes that can identify them. Library models also need an attribute to identify always-on input pins so that tools can trace all always-on nets crossing domains that are shut down.

Liberty syntax supports always-on cells and pins. There is no restriction on any specific cell. The following cell categories support always-on signals:

- `always_on` cell buffers or inverters
- Pins on retention cells
- Pins on switch cells
- Pins on isolation cells

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 or signal pins. Specify the attribute at the cell level to determine whether a cell is an always-on cell. Specify the attribute at the pin level to determine whether a pin is an always-on signal pin.

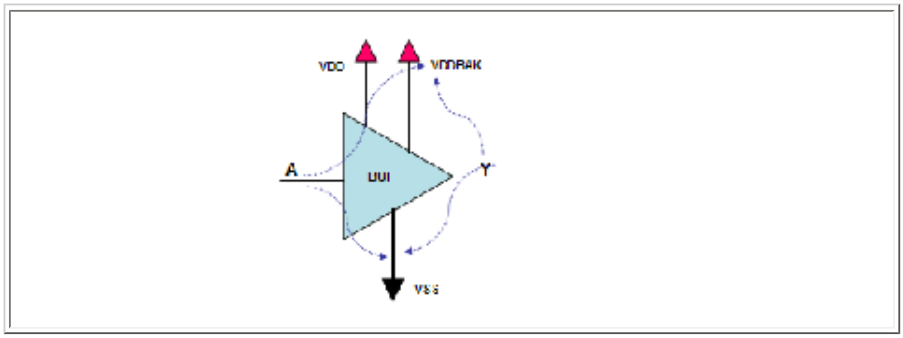
Note:

Some macro cell input pins require that you specify the `always_on` attribute for always-on pins.

Always-On Simple Buffer Example

Figure 8-15 shows a simple always-on cell buffer. The blue arrows in the schematic highlight the signal pins that are associated with the power and ground pin pairs.

Figure 8-15 Simple Always-On Cell Buffer



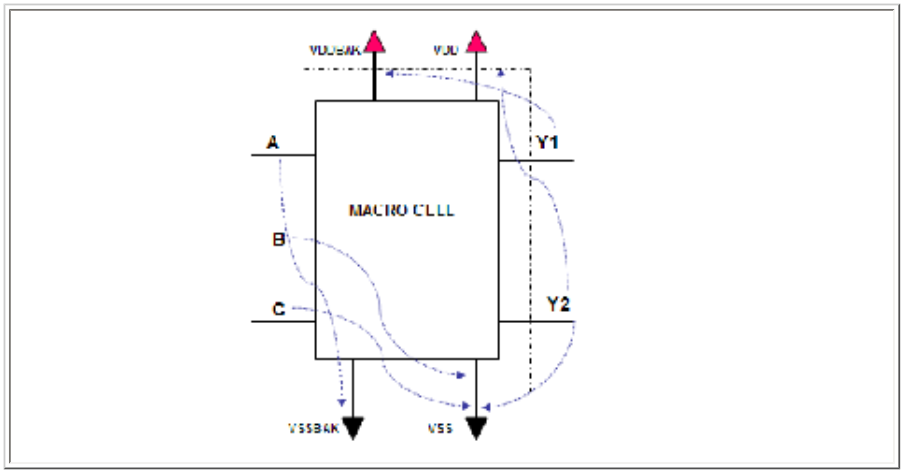
Example

```
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;
    /* 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 8-15 shows a macro cell with one always-on pin. The blue arrows in the schematic highlight the signal pins that are associated with the power and ground pin pairs.

Figure 8-16 Macro Cell with an Always-On Pin



Example

```
library (my_library) {
  …

    voltage_map (VDD, 2.0) ;
    voltage_map (VSS, 0.1) ;
    voltage_map (VDDBAK, 1.0) ;
    voltage_map (VSSBAK, 0.0) ;
  ...

  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;
    }
    pg_pin(VSSBAK) {
      voltage_name : VSSBAK;
      pg_type : backup_ground;
    }
    …
    pin (A) {
      always_on : true;
      related_power_pin : VDDBAK;
      related_ground_pin : VSSBAK;
    /* 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 : VSSBAK;
      power_down_function : "!VDD + !VDDBAK + VSS + VSSBAK";
    /* 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 + VSSBAK";
    /* Other pin specific information */
    } /* End Pin group */
                    …
  } /* End Cell group */
                    …
} /* End Library group */
```

9. 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:

9.1 Modeling Power Terminology

The power a circuit dissipates falls into two broad categories:

- Static power
- Dynamic power

9.1.1 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*.

9.1.2 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

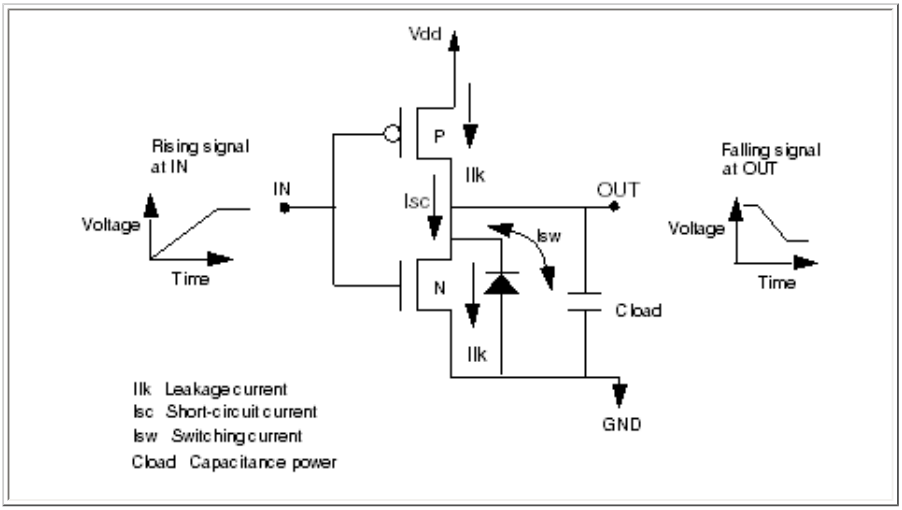
Internal Power

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 9-1 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 9-1 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.

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 9-1 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.

9.2 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.

9.3 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.

9.4 Representing Leakage Power Information

The Liberty syntax lets you represent leakage power information with

- The `cell_leakage_power` attribute
- The `leakage_power` group with a single value
- The `leakage_power` group with a polynomial
- The associated library-level attributes that specify scaling factors, units, and a default value for both leakage and power density

9.4.1 cell_leakage_power Simple Attribute

This attribute specifies the leakage power of a cell. You must define this attribute for cells with state-dependent leakage power. If this attribute is missing or negative, the value of the `default_cell_leakage_power` attribute is used.

Syntax

cell_leakage_power : *value*$_{float}$;

value

A floating-point number indicating the leakage power of the cell.

Syntax

`cell_leakage_power : value ;`

9.4.2 Using the leakage_power Group for a Single Value

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

- when
- value

Syntax

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

when Simple Attribute

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

Syntax

when : "*Boolean expression*";

Boolean expression

The name or names of pins, buses, and bundles with their corresponding Boolean operators. Table 9-1 lists valid operators.

Table 9-1 Valid Boolean Operators

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

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

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

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

Syntax

when : "*bus_name*";

Example

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

value Simple Attribute

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

Syntax

value : $value_{float}$;

value

A floating-point number representing the leakage power value.

The following example defines the `leakage_power` group and the `cell_leakage_power` simple attribute in a cell:

Example

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

```
  cell_leakage_power : 3.0;
}
```

9.4.3 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 Liberty 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 Liberty 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() {
      ...
    }
  }
}
```

9.4.4 leakage_power_unit Simple Attribute

This attribute indicates the units of the leakage-power values in the library. Table 9-2 shows the possible unit values you can enter and their corresponding mathematical symbol equivalent.

Syntax

leakage_power_unit : *value*$_{enum}$;

value

Valid values are 1mW, 100uW (for 100mW), 10uW (for 10mW), 1uW (for 1mW), 100nW, 10nW, 1nW, 100pW, 10pW, and 1pW.

Table 9-2 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;
```

9.4.5 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.

Syntax

default_cell_leakage_power : *value* ;

value

A non-negative floating-point number.

Example

```
default_cell_leakage_power : 0.5;
```

9.4.6 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 ;
    }
  }
}
```

9.5 Modeling for Internal and Switching Power

These are two compatible definitions of internal or short-circuit power:

- Short-circuit power is the power dissipated by the instantaneous short-circuit connection between Vdd and GND while the gate is in transition.
- Internal power is all the power dissipated within the boundary of the gate. This definition does not distinguish between the cell's short-circuit power and the component of switching power that is being dissipated internally to the cell as a result of the drain-to-substrate capacitance that is being charged and discharged. In this definition, the interconnect switching power is the power dissipated because of lumped wire capacitance and input pin capacitances but not because of the output pin capacitance.

Library developers must choose one of these definitions and specify internal power and capacitance numbers accordingly. Library developers can choose

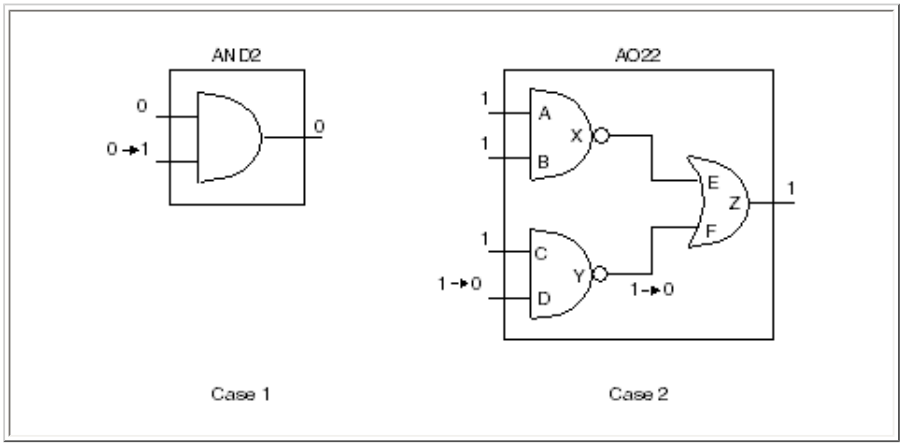
- To include the effect of the output capacitance in the `internal_power` attribute, which gives the output pins zero capacitance
- To give the output pins a real capacitance, which causes them to be included in the switching power, and model only short-circuit power as the cell's internal power

Together, internal power and switching power contribute to the total dynamic power dissipation. Like switching power, internal power is dissipated whenever an output pin makes a transition.

Some power is 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 9-2 shows two examples of an input transition that does not cause a corresponding output transition.

Figure 9-2 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.

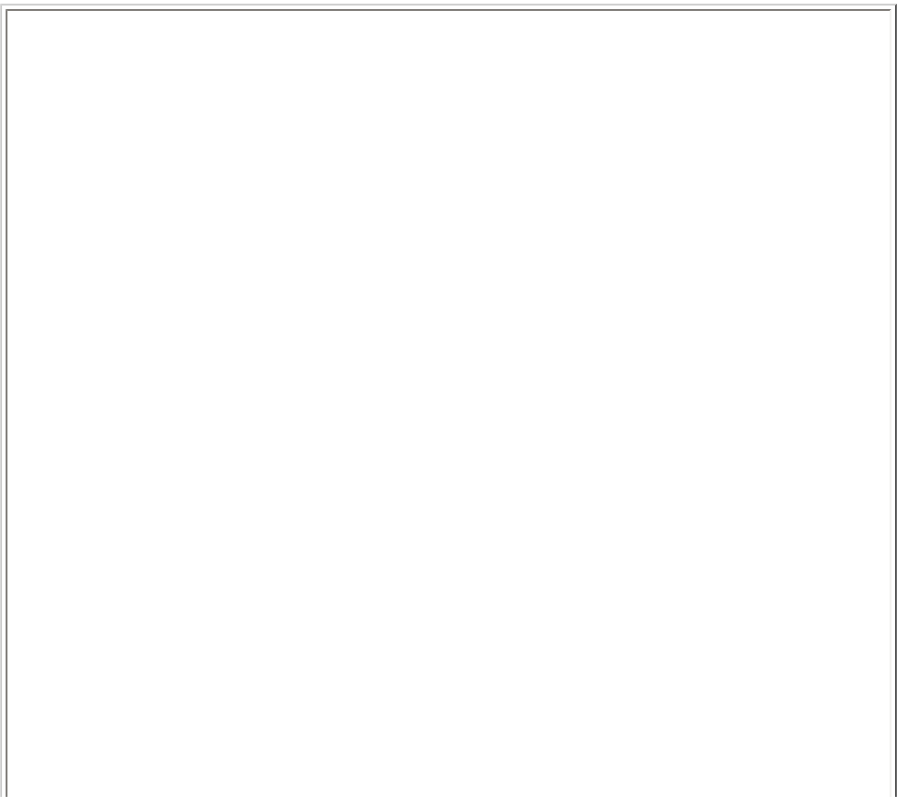
Pins transitioning within the same time period can lower the level of power consumption.

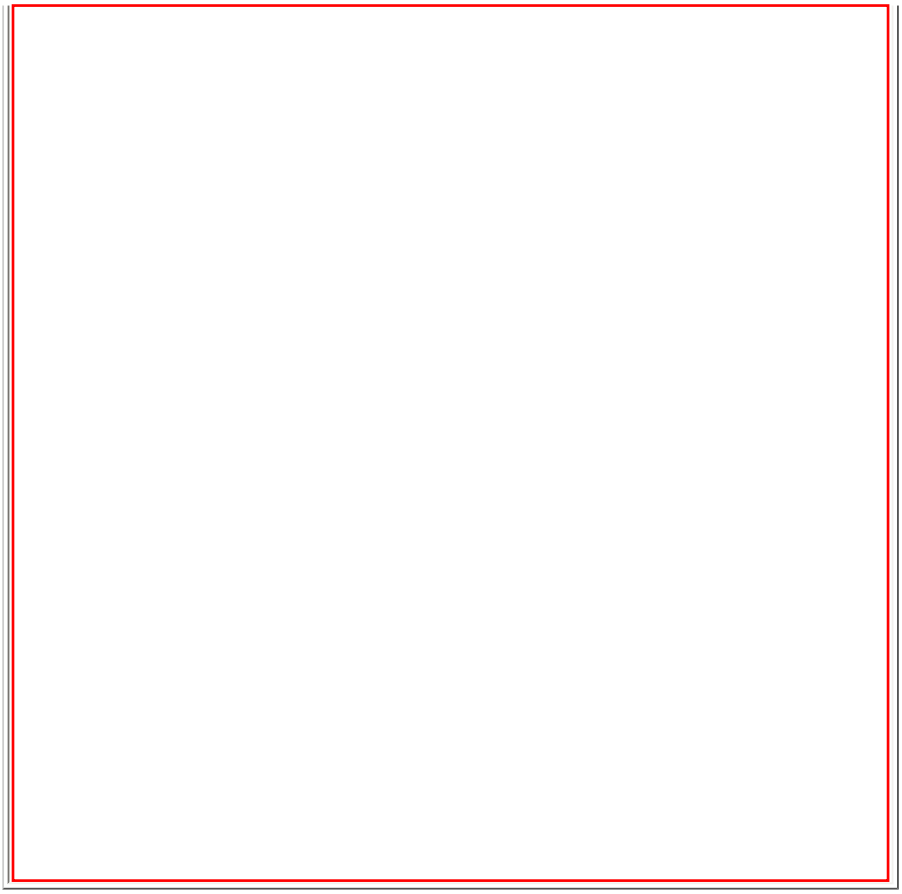
9.5.1 Modeling Internal Power Lookup Tables

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.

Figure 9-3 is an example of the two-dimensional lookup table for modeling output pin power in a cell.

Figure 9-3 Internal Power Table for Cell Output





9.6 Representing Internal Power Information

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

9.6.1 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, table_lookup is assumed..

9.6.2 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")
- 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.

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".

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 9-1 illustrates.

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

Example 9-1 shows four `power_lut_template` groups that have one-, two-, or three-dimensional templates.

Example 9-1 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.

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.

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.

9.6.3 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")
- The associated library-level attributes that specify scaling factors and a default value

power_poly_template Group

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

Syntax

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

variable_n_range(*min_value*$_{float}$, max_value$_{float}$)

mapping(*value*$_{enum}$, *power_rail_name*$_{id}$)

}

}

}

power_poly_template Variables

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

temperature

The temperature.

voltage

The primary power supply voltage.

voltagei

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

input_net_transition

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

total_output_net_capacitance

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

equal_or_opposite_output_net_capacitance

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

parameter 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, the value defined in the voltage attribute within the operating_conditions group is assumed.

```
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`.

```
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) ;
```

9.7 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.

9.7.1 Naming Power Relationships, Using the internal_power Group

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

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

- Between a single pin and a single related pin

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

Power Relationship Between a Single Pin and a Single Related Pin

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

Example

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

    direction : input;
    capacitance : 1;
  }
  pin (B) {
    direction : output
    function : "A′";
    internal_power (A_B) }
      related_pin : "A";
    ...
    }/* end internal_power() */
  }/* end pin B */
}/* end cell */
```

The power relationship is as follows:

From pin	To pin	Label
A	B	A_B

Power Relationships Between a Single Pin and Multiple Related Pins

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

Example

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

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

```
      ...
    }/* end internal_power() */
  }/* end pin B */
}/* end cell */
```

The power relationships are as follows:

From pin	To pin	Label
A	C	A_C
B	C	B_C

Power Relationships Between a Bundle and a Single Related Pin

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

Example

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

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

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

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

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

G3	Q3	G_Q3

Note:

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

Power Relationships Between a Bundle and Multiple Related Pins

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

Example

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

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

From pin	To pin	Label
G	Q0	G_Q0
H	Q0	H_Q0
G	Q1	G_Q1
H	Q1	H_Q1
G	Q2	G_Q2
H	Q2	H_Q2
G	Q3	G_Q3
H	Q3	H_Q3

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

From pin	To pin	Label
G0	Q0	G_Q0
H	Q0	H_Q0
G1	Q1	G_Q1

H	Q1	H_Q1
G2	Q2	G_Q2
H	Q2	H_Q2
From pin	To pin	Label
G3	Q3	G_Q3
H	Q3	H_Q3

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

Note:

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

Power Relationships Between a Bus and a Single Related Pin

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

Example

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

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

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

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

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

Power Relationships Between a Bus and Multiple Related Pins

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

Example

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

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

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

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

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

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

Power Relationships Between a Bus and Related Bus Pins

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

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

9.7.2 internal_power Group

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

Simple attributes:

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

Groups:

- `fall_power`
- `power`
- `rise_power`

equal_or_opposite_output Simple Attribute

This attribute designates an optional output pin or pins whose capacitance provides access to a 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 " 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.

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.

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.

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.

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 " ; "rising_together_group Simple Attribute " ; 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 ” 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 9-1 lists the Boolean operators valid in a when statement.

Example

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

fall_power Group

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

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

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

Complex Attributes

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

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
```

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
```

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 9- shows cells that contain internal power information in the `pin` group.

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 ;
```

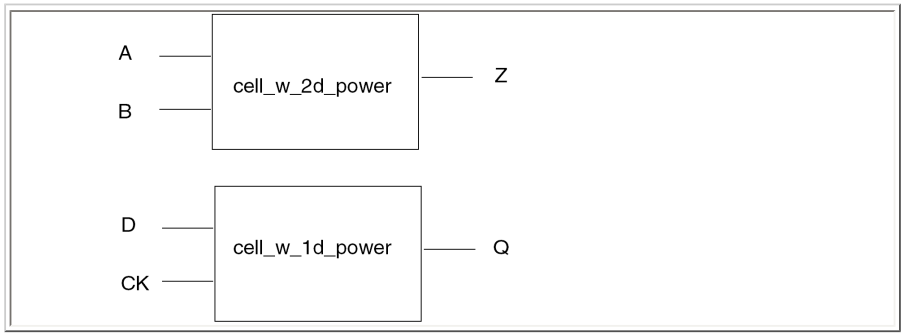
Example

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

9.7.3 Internal Power Examples

The examples in this section show how to describe the internal power of the 2-input sequential gate in Figure 9-4, using one-, two-, and three-dimensional lookup tables.

Figure 9-4 Library Cells With Internal Power Information



One-Dimensional Power Lookup Table

Example 9-2 is the library description of the cell shown in Figure 9-4, a cell with a one-dimensional internal power table defined in the `pin` groups.

Example 9-2 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") ;
        }
      }
    }
  }
}
```

Two-Dimensional Power Lookup Table

Example 9-3 is the library description of the cell in Figure 9-4, a cell with a two-dimensional internal power table defined in the `pin` groups.

Example 9-3 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 ;
      ...
    }
  }
}
```

Three-Dimensional Power Lookup Table

Example 9-4 is the library description for the cell in Figure 9-4, a cell with a three-dimensional internal power table.

Example 9-4 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" ;
      }
      ...
    }
    ...
  }
}
```

9.8 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.

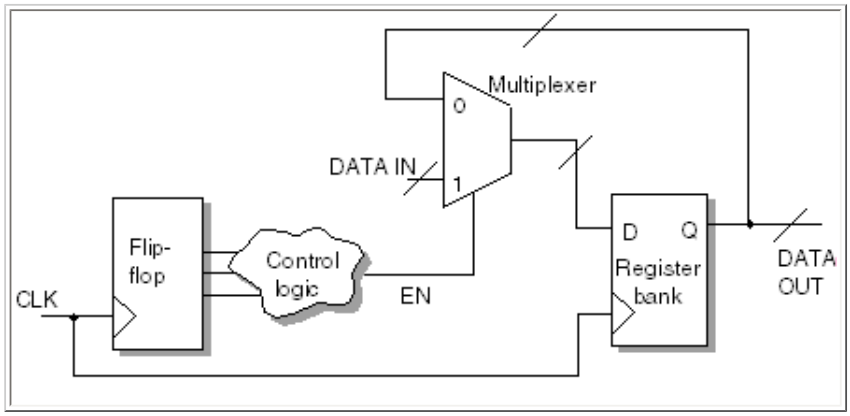
9.8.1 What Clock Gating Does

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.

Figure 9-5 shows a simple implementation of a register bank using a multiplexer and a feedback loop.

Figure 9-5 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” .

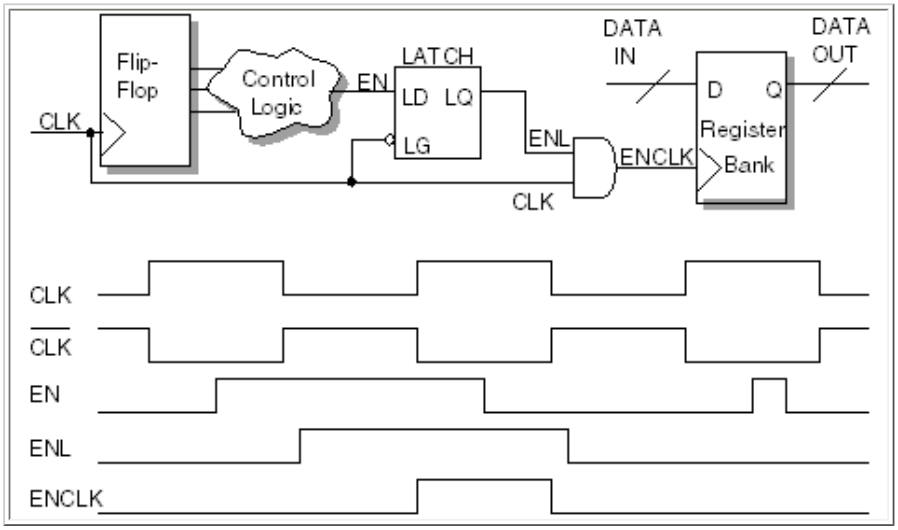
9.8.2 Looking at a Gated Clock

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 9-5, 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 9-6 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 9-5. 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.

9.8.3 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.

9.8.4 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 9-3. Setting some of the pin attributes, such as those for test and observability, is optional.

Table 9-3 Pin Attributes for Integrated Clock-Gating Cells

Integrated cell pin name	Data direction	Required attribute

clock	in	clock_gate_clock_pin
enable	in	clock_gate_enable_pin
test_mode or scan_enable	in	clock_gate_test_pin
observability	out	clock_gate_obs_pin
enable_clock	out	clock_gate_out_pin

clock_gate_clock_pin Attribute

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

Syntax

clock_gate_clock_pin : true | false ;

true | *false*

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

Example

clock_gate_clock_pin : true;

clock_gate_enable_pin Simple Attribute

Use the clock_gate_enable_pin attribute to compile a design containing gated clocks.

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

Syntax

clock_gate_enable_pin : true | false ;

true | *false*

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

Example

clock_gate_enable_pin : true;

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

clock_gate_test_pin Attribute

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

Syntax

clock_gate_test_pin : true | false ;

true | *false*

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

Example

clock_gate_test_pin : true;

clock_gate_obs_pin Attribute

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

Syntax

clock_gate_obs_pin : true | false ;

true | *false*

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

Example

clock_gate_obs_pin : true;

clock_gate_out_pin Attribute

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

Syntax

clock_gate_out_pin : true | false ;

true | *false*

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

Example

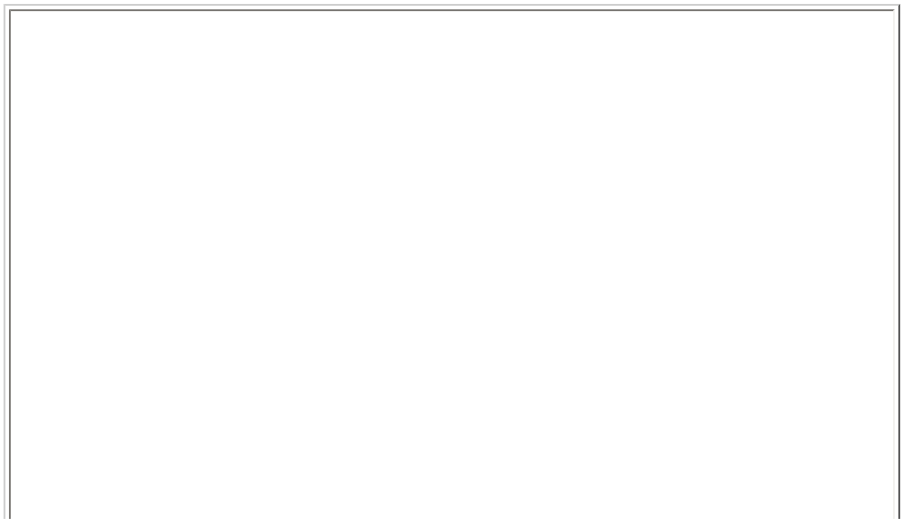
clock_gate_out_pin : true;

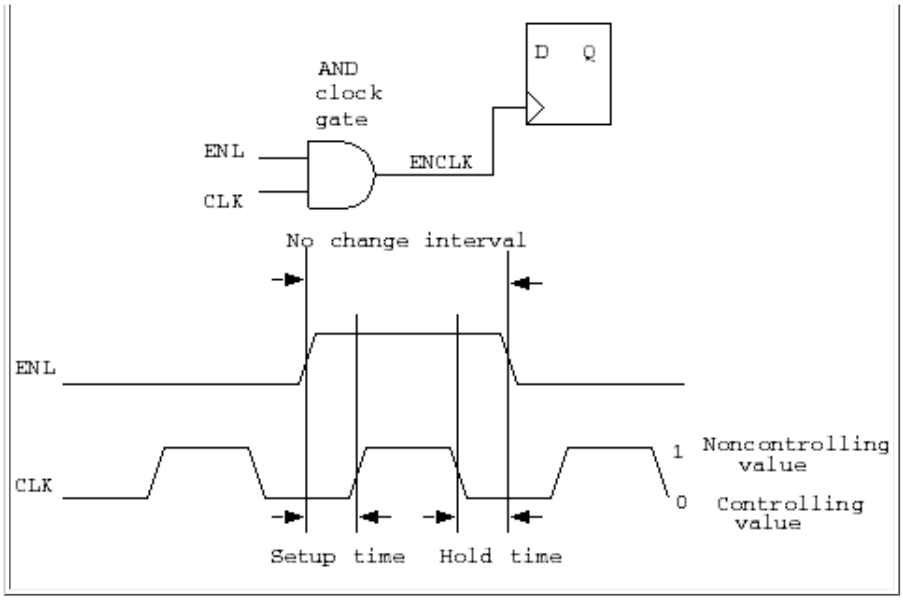
Clock-Gating Timing Considerations

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

Figure 9-7 and Figure 9-8 show the relationship of setup and hold times to a clock waveform. Figure 9-7 shows the relationship when an AND gate is the clock-gating element. Figure 9-8 shows the relationship when an OR gate is the clock-gating element.

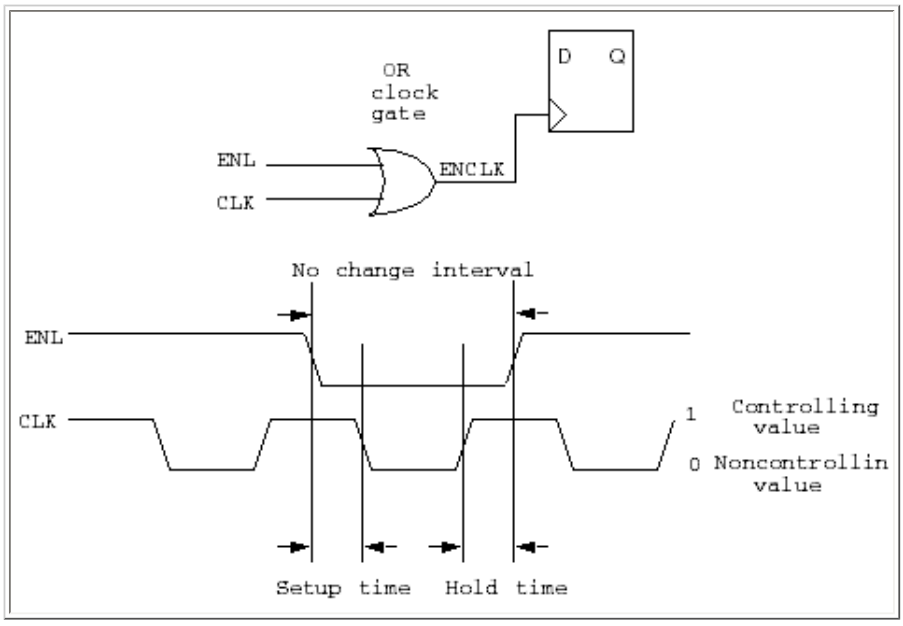
Figure 9-7 Setup and Hold Times for an AND Clock Gate





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 9-8 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 9-4 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 9-4 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

The following rules apply to the checking of timing arcs on integrated clock-gating cells:

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

Integrated Clock-Gating Cell Example

Example 9-5 shows what you might enter in setting up a cell for integrated clock gating. This example uses the `latch_posedge_precontrol_obs` value option for the `clock_gating_integrated_cell` attribute.

Example 9-5 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;
  }
}
```

9.9 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.

9.9.1 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.

9.9.2 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") ;
```

9.9.3 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 ;
```

9.9.4 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");
      }
   }
}
```

10. Timing Arcs

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

For information on describing delay, see these sections:

- Understanding Timing Arcs
- Modeling Method Alternatives
- Describing Three-State Timing Arcs
- Describing Edge-Sensitive Timing Arcs
- Describing Clock Insertion Delay
- Describing Intrinsic Delay
- Describing Transition Delay
- Modeling Load Dependency
- Describing Slope Sensitivity
- Describing State-Dependent Delays

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:

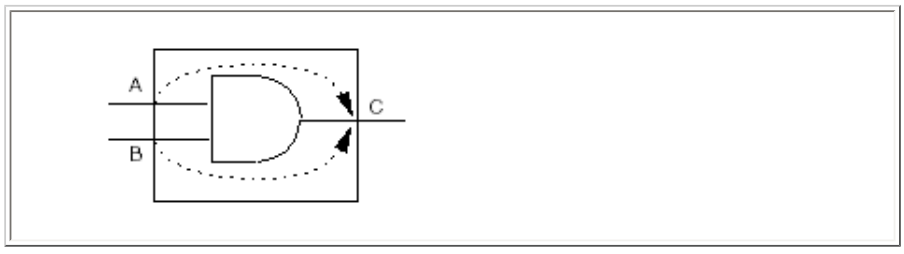
- Timing Arc Restrictions
- Examples of Libraries Using Delay Models
- Describing a Transparent Latch Clock Model
- Driver Waveform Support
- Sensitization Support
- Phase-Locked Loop Support

10.1 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 10-1 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 10-1 Timing Arcs



10.1.1 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” .

10.1.2 Sequential Timing Arcs

Sequential timing arcs describe the timing characteristics of sequential elements. In descriptions of the relationship between a clock transition and data output (input to output), the timing arc is considered a *delay* arc. In descriptions of the relationship between a clock transition and data input (input to input), the timing arc is considered a *constraint* arc.

A sequential timing arc is of one of the following types:

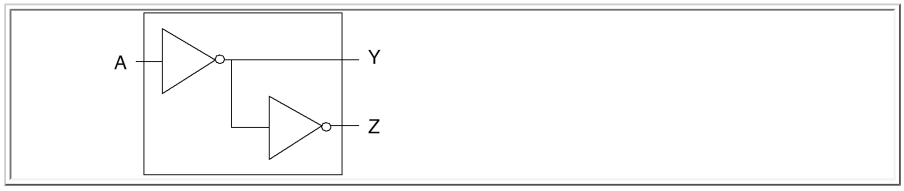
- Edge-sensitive (rising_edge or falling_edge)
- Preset or clear
- Setup or hold (setup_rising, setup_falling, hold_rising, or hold_falling)
- Nonsequential setup or hold (non_seq_setup_rising, non_seq_setup_falling, non_seq_hold_rising, non_seq_hold_falling)
- Recovery or removal (recovery_rising, recovery_falling, removal_rising, or removal_falling)
- No change (nochange_high_high, nochange_high_low, nochange_low_high, nochange_low_low)

For information on describing sequential timing types, see “timing Group Attributes” .

10.2 Modeling Method Alternatives

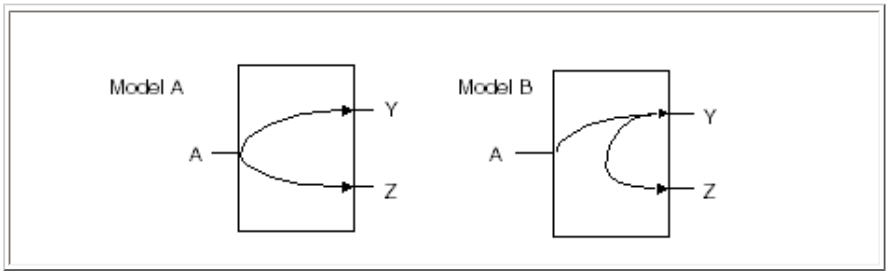
Timing information for combinational cells such as the one in Figure 10-2 can be modeled in one of two ways, as Figure 10-3 shows.

Figure 10-2 Two-Inverter Cell



In Figure 10-3, 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 10-3 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.

10.3 Defining timing Groups

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

`timing` groups describe

- Timing relationships between an input pin and an output pin
- Timing relationships between two output pins
- Timing arcs through a noncombinational element
- Setup and hold times on flip-flop or latch input
- Optionally, the names of the timing arcs

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

A `timing` group is defined in a `pin` group. This is the syntax:

```
library (lib_name) {
  cell (cell_name) {
    pin (pin_name) {
      timing () {
      ... timing description ...
      }
    }
  }
}
```

Define the `timing` group in the `pin` group of the endpoint of the timing arc, as illustrated by pin C in Figure 10-1.

10.3.1 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.

Example

```
...
bundle (Q){
  members (Q0, Q1, Q2, Q3);
  direction : output;
  function : "IQ";
  timing (G_Q0, G_Q1, G_Q2, G_Q3){
    timing_type : rising_edge;
    intrinsic_rise : 0.99;
    intrinsic_fall : 0.96;
    rise_resistance : 0.1458;
    fall_resistance : 0.0653;
    related_pin : "G";
  }
}
```

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

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

If G is another bundle of member size 4 and we assume that G0, G1, G2, and G3 are members of bundle G, the timing arcs are as follows:

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

Timing Arcs Between a Bundle and Multiple Related Pins

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

Example

```
bundle (Q){
  members (Q0, Q1, Q2, Q3);
  direction : output;
  function : "IQ";
  timing (G_Q0, H_Q0, G_Q1, H_Q1, G_Q2, H_Q2, G_Q3, H_Q3){
    timing_type : rising_edge;
    intrinsic_rise : 0.99;
    intrinsic_fall : 0.96;
    rise_resistance : 0.1458;
    fall_resistance : 0.0653;
    related_pin : "G H";
  }
}
```

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

From pin	To pin	Label
G	Q0	G_Q0
H	Q0	H_Q0
G	Q1	G_Q1
H	Q1	H_Q1
G	Q2	G_Q2
H	Q2	H_Q2
G	Q3	G_Q3
H	Q3	H_Q3

If G was another bundle of member size 4 and we assume that G0, G1, G2, and G3 are members of bundle G, the timing arcs are as follows:

From pin	To pin	Label
G0	Q0	G_Q0
H	Q0	H_Q0
G1	Q1	G_Q1
H	Q1	H_Q1

G2	Q2	G_Q2
H	Q2	H_Q2
G3	Q3	G_Q3
H	Q3	H_Q3

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

Timing Arcs Between a Bus and a Single Related Pin

This section describes how to identify the timing arcs created when a `timing` group is within a `bus` group that has several bits with the same single related pin. You identify the resulting multiple timing arcs by entering a name list with the `timing` group.

Example

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

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

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

If B is another 4-bit bus and we assume that B[0] is the MSB for bus B, the timing arcs are as follows:

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

Timing Arcs Between a Bus and Multiple Related Pins

This section describes the timing arcs created when a `timing` group is within a `bus` group that has several bits, where each bit has its own related pin. You identify the resulting multiple timing arcs by entering a name list with the `timing` group.

Example

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

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

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

If B is another 4-bit bus and we assume that B[0] is the MSB for bus B, the timing arcs are as follows:

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

C	X[2]	C_X2
B[3]	X[3]	B_X3
C	X[3]	C_X3

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

Timing Arcs Between a Bus and Related Bus Pins

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

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

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:

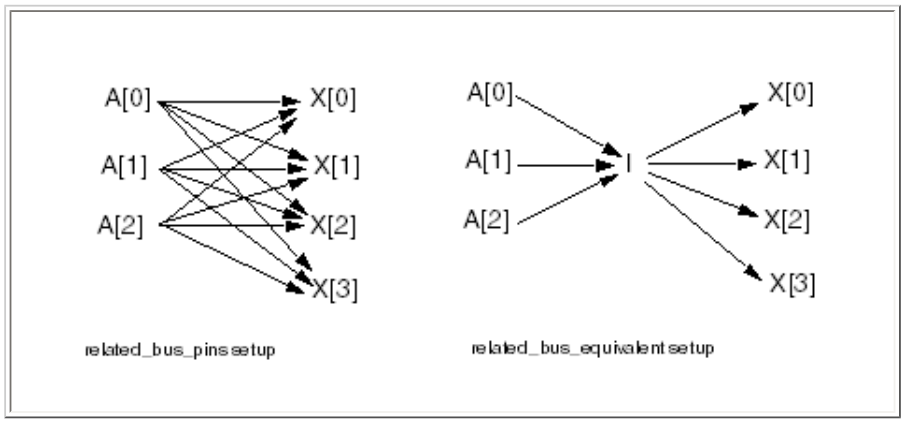
From pin	To pin	Label
B[0]	X[0]	B0_X0
B[0]	X[1]	B0_X1
B[0]	X[2]	B0_X2
B[0]	X[3]	B0_X3
B[1]	X[0]	B1_X0
B[1]	X[1]	B1_X1
B[1]	X[2]	B1_X2
B[1]	X[3]	B1_X3

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 10-4 compares the setup created with the `related_bus_pins` attribute with a setup created with the `related_bus_equivalent` attribute.

Figure 10-4 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.

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

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

Note:

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

Example

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

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

From pin	To pin	Label
A[0]	I	A0_I
A[1]	I	A1_I

A[2]	I	A2_I
I	X[0]	I_X0
I	X[1]	I_X1
I	X[2]	I_X2
I	X[3]	I_X3

10.3.2 Delay Models

`timing` groups are defined by the `timing` group attributes. The delay model you use determines which set of delay calculation attributes you specify in a `timing` group.

The following delay models are supported:

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.

Syntax

delay_model : $value_{enum}$;

value

Valid values are `generic_cmos`, `table_lookup` (nonlinear delay model), `piecewise_cmos`, `dcm` (delay calculation module), and `polynomial` (scalable polynomial delay model).

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");
}
```

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 10-1 lists the combinations of values you can assign to the different variables for the varying dimensional tables specifying timing delays.

Table 10-1 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 10-2 lists the combination of values you can assign to the different variables for the varying dimensional tables specifying load-dependent constraints.

Table 10-2 Variable Values for Load-Dependent Constraint Tables

Template dimension	Variable_1	Variable_2	Variable_3
1	set1		
1	set2		
2	set1	set2	
2	set2	set1	
3	set1	set2	set3
3	set1	set3	set2
3	set2	set1	set3
3	set2	set3	set1
3	set3	set1	set2

3	set3	set2	set1

Template Breakpoints

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

The index values are lists of floating-point numbers greater than or equal to 0.0. The values in the list must be in increasing order. The size of each dimension is determined by the number of floating-point numbers in the indexes.

You must define at least one `index_1` in the `lu_table_template` group. For a one-dimensional table, use only `variable_1`.

Creating Lookup Tables

The rules for specifying lookup tables apply to delay arcs as well as to constraints. "Defining Delay Arcs With Lookup Tables" 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 10-3.
- 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 10-3 shows a library that uses the CMOS nonlinear delay model to describe the delay.

Defining the Scalable Polynomial Delay Model Template

Polynomial templates store common format information that equations can use.

poly_template Group

You use a `poly_template` group to specify the equation variables, the variable ranges, the voltage mapping, and the piecewise data. Assign each `poly_template` group a unique name, so that equations in the `timing` group can refer to it.

Syntax

poly_template(*name*$_{id}$) {
  variables(*variable_i*$_{enum}$, ..., *variable_n*$_{enum}$)
  variable_i_range(*float*, *float*)
  ...
  variable_n_range(*float*, *float*)
  mapping(*voltage*$_{enum}$, *power_rail*$_{id}$)
  domain(*domain_name*$_{id}$) {
    calc_mode : *name*$_{id}$;
    variables(*variable_i*$_{enum}$)..., *variable_n*$_{enum}$)
    variable_i_range(*float*, *float*)
      ...
    variable_n_range(*float*, *float*)
    mapping(*voltage*$_{enum}$, *power_rail*$_{id}$)
  }
}

poly_template Variables

The `poly_template` group that defines timing delays can have up to *n* variables (variable_ *i* , ..., variable_ *n*), which you specify in the `variables` complex attribute. The variables you specify represent the following in the equation:

- The input transition time of a constrained pin
- The output net length and capacitance
- The output loading of a related pin
- The default power supply voltage
- The voltage*i* for multivoltage cells
- The temperature
- User parameters (parameter1...parameter5)

The following list shows the valid values, divided into four sets, that you can assign to variables in an equation:

- Set 1:
  input_net_transistion constrained_pin_transition
- Set 2:
  total_output_net_capacitance output_net_length output_net_wire_cap, output_net_pin_cap related_pin_transistion
- Set 3:
  related_out_total_output_net_capacitance related_out_output_net_length related_out_output_net_wire_cap related_out_output_net_pin_cap
- Set 4:
  temperature voltage*i* parameter *n*

delay_model Simple Attribute

Use the `delay_model` attribute in the `library` group to specify the scalable polynomial delay model.

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

10.3.3 timing Group Attributes

The delay model you use determines which set of attributes for delay model calculation you specify in a `timing` group. Table 10-3 shows the available delay models and the supported `timing` group attributes for each. See Chapter , “,” for detailed information about the delay models.

Table 10-3 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`
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.

Syntax

related_pin : "*name1* [*name2 name3* ...]" ;

In a cell with input pin A and output pin B, define A and its relationship to B in the `related_pin` attribute statement in the `timing` group that describes pin B.

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.

Syntax

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

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.

Syntax

```
timing_sense : positive_unate | negative_unate
| non_unate ;
```

positive_unate

Combines incoming rise delays with local rise delays and compares incoming fall delays with local fall delays.

negative_unate

Combines incoming rise delays with local fall delays and compares incoming fall delays with local rise delays.

non_unate

Combines local delays with the worst-case incoming delay value. The non-unate timing sense represents a function whose output value change cannot be determined from the direction of the change in the input value.

Example

```
timing ( ) {
  timing_sense : positive_unate;
}
```

A function is said to be *unate* if a rising (or falling) change on a positive (or negative) unate input variable causes the output function variable to rise (or fall) or not change. For a non-unate variable, further state information is required to determine the effects of a particular state transition.

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.

Syntax

```
timing_type : 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 | rising_edge | falling_edge |
   preset | clear | hold_rising | hold_falling |
   setup_rising | setup_falling | recovery_rising |
   recovery_falling | skew_rising | skew_falling |
   removal_rising | removal_falling | min_pulse_width |
   minimum_period | max_clock_tree_path |
   min_clock_tree_path |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 ;
```

The following sections show the `timing_type` attribute values for the following types of timing arcs:

- Combinational
- Sequential
- Nonsequential
- No-change

Values for Combinational Timing Arcs

The timing type and timing sense define the signal propagation pattern. The default timing type is combinational.

Timing type		**Timing sense**	
positive_unate	**negative_unate**	**non_unate**	
`combinational`	R->R,F->F	R->F,F->R	{R,F}->{R,F}
`combinational_rise`	R->R	F->R	{R,F}->R
`combinational_fall`	F->F	R->F	{R,F}->F
`three_state_disable`	R->{0Z,1Z}	F->{0Z,1Z}	{R,F}->{0Z,1Z}
`three_state_enable`	R->{Z0,Z1}	F->{Z0,Z1}	{R,F}->{Z0,Z1}
`three_state_disable_rise`	R->0Z	F->0Z	{R,F}->0Z
`three_state_disable_fall`	R->1Z	F->1Z	{R,F}->1Z
`three_state_enable_rise`	R->Z1	F->Z1	{R,F}->Z1
`three_state_enable_fall`	R->Z0	F->Z0	{R,F}->Z0

Values for Sequential Timing Arcs

You use sequential timing arcs to model the timing requirements for sequential cells.

rising_edge

Identifies a timing arc whose output pin is sensitive to a rising signal at the input pin.

falling_edge

Identifies a timing arc whose output pin is sensitive to a falling signal at the input pin.

preset

Preset arcs affect only the rise arrival time of the arc's endpoint pin. A preset arc implies that you are asserting a logic 1 on the output pin when the designated related pin is asserted.

clear

Clear arcs affect only the fall arrival time of the arc's endpoint pin. A clear arc implies that you are asserting a logic 0 on the output pin when the designated related pin is asserted.

hold_rising

Designates the rising edge of the related pin for the hold check.

hold_falling

Designates the falling edge of the related pin for the hold check.

setup_rising

Designates the rising edge of the related pin for the setup check on clocked elements.

setup_falling

Designates the falling edge of the related pin for the setup check on clocked elements.

recovery_rising

Uses the rising edge of the related pin for the recovery time check. The clock is rising-edge-triggered.

recovery_falling

Uses the falling edge of the related pin for the recovery time check. The clock is falling-edge-triggered.

skew_rising

The timing constraint interval is measured from the rising edge of the reference pin (specified in `related_pin`) to a transition edge of the parent pin in the `timing` group. The `intrinsic_rise` value is the maximum skew time between the reference pin rising and the parent pin rising. The `intrinsic_fall` value is the maximum skew time between the reference pin falling and the parent pin falling.

skew_falling

The timing constraint interval is measured from the falling edge of the reference pin (specified in `related_pin`) to a transition edge of the parent pin in the `timing` group. The `intrinsic_rise` value is the maximum skew time between the reference pin falling and the parent pin rising. The `intrinsic_fall` value is the maximum skew time between the reference pin falling and the parent pin falling.

removal_rising

Used when the cell is a low-enable latch or a rising-edge-triggered flip-flop. For active-low asynchronous control signals, define the removal time with the `intrinsic_rise` attribute. For active-high asynchronous control signals, define the removal time with the `intrinsic_fall` attribute.

removal_falling

Used when the cell is a high-enable latch or a falling-edge-triggered flip-flop. For active-low asynchronous control signals, define the removal time with the `intrinsic_rise` attribute. For active-high asynchronous control signals, define the removal time with the `intrinsic_fall` attribute.

minimum_pulse_width

This value, together with the `minimum_period` value, lets you specify the minimum pulse width for a clock pin. The timing check is performed on the pin itself, so the related pin should be the same. You can also include rise and fall constraints, as with other timing checks.

minimum_period

This value, together with the `minimum_pulse_width` value, lets you specify the minimum pulse width for a clock pin. The timing check is performed on the pin itself, so the related pin should be the same. You can also include rise and fall constraints as with other timing checks.

max_clock_tree_path

Used in `timing` groups under a clock pin. Defines the maximum clock tree path constraint.

min_clock_tree_path

Used in `timing` groups under a clock pin. Defines the minimum clock tree path constraint.

Values for Nonsequential Timing Arcs

In some nonsequential cells, the setup and hold timing constraints are specified on the data pin with a nonclock pin as the related pin. The signal of a pin must be stable for a specified period of time before and after another pin of the same cell range state for the cell to function as expected.

non_seq_setup_rising

Defines (with `non_seq_setup_falling`) the timing arcs used for setup checks between pins with nonsequential behavior. The related pin in a timing arc is used for the timing check.

non_seq_setup_falling

Defines (with `non_seq_setup_rising`) the timing arcs used for setup checks between pins with nonsequential behavior. The related pin in a timing arc is used for the timing check.

non_seq_hold_rising

Defines (with `non_seq_hold_falling`) the timing arcs used for hold checks between pins with nonsequential behavior. The related pin in a timing arc is used for the timing check.

non_seq_hold_falling

Defines (with `non_seq_hold_rising`) the timing arcs used for hold checks between pins with nonsequential behavior. The related pin in a timing arc is used for the timing check.

Values for No-Change Timing Arcs

You use no-change timing arcs to model the timing requirement for latch devices with latch-enable signals. The four no-change timing types define the pulse waveforms of both the constrained signal and the related signal in 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);
```

Example

```
timing() {
  mode(rw, read);
}
```

Example 10-1 shows a `mode` instance description.

Example 10-1 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 10-2 shows multiple `mode` descriptions.

Example 10-2 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;
 }
}
```

10.4 Describing Three-State Timing Arcs

Three-state arcs describe a three-state output pin in a cell.

10.4.1 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” for more information.

10.4.2 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" for more information.

10.5 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" 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

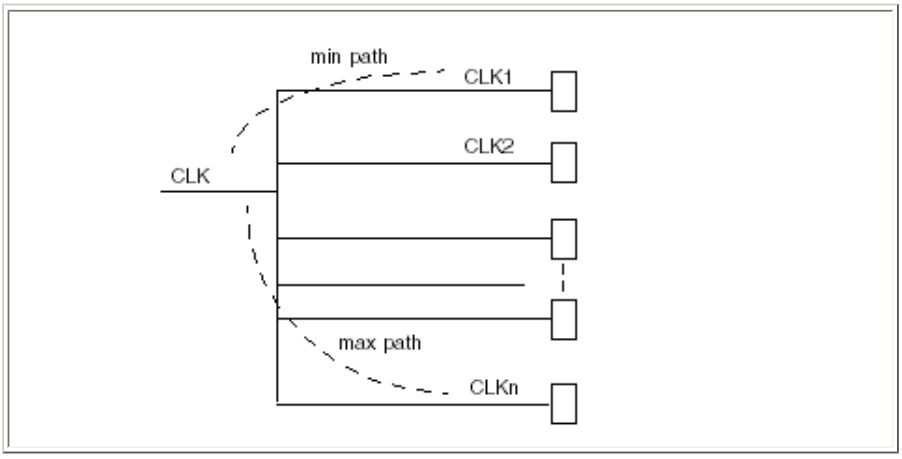
```
pin(QN) {
  direction : output ;
  function : "IQN" ;
  timing() {
    related_pin : "CP" ;
    timing_type : rising_edge ;
    intrinsic_rise : 1.29 ;
    intrinsic_fall : 1.61 ;
    rise_resistance : 0.1723 ;
    fall_resistance : 0.0553 ;
  }
}
```

The QN pin makes a transition after the clock signal rises.

10.6 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 transisiton, as shown in Figure 10-5.

Figure 10-5 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.

10.7 Describing Intrinsic Delay

The intrinsic delay of an element is the zero-load (fixed) component of the total delay equation. Intrinsic delay attributes have different meanings, depending on whether they are for an input or an output pin.

When describing an output pin, the intrinsic delay attributes define the fixed delay from input to output pin. These values are used to calculate the intrinsic delay of the total delay equation.

When describing an input pin, such as in a setup or hold timing arc, intrinsic attributes define the timing requirements for that pin. Pin D in Example 10-11 illustrates the intrinsic delay attributes used as timing constraints. Timing constraints are not used in the delay equation.

10.7.1 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.

10.7.2 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. .

10.7.3 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" to find out how to create and use templates for lookup tables in a library, using the CMOS nonlinear delay model.

10.7.4 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" to learn how to create and use templates for scalable polynomials in a library, using the

CMOS scalable polynomial nonlinear delay model.

10.8 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.

10.8.1 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.

10.8.2 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.

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.

10.8.3 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. .

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`

Example 10-3 shows the use of lookup tables and templates for describing cell delay with the CMOS nonlinear delay model.

Example 10-3 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_dimensonal) {

          /*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.

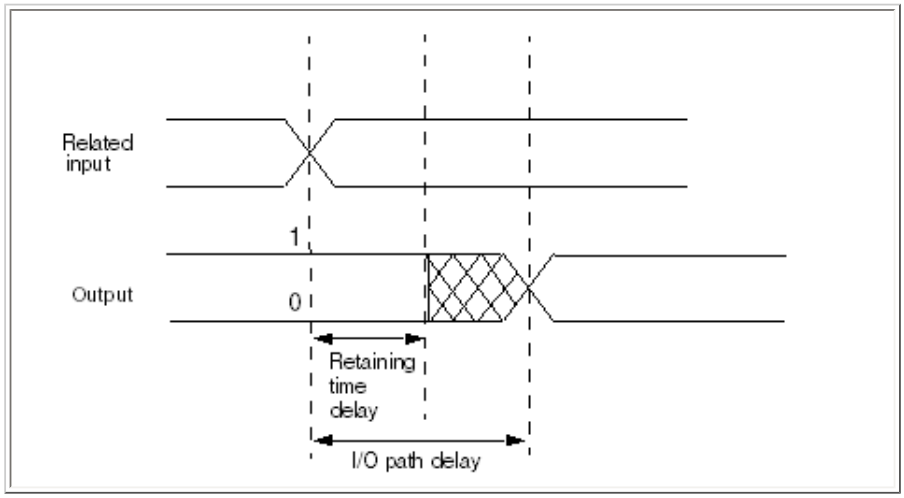
The value you enter for the `retaining_fall` attribute determines how long the output will retain its current value, 1, after the value at the related input port has changed.

Note:

Retaining time works only on a nonlinear delay model.

Figure 10-6 shows retaining delay in regard to changes in a related input port.

Figure 10-6 Retaining Time Delay



Example 10-4 shows how to use the `retaining_rise` and `retaining_fall` attributes.

Example 10-4 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" for information about calculating delay factors. For information about including propagation delay in total cell delay calculations, see "".

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.

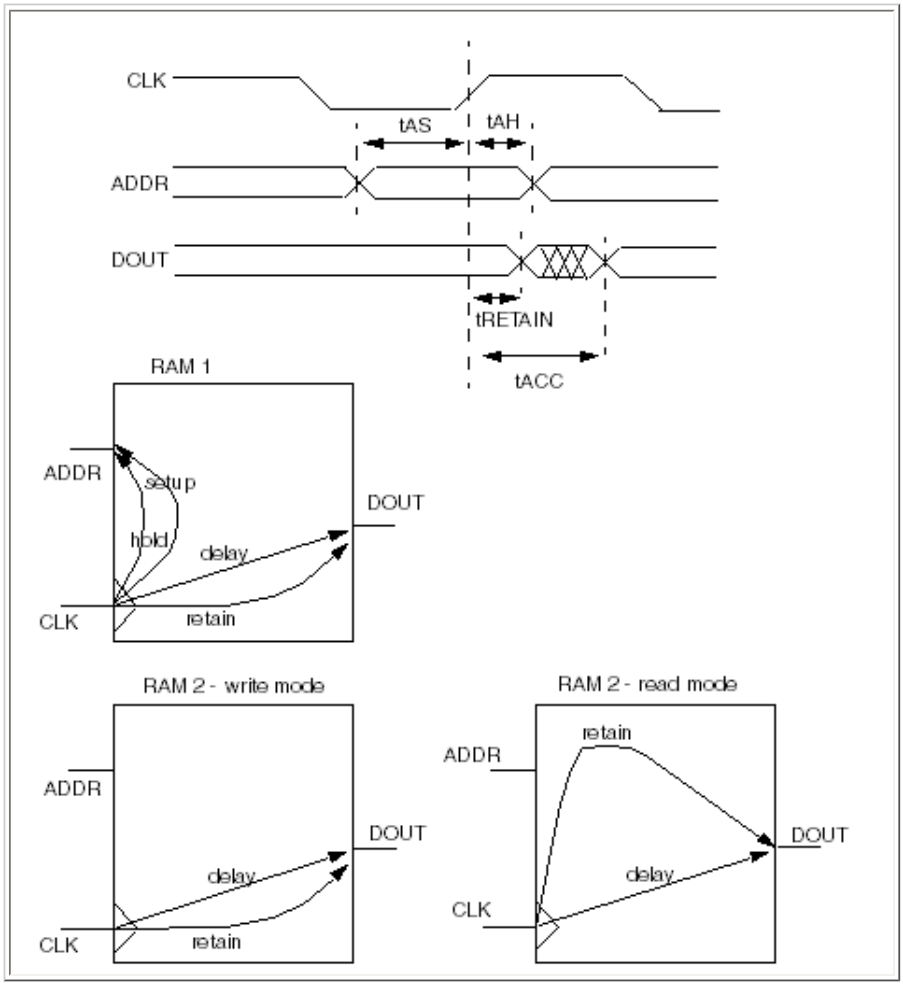
The value you enter for the `retain_fall_slew` attribute determines how long the output will retain its current value, 1, after the value at the related input port has changed.

Note:

Retaining time works only on a nonlinear delay model.

Figure 10-7 shows a timing diagram of synchronous RAM.

Figure 10-7 Timing Diagram of Synchronous RAM



Example

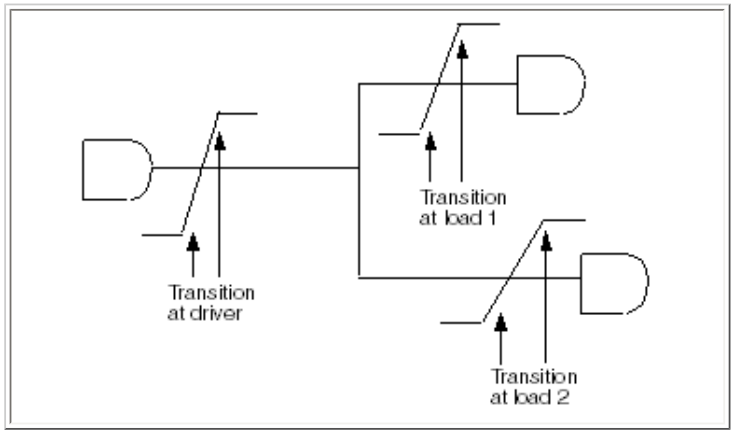
```
library(library_name) {
 ...
 lu_table_template (retaining_table_template){
   ...
   variable_1: total_output_net_capacitance;
   variable_2: input_net_transition;
   index_1 ("0.0, 1.5");
   index_2 ("1.0, 2.1");
 }
 ...
 cell (cell_name){
   ...
   pin (A) {
     direction : output;
     ...
     timing(){
       related_pin : "B"
       ...
       retaining_rise (retaining_table_template){
         values ("0.00, 0.23", "0.11, 0.28");
       }
       retaining_fall (retaining_table_template){
         values ("0.01, 0.30", 0.12, 0.18");
       }
       retain_rise_slew(retaining_time_template){
         values("0.01,0.02");
       }
       retain_fall_slew(retaining_time_template){
         values("0.01,0.02");
       }
```

```
    }/*end of pin() */
    ...
  }/*end of cell() */
    ...
}/*end of library() */
```

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 10-8. The higher the interconnect load, the greater the flattening effect and the greater the transition delay.

Figure 10-8 Transition Time Degradation



To model the degradation of the transition time as it propagates from an output pin over the net to the next input pin, include these library-level groups in your library:

- `rise_transition_degradation`
- `fall_transition_degradation`

These groups contain the values describing the degradation functions for rise and fall transitions in the form of lookup tables. The lookup tables are indexed by

- Transition time at the net driver
- Connect delay between the driver and a load

These are the supported values for transition degradation (`variable_1` and `variable_2`):

- output_pin_transition
- connect_delay

You can assign either `connect_delay` or `output_pin_transition` to `variable_1` or `variable_2`, 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))

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 10-5 shows the use of the degradation tables. In this example, `trans_deg` is the name of the template for the transition degradation.

Example 10-5 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) {
....
}
```

10.9 Modeling Load Dependency

"Describing Transition Delay" 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.

10.9.1 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` attributevalue.
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" .
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" .) The following groups are valid lookup tables for output load modeling:
   - rise_constraint
   - fall_constraint

   See "Setting Setup and Hold Constraints" for information about these groups.

Example of a Load-Dependent Model

Example 10-6 is an example of a library that includes a load-dependent model.

Example 10-6 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");
      }
```

```
    }
    ...
  }
  ...
}
```

10.9.2 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` attributevalue.
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" .
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" for information about these groups.

Example of a Load-Dependent Model

Example 10-7 is an example of a library that includes a load-dependent model.

Example 10-7 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 */
```

10.10 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.

10.10.1 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;
```

10.11 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.

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.

10.11.1 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 10-4 for the valid Boolean operators.

Table 10-4 Valid Boolean Operators

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

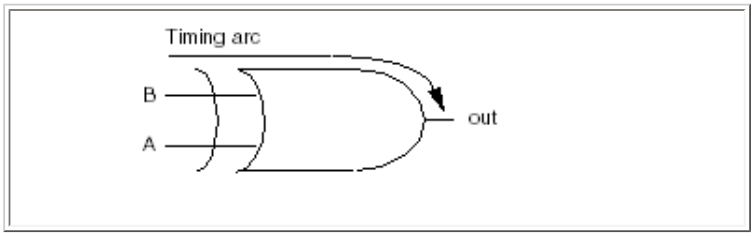
The order of precedence of the operators is left to right, with inversion performed first, then XOR, then AND, then OR.

Example

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

Figure 10-9 shows an XOR gate.

Figure 10-9 XOR Gate With State-Dependent Timing Arc



Example 10-8 shows how to use the when attribute for an XOR gate. In the description of the XOR cell, pin A sets conditional timing for the output pin “out” when you define the timing arc for related_pin B. In this example, when you set conditional timing for related_pin A with the when : “B” statement, the output pin gets the negative_unate value of A when the condition is B.

There are limitations on the pin types that can be used with different types of cells with the when attribute.

For a combinational cell, these pins are valid with the when attribute:

- Pins in the function attribute for regular combinational timing arcs
- Pins in the three_state attribute for the endpoint of the timing arc

For a sequential cell, valid pins or variables to use with the when attribute are determined by the timing type of the arc.

- For timing types rising_edge and falling_edge: Pins in these attributes are allowed in the when attribute:
  - next_state
  - clocked_on
  - clocked_on_also
  - enable
  - data_in
- For timing type clear
  - If the pin's function is the first state variable in the flip-flop or latch group, the pin that defines the clear condition in the flip-flop or latch group is allowed in the when construct.
  - If the pin's function is the second state variable in the flip-flop or latch group, the pin that defines the preset condition in the flip-flop or latch group is allowed in the when construct.
- For timing type preset
  - If the pin's function is the first state variable in the flip-flop or latch group, the pin that defines the preset condition in the flip-flop or latch group is allowed in the when construct.
  - If the pin's function is the second state variable in the flip-flop or latch group, the pin that defines the clear condition in the flip-flop or latch group is allowed in the when construct.

See "timing_type Simple Attribute" for more information.

All input pins in a black box cell (a cell without a function attribute) are allowed in the when attribute.

10.11.2 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.

Example 10-8 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 10-8 XOR Cell With State-Dependent Timing

```
cell(XOR) {
  pin(A) {
    direction : input;
  ...
  }
  pin(B) {
    direction : input;
  ...
  }
  pin(out) {
    direction : output;
    function : "A ^ B";
    timing() {
      related_pin : "A";
      timing_sense : negative_unate;
```

```
      when : "B";
      sdf_cond : " B == 1'B1 ";
      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;
    }
  }
}
```

10.12 Setting Setup and Hold Constraints

Signals arriving at an input pin have ramp times. Therefore, you must ensure that the data signal has stabilized before latching its value by defining setup and hold arcs as timing requirements.

- Setup constraints describe the minimum time allowed between the arrival of the data and the transition of the clock signal. During this time, the data signal must remain constant. If the data signal makes a transition during the setup time, an incorrect value 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 10-10 shows setup and hold timing for a rising-edge-triggered flip-flop. The timing checks for flip-flops use the activating edge of the clock, which is the rising edge in this case.

Figure 10-10 Setup and Hold Constraints for Rising-Edge-Triggered Flip-Flop

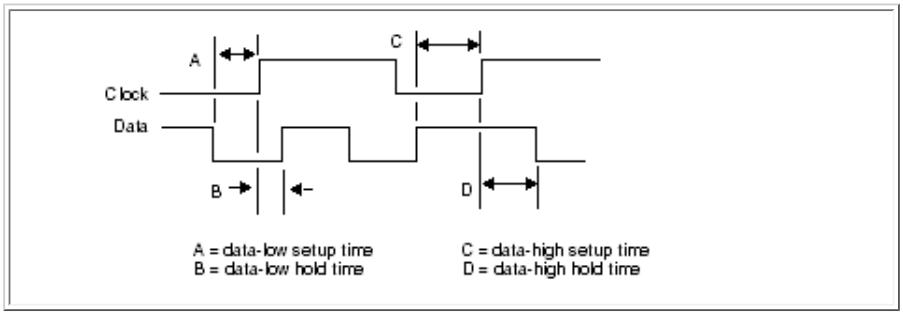
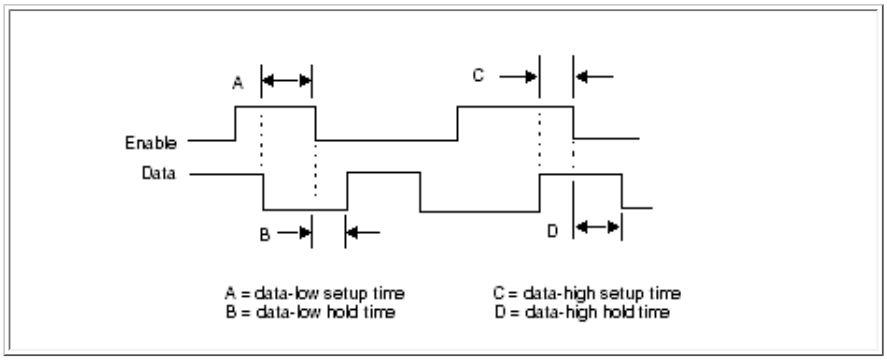


Figure 10-11 illustrates setup and hold timing for a high-enable latch. The timing checks for latches generally use the deactivating edge of the enable signal, which is the falling edge in this case. However, the method used depends on the vendor.

Figure 10-11 Setup and Hold Constraints for High-Enable Latch



10.12.1 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.

Example

This example describes the setup arc for the data pin on the rising-edge-triggered D flip-flop shown in Figure 10-10. 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 10-11. 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" ;
}
```

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.

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 10-10, 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 10-11, 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" ;
  }
}
```

10.12.2 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" and "Creating Lookup Tables".

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 10-9 shows how to use tables to specify setup constraints for a flip-flop.

Example 10-9 CMOS Nonlinear Timing Model Using Constraint

```
library( vendor_b ) {

  /* 1. Use delay lookup table */
  delay_model : table_lookup;

  /* 2. Define template of size 3 x 3*/
  lu_table_template(constraint_template) {
    variable_1 : constrained_pin_transition;
    variable_2 : related_pin_transition;
    index_1 ("0.0, 0.5, 1.5");
    index_2 ("0.0, 2.0, 4.0");
  }
  . . .
  cell(dff) {
    pin(d) {
      direction: input;
      timing() {
        related_pin : "clk";
        timing_type : setup_rising;

        /* Inherit the constraint_template template */
        rise_constraint(constraint_template) {

          /* Specify all the values */
          values ("0.0, 0.13, 0.19", \
          "0.21, 0.23, 0.41", \
           "0.33, 0.37, 0.50");
        }
        fall_constraint(constraint_template) {
          values ("0.0, 0.14, 0.20", \
           "0.22, 0.24, 0.42", \
            "0.34, 0.38, 0.51");
        }
      }
      . . .
    }
  }
}
```

10.12.3 In the Scalable Polynomial Delay Model

Example 10-10 shows how to specify constraint in a scalable polynomial delay model.

Example 10-10 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);
      }
    }
    .......
  }
 }
.....
}
```

10.12.4 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 Liberty Reference Manual.

10.13 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.

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.

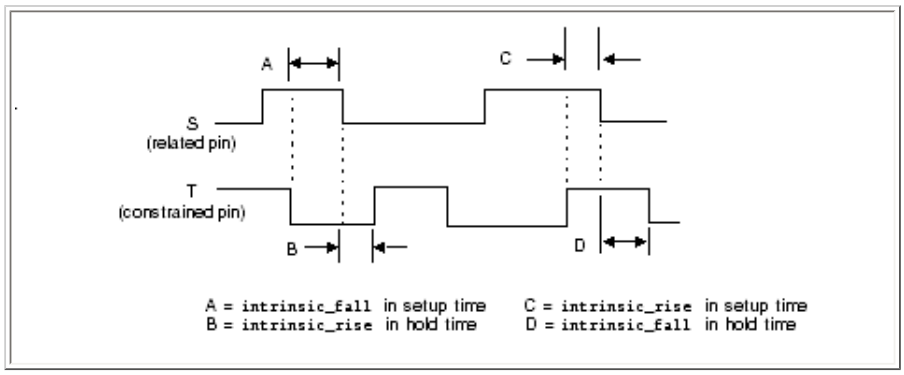
   You can use both rising and falling edges as the active edge of the related pin for one cell.

Example

```
pin(T) {
  timing() {
    timing_type : non_seq_setup_falling;
    intrinsic_rise : 1.5;
    intrinsic_fall : 1.5;
    related_pin : "S";
  }
}
```

Figure 10-12 shows the waveforms for the nonsequential timing arc described in the preceding example. In this timing arc, the constrained pin is T and its related pin is S. The intrinsic rise value describes setup time C or hold time B. The intrinsic fall value describes setup time A or hold time D.

Figure 10-12 Nonsequential Setup and Hold Constraints



10.14 Setting Recovery and Removal Timing Constraints

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

10.14.1 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 10-13 shows the recovery timing arc for a rising-edge-triggered flip-flop with active-low clear.

Figure 10-14 shows the recovery timing arc for a low-enable latch with active-high preset.

Figure 10-13 Recovery Timing Constraint for a Rising-Edge-Triggered Flip-Flop

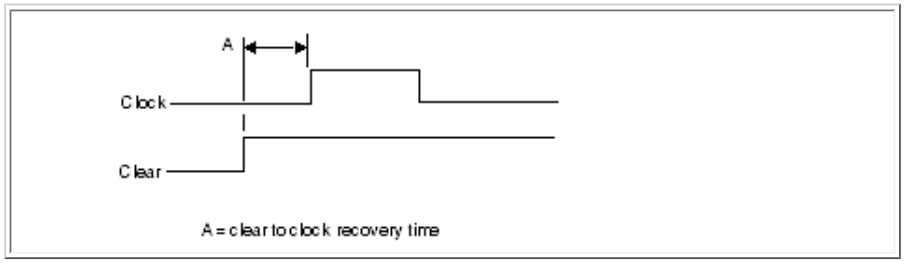
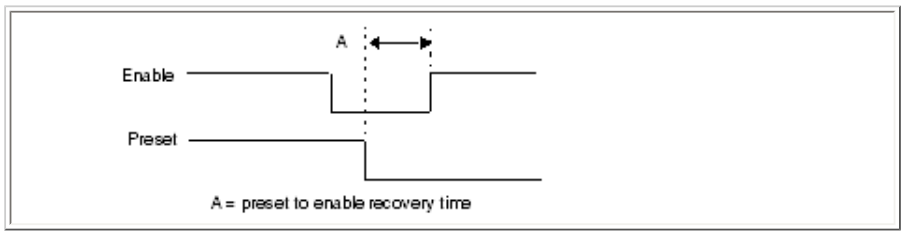


Figure 10-14 Recovery Timing Constraint for a Low-Enable Latch



The values you can assign to a `timing_type` attribute to define a recovery time constraint are

`recovery_rising`

Uses the rising edge of the related pin for the recovery time check; the clock is rising-edge-triggered.

`recovery_falling`

Uses the falling edge of the related pin for the recovery time check; the clock is falling-edge-triggered.

To define a recovery time constraint for an asynchronous control pin,

1. Assign a value to the `timing_type` attribute.

   Use `recovery_rising` for rising-edge-triggered flip-flops and low-enable latches; use `recovery_falling` for negative- edge-triggered flip-flops and high-enable latches.
2. Identify the synchronous clock pin as the `related_pin`.

   For active-low control signals, define the recovery time with the `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 10-13.

```
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 10-14.

```
pin (Preset) {
  direction : input ;
    capacitance : 1 ;
    timing() {
```

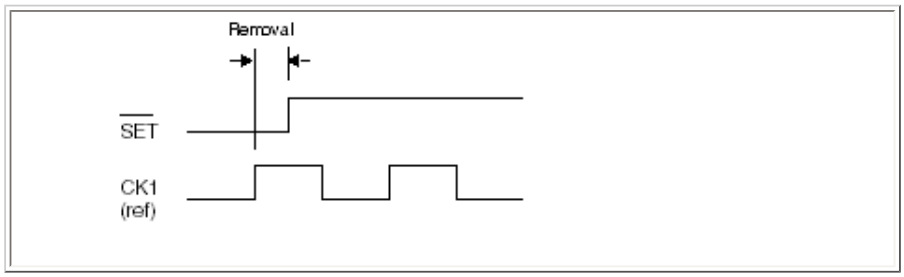
```
    related_pin : "Enable" ;
    timing_type : recovery_rising;
    intrinsic_fall : 1.0 ;
  }
}
```

10.14.2 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 10-15).

Figure 10-15 Timing Diagram for Removal Constraint



The values you can assign to a `timing_type` attribute to define a removal constraint are

`removal_rising`

Use when the cell is a low-enable latch or a rising-edge-triggered flip-flop.

`removal_falling`

Use when the cell is a high-enable latch or a falling-edge-triggered flip-flop.

To define a removal constraint,

1. Assign a value to the `timing_type` attribute.
2. Identify the synchronous clock pin as the `related_pin`.
3. For active-low asynchronous control signals, define the removal time with the `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 ;
    }
}
```

10.15 Setting No-Change Timing Constraints

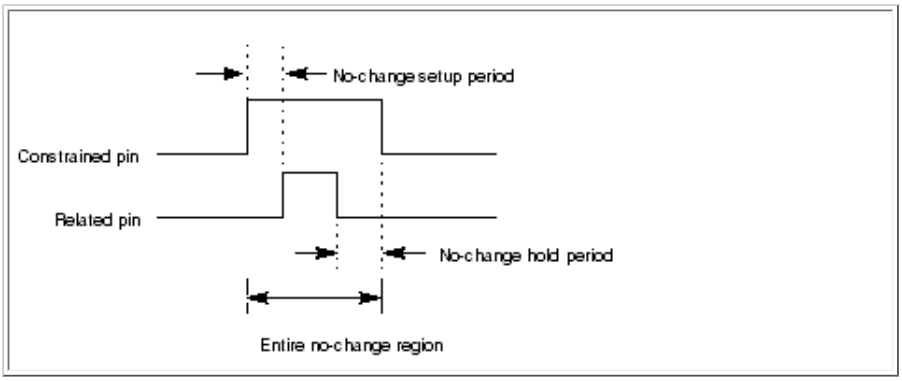
A no-change timing check checks a constrained signal against a level-sensitive related signal. The constrained signal must remain stable during an established setup period, for the width of the related pulse, and during an established hold period.

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 10-16 shows a no-change timing check between a constrained pin and its level-sensitive related pin.

Figure 10-16 No-Change Timing Check



The values you can assign to a `timing_type` attribute to define a no-change timing constraint are

`nochange_high_high`

Specifies a positive pulse on the constrained pin and a positive pulse on the related pin.

`nochange_high_low`

Specifies a positive pulse on the constrained pin and a negative pulse on the related pin.

`nochange_low_high`

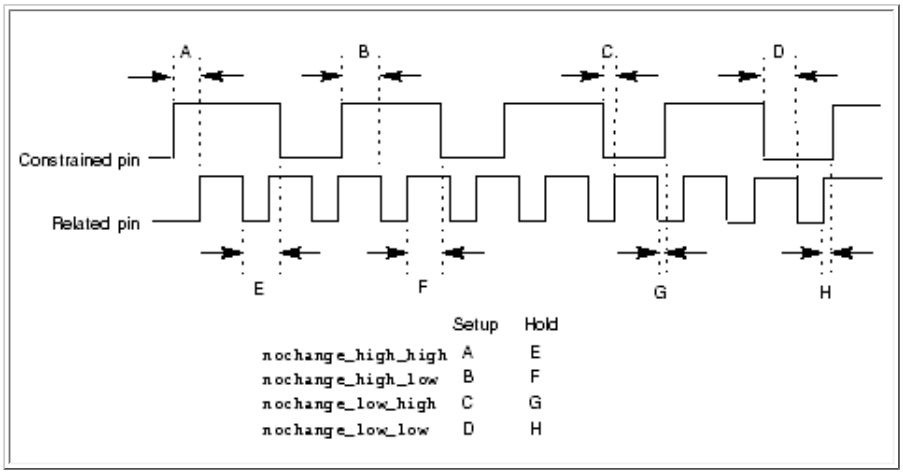
Specifies a negative pulse on the constrained pin and a positive pulse on the related pin.

`nochange_low_low`

Specifies a negative pulse on the constrained pin and a negative pulse on the related pin.

Figure 10-17 shows the waveforms for these constraints.

Figure 10-17 No-Change Setup and Hold Constraint Waveforms



To model no-change timing constraints,

1. Assign a value to the `timing_type` attribute.
2. Specify a related pin with the `related_pin` attribute in the `timing` group.

The 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 10-5.

   Note:

   With no-change timing constraints, conditional timing constraints have different interpretations than they do with other constraints. See "Setting Conditional Timing Constraints" for more information.

Table 10-5 Summary of No-Change Constraints and Delay Model Attributes

No-change constraint	Setup attribute for generic delay and nonlinear delay models	Hold attribute for generic delay and nonlinear delay models
nochange_high_high	`intrinsic_rise` / `rise_constraint`	`intrinsic_fall`/ `fall_constraint`
nochange_high_low	`intrinsic_rise` / `rise_constraint`	`intrinsic_fall`/ `fall_constraint`
nochange_low_high	`intrinsic_fall` / `fall_constraint`	`intrinsic_rise` / `rise_constraint`
nochange_low_low	`intrinsic_fall` / `fall_constraint`	`intrinsic_rise` / `rise_constraint`

10.15.1 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 */
}
```

10.15.2 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);
          }
        }
      ...
      }
     ...
    }
  ...
}
```

10.15.3 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 */
```

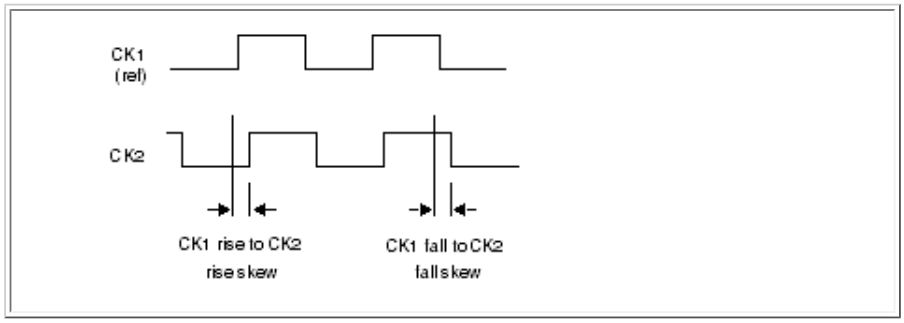
```
            orders ("1, 1");
            coefs ("0.2487 +0.0520 -0.0268 -0.0053 " );
            variable_1_range (0.01, 3.00);
            variable_2_range (0.01, 3.00);
           }
          fall_constraint (poly_template) { /* hold time */
            orders ("1, 1");
            coefs ("0.2487 +0.0520 -0.0268 -0.0053 " );
            variable_1_range (0.01, 3.00);
            variable_2_range (0.01, 3.00);
          }
        }
      ...
       }
     ...
    }
  ...
 }
```

10.16 Setting Skew Constraints

The skew constraint defines the maximum separation time allowed between two clock signals.

Figure 10-18 is a timing diagram showing skew constraint.

Figure 10-18 Timing Diagram for Skew Constraint



The values you can assign to a `timing_type` attribute to define a skew constraint are

skew_rising

The timing constraint interval is measured from the rising edge of the reference pin (specified in `related_pin`) to a transition edge of the parent pin of the `timing` group.

skew_falling

The timing constraint interval is measured from the falling edge of the reference pin (specified in `related_pin`) to a transition edge of the parent pin of the `timing` group.

To set skew constraint,

1. Assign a value to the `timing_type` attribute.
2. Define only one of these attributes in the `timing` group:
   - `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 ;
  }
}
```

10.17 Setting Conditional Timing Constraints

A conditional timing constraint describes a check that is performedwhen 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`

10.17.1 when and sdf_cond Simple Attributes

The `when` attribute defines enabling conditions for timing checks such as setup, hold, and recovery.

If you define `when`, you must define `sdf_cond`.

Using the `when` and `sdf_cond` pair is a short way of specifying `when_start`, `sdf_cond_start`, `when_end`, and `sdf_cond_end` when the start condition is identical to the end condition.

"Describing State-Dependent Delays" describes the `sdf_cond` and `when` attributes in defining state-dependent timing arcs.

10.17.2 when_start Simple Attribute

In a `timing` group, `when_start` defines a timing check condition specific to a start event. The expression in this attribute contains any pin, including input, output, I/O, and internal pins. You must use real pin names. Bus and bundle names are not allowed.

Syntax

when_start : "*Boolean expression*" ;

Boolean expression

A Boolean expression containing the names of input, output, inout, and internal pins.

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`.

10.17.3 sdf_cond_start Simple Attribute

In a `timing` group, `sdf_cond_start` defines a timing check condition specific to a start event in OVI SDF 2.1 syntax.

Syntax

```
sdf_cond_start : "SDF expression" ;
```

SDF expression

An SDF expression containing names of input, output, inout, and internal pins.

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`.

10.17.4 when_end Simple Attribute

In a `timing` group, `when_end` defines a timing check condition specific to an end event. The expression in this attribute contains any pin, including input, output, I/O, and internal pins. Pins must use real pin names. Bus and bundle names are not allowed.

Syntax

```
when_end : "Boolean expression" ;
```

Boolean expression

A Boolean expression containing names of input, output, inout, and internal pins.

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`.

10.17.5 sdf_cond_end Simple Attribute

In a `timing` group, `sdf_cond_end` defines a timing check condition specific to an end an in OVI SDF 2.1 syntax.

Syntax

```
sdf_cond_end : "SDF expression" ;
```

SDF expression

An SDF expression containing names of input, output, inout, and internal pins.

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`.

10.17.6 sdf_edges Simple Attribute

The `sdf_edges` attribute defines edge-specific information for both the start pins and the end pins. Edge types can be `noedge`, `start_edge`, `end_edge`, or `both_edges`. The default is `noedge`.

Syntax

```
sdf_edges : sdf_edge_type;
```

sdf_edge_type

One of these four edge types: `noedge`, `start_edge`, `end_edge`, or `both_edges`. The default is `noedge`.

Example

```
sdf_edges : both_edges;
```

10.17.7 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.

Syntax

```
pin() {
   ...
   min_pulse_width() {
      constraint_high : value ;
      constraint_low : value ;
      when : "Boolean expression" ;
      /* enabling condition              */
      sdf_cond : "Boolean expression" ;
      /* in SDF syntax */
   }
}
```

Example

```
pin(A) {
  . . .
  min_pulse_width() {
    constraint_high : 3.0 ;
    constraint_low : 3.5 ;
    when : "SE" ;
    sdf_cond : "SE == 1'B1" ;
  }
}
```

Example

```
min_pulse_width() {
 constraint_high : 3.0; /* min_pulse_width_high */
 constraint_low : 3.5; /* min_pulse_width_low */
 when : "SE";
```

```
 sdf_cond : "SE == 1'B1";
}
```

constraint_high and constraint_low Simple Attributes

At least one of these attributes must be defined in the `min_pulse_width` group. The `constraint_high` attribute defines the minimum length of time the pin must remain at logic 1. The `constraint_low` attribute defines the minimum length of time the pin must remain at logic 0.

when and sdf_cond Simple Attributes

These attributes define the enabling condition for the timing check. Both attributes are required in the `min_pulse_width` group.

10.17.8 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`.

Syntax

```
minimum_period() {
   constraint : value ;
   when : "Boolean expression" ;
   sdf_cond : "Boolean expression" ;
}
```

Example

```
minimum_period() {
 constraint : 9.5; /* min_period */
 when : "SE";
 sdf_cond : "SE == 1'B1";
}
```

constraint Simple Attribute

This required attribute defines the minimum clock period for the pin.

when and sdf_cond Simple Attributes

These attributes define the enabling condition for the timing check. Both attributes are required in the `minimum_period` group.

10.17.9 Using Conditional Attributes With No-Change Constraints

As shown in Table 10-6, conditional timing check attributes have different interpretations when you use them with no-change timing constraints. See “Setting No-Change Timing Constraints” for a description of no-change timing constraint values.

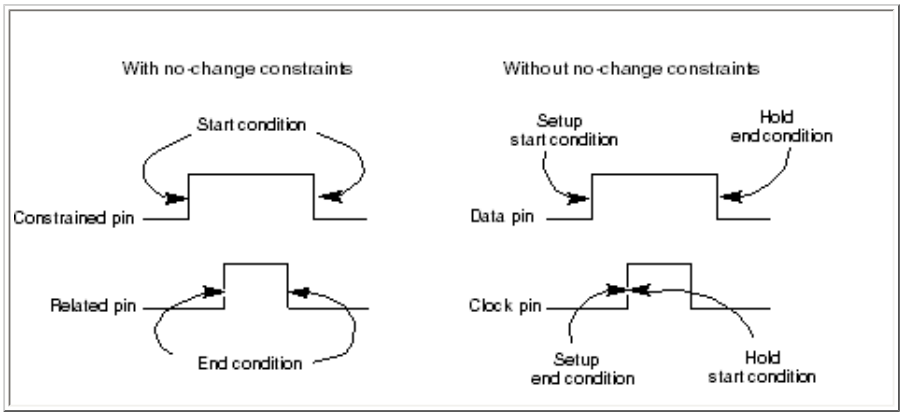
Table 10-6 Conditional Timing Attributes With No-Change Constraints

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 10-19 shows the different interpretation of setup and hold conditions when used with a no-change timing constraint.

Figure 10-19 Interpretation of Conditional Timing Constraints With No-Change Constraints



10.18 Timing Arc Restrictions

The following section describes timing arc limitations.

10.18.1 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).

10.19 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.

10.19.1 CMOS Generic Delay Model

In the CMOS D flip-flop description in Example 10-11, pin D contains setup and hold constraints; pins Q and QN contain preset, clear, and edge-sensitive arcs.

Example 10-11 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 ;
      }
    }
  }
}
```

10.19.2 CMOS Piecewise Linear Delay Model

In the CMOS piecewise linear D flip-flop description in Example 10-12, pin D contains setup and hold constraints; pins Q and QN contain preset, clear, and edge-sensitive arcs.

Example 10-12 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 */
    }
  }
}
```

10.19.3 CMOS Nonlinear Delay Model

In the nonlinear library description in Example 10-13, pin D contains setup and hold constraints; pins Q and QN contain preset, clear, and edge-sensitive arcs.

Example 10-13 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");
        }
      }
    }
  }
}
```

10.19.4 CMOS Scalable Polynomial Delay Model

In the scalable polynomial delay model in Example 10-14, pin D contains setup and hold constraints; pins Q and QN contain preset, clear, and edge-sensitive arcs.

Example 10-14 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 */
```

10.19.5 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) {
```

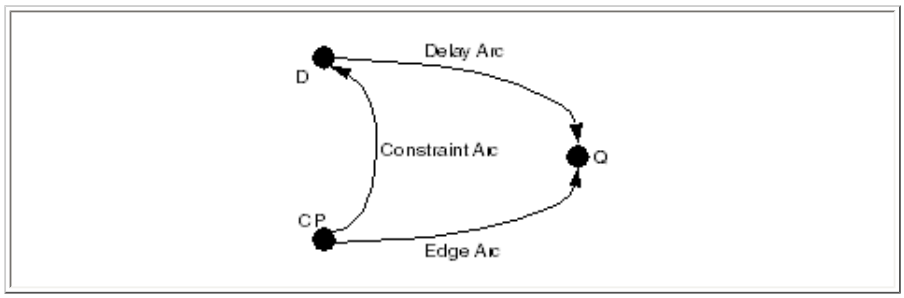
```
      ...
      pin(clk) {
        direction:input;
        timing() {
          timing_type :max_clock_tree_path;
          timing_sense :positive_unate;
          cell_rise(lu_template) {
            values ("0.1, 0.15, 0.20, 0.29");
          }
          cell_fall(lu_template) {
            values ("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");
          }
        }
      }
    ...
  }
```

10.20 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 10-20 shows a transparent latch timing model.

Figure 10-20 Transparent Latch Timing Model



Syntax

library (*name*$_{string}$) {

```
cell (name_string) {
  ...
  timing_model_type : value_enum ;
  ....
  pin (data_pin_name_string) {
    tlatch (enable_pin_name_string){
      edge_type : value_enum ;
      tdisable : 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, the latch clock pin is inferred 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, transparency is inferred 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, all data pin output pin arcs that reference a tlatch group whose tdisable attribute is set to true on an edge triggered flip flop are disabled and ignored..

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 ...
  }
}
```

10.21 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.

10.21.1 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.

10.21.2 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.

10.21.3 Cell-level Attributes

This section describes driver waveform attributes defined at the cell level.

driver_waveform Attribute

The `driver_waveform` attribute is an optional string attribute that allows you to define a cell-specific driver waveform. The value must be the `driver_waveform_name` predefined in the `normalized_driver_waveform` table.

When the attribute is defined, the cell uses the specified driver waveform during characterization. When it is not specified, the common driver waveform (the `normalized_driver_waveform` table without the `driver_waveform_name` attribute) is used for the cell.

driver_waveform_rise and driver_waveform_fall Attributes

The `driver_waveform_rise` and `driver_waveform_fall` string attributes are similar to the `driver_waveform` attribute. These two attributes allow you to define rise-specific and fall-specific driver waveforms. The `driver_waveform` attribute can coexist with the `driver_waveform_rise` and `driver_waveform_fall` attributes, though the `driver_waveform` attribute becomes redundant.

You should specify a driver waveform for a cell by using the following priority:

1. Use the `driver_waveform_rise` for a rise arc and the `driver_waveform_fall` for a fall arc during characterization. If they are not defined, specify the second and third priority driver waveforms.

2. Use the cell-specific driver waveform (defined by the `driver_waveform` attribute).
3. Use the library-level default driver waveform (defined by the `normalized_driver_waveform` table without the `driver_waveform_name` attribute).

The `driver_waveform_rise` attribute can refer to a `normalized_driver_waveform` that is either rising or falling. You can invert the waveform automatically during runtime if necessary.

10.21.4 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".

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".

10.21.5 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 */
```

10.22 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 11, “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.

10.22.1 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 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" );
```

10.22.2 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.

10.22.3 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).

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).

10.22.4 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*/
```

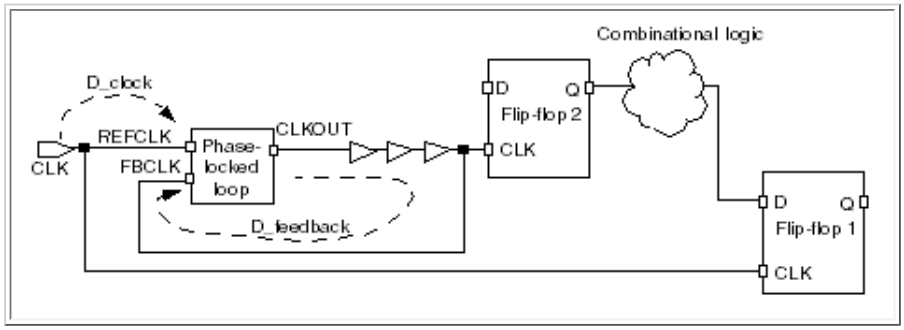
10.23 Phase-Locked Loop Support

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 10-21 shows a circuit with a phase-locked loop. Without the phase-locked loop block, there is a large clock skew in the clocks arriving at the launch point and at the flip-flops. This large skew results in an extremely tight delay constraint for the combinational logic. A phase-locked loop reduces the large skew between the launch point and the flip-flops.

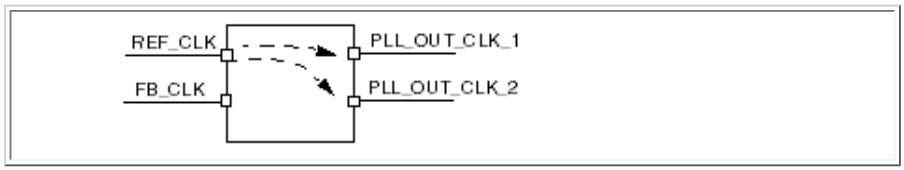
Figure 10-21 Phase-Locked Loop Circuit



The phase-locked loop generates a phase-shifted version of the reference clock at the output pin so that the phase of the feedback clock matches the phase of the reference clock. If the clock arrives at the reference pin of the phase-locked loop at the time, D_clock, and the delay of the feedback path is D_feedback, the source latency of the clock generated at the output of the phase-locked loop is D_clock - D_feedback. The clock arrives at the feedback pin at time, D_clock, which is the same as the time the clock arrives at the reference pin. Therefore, the phase-locked loop eliminates the latency on the launch path and relaxes the delay constraint on the combinational logic.

Figure 10-22 shows a simple phase-locked loop model. The phase-locked loop information that a library should contain are pins and timing arcs inside the phase-locked loop. The forward arcs from REF_CLK to PLL_OUT_CLK_* in the figure simulate the phase shift behavior of the phase-locked loop. There are two half-unate arcs from the reference clock to each output of the phase-locked loop.

Figure 10-22 Simple Phase-Locked Loop Model



10.23.1 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;
    …
  }
}
```

10.23.2 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.

10.23.3 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.

10.23.4 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 */
```

11. Composite Current Source Modeling

This chapter provides an overview of composite current source modeling (CCS). It covers the new syntax for CCS modeling in the following sections:

11.1 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.

11.2 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.

11.2.1 Composite Current Source Lookup Table Model

You can represent composite current source driver models in your libraries by using lookup tables. To define your lookup tables, use the following groups and attributes:

- `output_current_template` group in the `library` group level
- `output_current_rise` and `output_current_fall` groups in the `timing` group level

11.2.2 Defining the output_current_template Group

Use this library-level group to create templates of common information that multiple current vectors can use. A table template specifies the composite current source driver model and the breakpoints for the axis. Specifying `index_1`, `index_2`, and `index_3` values at the library level is optional.

Syntax

```
library(name_id) {
...
   output_current_template(template_name_id) {
      variable_1: input_net_transition;
      variable_2: total_output_net_capacitance;
      variable_3: time;
      ...
   }
...   }
```

Template Variables

The table template specifying composite current source driver models can have three variables: `variable_1`, `variable_2`, and `variable_3`. The valid values for `variable_1` and `variable_2` are `input_net_transition` and `total_output_net_capacitance`. The only valid value for `variable_3` is `time`.

Example

```
library (new_lib) {
...
  output_current_template (CCT) {
    variable_1: input_net_transition;
    variable_2: total_output_net_capacitance;
    variable_3: time;
    ...
  }
  ...
}
```

Defining the Lookup Table Output Current Groups

To specify the output current for the nonlinear table model, use the `output_current_rise` and `output_current_fall` groups within the `timing` group.

Syntax

```
timing() {
   output_current_rise () {
      vector (template_name_id){
         reference_time : float;
         index_1 (float);
         index_2 (float);
         index_3 ("float,..., float");
         values("float,..., float");
      }
   }
}
```

vector Group

Define the `vector` group in the `output_current_rise` or `output_current_fall` group. This group stores current information for a particular input slew and output load.

reference_time Simple Attribute

Define the `reference_time` simple attribute in the `vector` group. The `reference_time` attribute represents the time at which the input waveform crosses the rising or falling input delay threshold.

Template Variables

The table template specifying composite current source driver models can have three variables: `variable_1`, `variable_2`, and `variable_3`. The valid values for `variable_1` and `variable_2` are `input_net_transition` and `total_output_net_capacitance`. The only valid value for `variable_3` is `time`.

The index value for `input_net_transition` or `total_output_net_capacitance` is a single floating-point number. The index values for `time` are a list of floating-point numbers.

The `values` attribute defines a list of floating-point numbers that represent the current values of the driver model.

Example

```
library (new_lib) {
...
  output_current_template (CCT) {
```

```
    variable_1: input_net_transition;
    variable_2: total_output_net_capacitance;
    variable_3: time;
  }
  ...
  timing() {
    output_current_rise() {
      vector(CCT) {
        reference_time : 0.05;
        index_1(0.1);
        index_2(2.1);
        index_3("1.0, 1.5, 2.0, 2.5, 3.0");
        values("1.1, 1.2, 1.4, 1.3, 1.5");
      ...
      }
    }
    output_current_fall() {
      vector(CCT) {
        reference_time : 0.05;
        index_1(0.1);
        index_2(2.1);
        index_3("1.0, 1.5, 2.0, 2.5, 3.0");
        values("1.1, 1.2, 1.4, 1.3, 1.5");
      ...
      }
    }
  }
}
```

11.3 Mode and Conditional Timing Support for Pin-Level CCS Receiver Models

Liberty 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.

Liberty 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.

11.3.1 Conditional Timing Support Syntax

Liberty 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.

Example

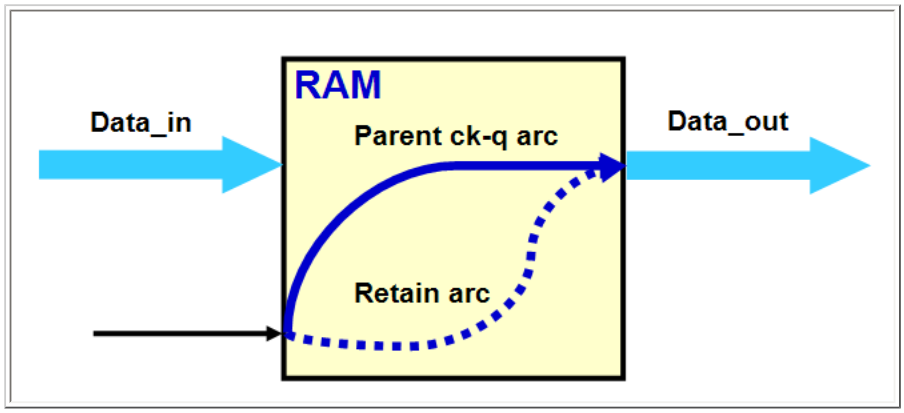
```
library(new_lib) {
 ...
 output_current_template(CCT) {
  variable_1: input_net_transition;
  variable_2: total_output_net_capacitance;
  variable_3: time;
  index_1("0.1, 0.2");
  index_2("1, 2");
  index_3("1, 2, 3, 4, 5");
 }
 lu_table_template(LTT1) {
  variable_1: input_net_transition;
  index_1("0.1, 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 */
```

11.4 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 11-1, ensures that the output does not change during this time interval.

Figure 11-1 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`

11.4.1 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_ouput_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.

11.4.2 Compact CCS Retain Arc Syntax

The compact CCS retain arc format is the same as a general compact CCS timing arc. Liberty 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.

11.5 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.

11.5.1 Composite Current Source Lookup Table Model

Library information for composite current source receiver modeling can be defined as follows:

- At the pin level inside the `receiver_capacitance` group
- At the timing level by using the following groups: `receiver_capacitance1_rise` `receiver_capacitance1_fall` `receiver_capacitance2_rise` `receiver_capacitance2_fall`

Values for rise and fall can be defined at the pin or timing level. The pin-level definition does not depend on output capacitance and is useful when there are no forward timing arcs.

11.5.2 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.

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.

Syntax

library(*name*$_{id}$) {
…
  lu_table_template(*template_name*$_{id}$) {
    variable_1: input_net_transition;
    index_1 ("*float,..., float*");
    ...
  }
...}

Template Variables

In the `pin` group, the table template specifying composite current source receiver models can have one variable: `variable_1`. The only valid value is `input_net_transition`.

The index values in the `index_1` attribute are a list of ascending floating-point numbers.

Example

```
...
  lu_table_template(LTT1) {
    variable_1: input_net_transition;
    index_1("0.1, 0.2, 0.3, 0.4");
  }
```

Defining the Lookup Table `receiver_capacitance` Group

To specify the receiver capacitance for the nonlinear table model, use the `receiver_capacitance` group within the `pin` group.

Syntax for Pin Level

pin(*name*$_{id}$) {
  direction: input; /* or "inout" */
  receiver_capacitance() {
    receiver_capacitance1_rise(*template_name*$_{id}$) {
      index_1("*float,..., float*"); /* optional */
      values("*float,..., float*");
    }

```
    receiver_capacitance1_fall(template_name_id) {
      index_1("float,..., float"); /* optional */
      values("float,..., float");
    }     receiver_capacitance2_rise(template_name_id)
      index_1("float,..., float"); /* optional */
      values("float,..., float");
    }     receiver_capacitance2_fall(template_name_id) {
      index_1("float,..., float"); /* optional */
      values("float,..., float");
    }
  }
}
```

Template Variables

In the `pin` group, the table template specifying receiver characteristics can have one variable: `variable_1`. The only valid value is `input_net_transition`.

The index values in the `index_1` attribute are a list of ascending floating-point numbers.

The `values` attribute defines a list of floating-point numbers that represent the capacitance of the receiver model.

Example for Pin Level

```
pin(A) { /* pin-based receiver model*/

  direction : input;
  receiver_capacitance() {
    receiver_capacitance1_rise(LTT1) {
      values("1.0, 4.1, 2.1, 3.0");
    }
    receiver_capacitance1_fall(LTT1) {
      values("1.0, 3.2, 2.1, 4.0");
    }
    receiver_capacitance2_rise(LTT1) {
      values("1.0, 4.1, 2.1, 3.0");
    }
    receiver_capacitance2_fall(LTT1) {
      values("1.0, 3.2, 2.1, 4.0");
    }
  }
}
```

11.5.3 Defining the Receiver Capacitance Groups at the Timing Level

At the timing level, you do not need to define the `receiver_capacitance` group. Define the receiver capacitance for the timing arcs by using only the `receiver_capacitance1_rise`, `receiver_capacitance1_fall`, `receiver_capacitance2_rise, and receiver_capacitance2_fall` groups.

Defining the lu_table_template Group

Use this library-level group to create templates of common information that multiple lookup tables can use. A table template specifies the table parameters and the breakpoints for each axis. Assign each template a name so that lookup tables can refer to it.

lu_table_template Group

Define your lookup table templates in the `library` group.

Syntax

```
library(name_id) {
```

```
...
  lu_table_template(template_name_id) {
    variable_1: input_net_transition;
    variable_2: total_output_net_capacitance;
    index_1 ("float,..., float");
    index_2 ("float,..., float");
    ...
  }
...  }
```

Template Variables

The table template specifying composite current source receiver models can have only two variables: `variable_1` and `variable_2`. The parameters are the input transition time and the total output capacitance of a constrained pin.

The index values in the `index_1 and index_2` attributes are a list of ascending floating-point numbers.

Example

```
...
  lu_table_template(LTT2) {
    variable_1: input_net_transition;
    variable_2: total_output_net_capacitance;
    index_1("0.1, 0.2, 0.4, 0.3");
    index_2("1.0, 2.0");
  }
```

Defining the Lookup Table receiver_capacitance Groups

To specify the receiver capacitance for the nonlinear table model, use the `receiver_capacitance1_rise, receiver_capacitance1_fall, receiver_capacitance2_rise receiver_capacitance2_fall` groups within the `timing` group.

Syntax for Timing Level

```
direction: output; /* or "inout"
*/
timing () {
  ...
  receiver_capacitance1_rise(template_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

In the timing level, the table template specifying composite current source receiver models can have two variables: `variable_1` and `variable_2`. The valid values for either variable are `input_net_transition` and `total_output_net_capacitance.`

The index values in the `index_1 and index_2` attributes are a list of ascending floating-point numbers.

The `values` attribute defines a list of floating-point numbers that represent the capacitance for the receiver model.

Example at the timing level

```
timing() { /* timing arc-based receiver model*/
  ...
  related_pin : "B"
  receiver_capacitance1_rise(LTT2) {
    values("1.1, 4., 2.0, 3.2");
  }
  receiver_capacitance1_fall(LTT2) {
    values("1.0, 3.2, 4.0, 2.1");
  }
  receiver_capacitance2_rise(LTT2) {
    values("1.1, 4., 2.0, 3.2");
  }
  receiver_capacitance2_fall(LTT2) {
    values("1.0, 3.2, 4.0, 2.1");
  }
  ...
}
```

11.6 Composite Current Source Driver and Receiver Model Example

Example 11-1 is an example of composite current source driver and receiver model syntax.

Example 11-1 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 */
```

12. 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

12.1 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.

12.2 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:

- Describing CCS timing base curves.
- Describing the syntax of base curves and the compact CCS driver modeling format.

12.2.1 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 12-1 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 12-1 Inverter Cell Rise Transition With Existing CCS Format

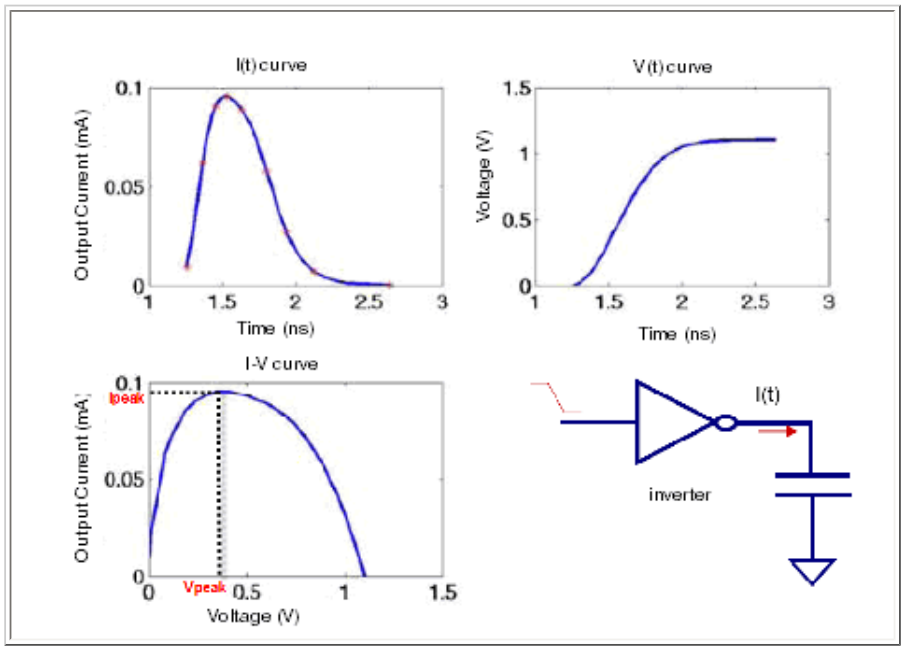


Figure 12-2 shows the mapping of I-V curves to base curves. The I-V curve is split into two halves at the peak, and an eleven point normalized base curve in the base curve database is selected to exactly match each half curve.

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.

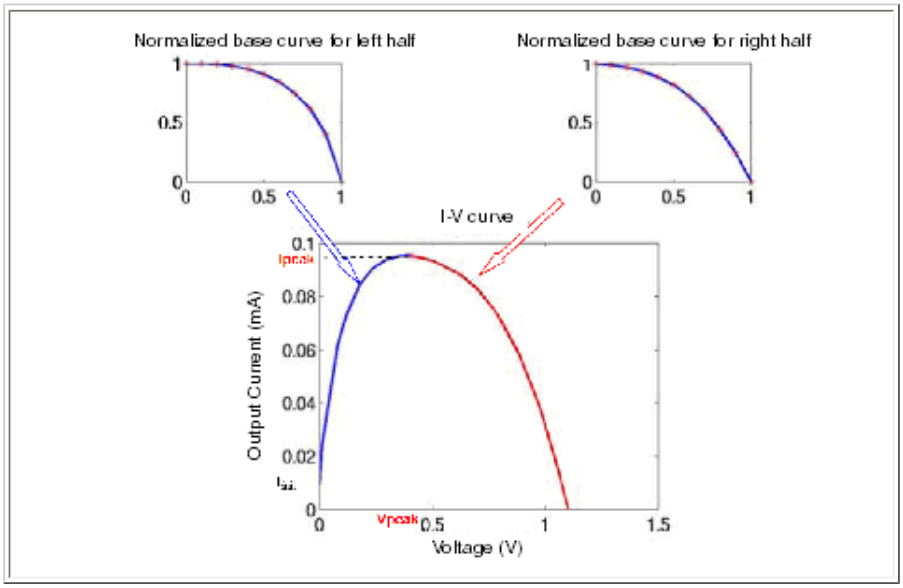
Left id

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 12-2 Using Base Curves to Simplify I-V Curve Modeling



12.2.2 Compact CCS Timing Model Syntax

In Figure 12-2, 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 12-2 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 12-2) 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_curve` group. When the `base_curve_type` is a `ccs_timing_half_curve`, at least six string parameters should be defined. They are `init_current`, `peak_current`, `peak_voltage`, `peak_time`, `left_id`, and `right_id`. If any of these six parameters are missing, a compilation error is issued.

compact_ccs_{rise|fall} Group

This is the compact CCS timing data in timing arc group, which has the following attributes:

base_curves_group Attribute

Defining this attribute is optional when the `base_curves_name` is same as that defined in the `compact_lut_template` group. This group is referenced by the `compact_ccs_{rise|fall}` group.

values Attribute

Values of compact CCS timing data depend on how you define `index_3` values.

For compact CCS timing, base curves data `left_id` and `right_id` values can only be integers.

12.2.3 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*/
```

12.3 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” .

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.

12.3.1 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 12-1.

The parameters defined in `default_operating_conditions` are process, temperature, and voltage. The voltage names are defined by using the `voltage_map` complex attribute. For more information see "voltage_map Complex Attribute".

You can use the following predefined parameters:

- For the parameters defined in `operating_conditions`, if `voltage_map` is defined, and you specify these attributes as values of `va_parameters`.
- 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`.

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.

- 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 12-2.
- All variation-aware compact CCS driver model groups inside the `timing_based_variation` share the same `va_parameters` attribute.

va_values Attribute

The `va_values` attribute specifies values of each variation parameter for all corners characterized in the variation-aware compact CCS driver model groups.

- Required for the variation-aware compact CCS driver model groups.
- The value of this attribute has a 1-to-1 mapping to the corresponding `va_parameters`.

For an example that shows how to specify `va_values` with three variation parameters, see Example 12-3. In the example, the first variation parameter has a nominal value of 0.50, the second parameter has a nominal value of 1.0, and the third parameter has a nominal value of 2.0. All parameters have a variation range of -10% to +10%.

values Attribute

The `values` attribute follows the same rules as the nominal compact CCS driver model groups with the following exceptions:

- The `left_id` and `right_id` are optional.
- The `left_id` and `right_id` values must be used together. They can either be omitted or defined together in the `compact_lut_template`.
- If `left_id` and `right_id` are not defined in the variation-aware compact CCS driver model group, they default to the values defined in the nominal compact CCS driver model group.

timing_based_variation and pin_based_variation Groups

These groups represent variation-aware receiver capacitance information under the timing and pin groups.

- If `receiver_capacitance` group exists in pin group, variation-aware CCS receiver model groups are required in `pin_based_variation`.
- If nominal CCS receiver model groups exist in the timing group, variation-aware CCS receiver model groups are required in `timing_based_variation`.

va_parameters Attribute

The `va_parameters` attribute specifies a list of variation parameters within a `timing_based_variation` or `pin_based_variation` group. See "va_parameters Attribute" 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 12-5.
- In timing-based models, if the nominal compact CCS driver model group and variation-aware CCS receiver model groups are defined under the same timing group, the nominal values are applied to the nominal compact CCS driver model groups.

va_receiver_capacitance1_rise, va_receiver_capacitance1_fall, va_receiver_capacitance2_rise, va_receiver_capacitance2_fall Groups

These groups specify characterization corners with variation values in timing_based_variation and pin_based_variation groups.

- 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.

12.3.2 Variation-Aware CCS Timing Receiver Model

The variation-aware CCS receiver model is expected to be used together with variation-aware compact CCS driver model. The timing_based_variation and pin_based_variation groups specify timing and pin groups respectively. In both groups, the variation-aware CCS receiver model groups are used to represent variation-aware CCS receiver information as defined below.

Syntax

```
library() {
  …
  lu_table_template(<timing_based_template_name>) {
   variable_1 : input_net_transition;
   variable_2 : total_output_net_capacitance; …}
  lu_table_template(<pin_based_template_name>) {
   variable_1 : input_net_transition; …}
  va_parameters(<string> , …);
  …
   pin(<pin_name>) {
    receiver_capacitance() {
    receiver_capacitance1_rise(<template_name>) { …}
    receiver_capacitance2_rise(<template_name>) { …}
    receiver_capacitance1_fall(<template_name>) { …}
    receiver_capacitance2_fall(<template_name>) { … }
   }
   pin_based_variation() {
    va_parameters(<string> , … );
    nominal_va_values(<float>,…);
    va_receiver_capacitance1_rise(<pin_based_template_name>) {
     va_values(<float>, …);
```

```
        values("<float>, …", …);
       …}
       va_receiver_capacitance2_rise(<pin_based_template_name>) {…}
       va_receiver_capacitance1_fall(<pin_based_template_name>) {...}
       va_receiver_capacitance2_fall(<pin_based_template_name>) { …}
      …} /* end of pin_based_variation */
    …} /* end of pin */
…
    pin(<pin_name>) {
    …
      timing() {
      receiver_capacitance1_rise(<template_name>) {…}
      receiver_capacitance2_rise(<template_name>) {…}
      receiver_capacitance1_fall(<template_name>) {…}
      receiver_capacitance2_fall(<template_name>) {…}
      timing_based_variation() {
       va_parameters(<string> , … );
       nominal_va_values(<float>,…);
       va_receiver_capacitance1_rise(<timing_based_template_name>) {
        va_values(<float>, …);
        values("<float>, …", …);
       …}
       va_receiver_capacitance2_rise(<timing_based_template_name>) {…}
       va_receiver_capacitance1_fall(<timing_based_template_name>) {…}
       va_receiver_capacitance2_fall(<timing_based_template_name>) {…}
     …} /* end of timing_based_variation */
    …} /*end of timing */
  …} /* end of pin */
…
```

timing_based_variation and pin_based_variation Groups

These groups represent variation-aware receiver capacitance information under the pin or timing group level.

- If the `receiver_capacitance` group exists in the pin group, variation-aware CCS receiver model groups are required in `pin_based_variation`.
- If nominal CCS receiver model groups exist in the timing group, variation-aware CCS receiver model groups are required in `timing_based_variation`.

va_parameters Complex Attribute

This attribute specifies a list of variation parameters within `timing_based_variation` or `pin_based_variation`. See "va_parameters Attribute" for more 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 12-5.
- In a timing-based model, if nominal CCS receiver model and variation-aware CCS receiver model groups are defined under the same timing group, the nominal values are applied to nominal CCS receiver model groups.

va_receiver_capacitance1_rise, va_receiver_capacitance1_fall, va_receiver_capacitance2_rise, and va_receiver_capacitance2_fall Groups

These groups specify characterization corners with variation values in the `timing_based_variation` and `pin_based_variation` groups.

- 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`.

12.3.3 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, as defined in the syntax below.

Syntax

```
library() {
 …
 lu_table_template(<template_name>) {
  variable_1 : <variables>;
  variable_2 : <variables>;
  variable_3 : <variables>; … }
 va_parameters(<string> , …);
 …
  timing () {
  …
  interdependence_id :<integer>;
  rise_constraint(<template_name>) { … }
  fall_constraint(<template_name>) { … }
  timing_based_variation() {
   va_parameters(<string> , … );
   nominal_va_values(<float>,…);
   va_rise_constraint(<template_name>) {
    va_values(<float>,…);
    values("<float>,…");
   …}
   va_fall_constraint(<template_name>) { …}
  … } /* end of timing_based_variation */
 … } /* end of timing */
 …} /* end of pin */
…} /* end of library */
```

timing_based_variation Group

The `timing_based_variation` group specifies the rise and fall timing constraints for variation parameters within a timing group. The rise and fall timing constraints are specified in the `va_rise_constraint` and `va_fall_constraint` groups respectively.

- The `va_rise_constraint` group is required only if `rise_constraint` group exists within a timing group.
- The `va_fall_constraint` group is required only if `fall_constraint` group exists within a timing group.

va_parameters Complex Attribute

This complex attribute specifies a list of variation parameters within `timing_based_variation` or `pin_based_variation`. See "va_parameters Attribute" 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" for more information.

va_rise_constraint and va_fall_constraint Groups

The `va_rise_constraint` and `va_fall_constraint` groups specify characterization corners with variation values in `timing_based_variation`.

- The template name refers to the `lu_table_template` group.
- Both groups can be specified under different `timing_based_variation` groups if they cannot share the same `va_parameters`.
- 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`.

12.3.4 Conditional Data Modeling for Variation-Aware Timing Receiver Models

Liberty provides the following syntax to support conditional data modeling for pin-based variation-aware timing receiver models.

Syntax

```
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.

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");
   }
  }
  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 */
```

12.3.5 Variation-Aware Compact CCS Retain Arcs

Variation-aware timing models include:

- Timing-based modeling for compact CCS timing drivers.
- Timing-based and pin-based modeling for CCS timing receivers.
- Timing-based modeling for regular or interdependent timing constraints.

Liberty provides the following syntax in the `timing_based_variation` group to support retain arcs for compact CCS driver models.

Syntax

```
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.

12.3.6 Variation-Aware Syntax Examples

Example 12-1 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”.

Example 12-2 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".

Example 12-3 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".

Example 12-4 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”.

Example 12-5 pin_based_variation Group

```
…
pin(pin_name) {
 receiver_capacitance() {
  receiver_capacitance1_rise(template_name) {
  /* nominal input capacitance table */ …}
 …}
 pin_based_variation() {
  nominal_va_values(2.0, 4.54, 0.23);
  /* These nominal values apply to nominal input capacitance tables. */
    va_receiver_capacitance1_rise(template_name) {
    /* variational input capacitance table */
 …}… }… } /* end of pin */
…
```

For information on peak_voltage in values attribute see “timing_based_variation and pin_based_variation Groups”.

Example 12-6 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” .

Example 12-7 nominal_va_values in Advanced CCS Modeling Usage

```
/* ASSUME that there is no voltage_map defined in library */

library (sample) {
 operating_conditions (typical) {
  process : 1.5 ;
  temperature : 70 ;
  voltage : 2.75 ; …}
 default_operating_conditions: typical;
 …
  timing_based_variation() {
   va_parameters(voltage, temperature, process);
   nominal_va_values(2.00, 70, 1.5);
   /* error : The nominal voltage defined in
    default_operating_conditions is 2.75. values 2.00 is wrong. */


/* There is voltage_map defined at library level. */
library (sample) {
 operating_conditions (typical) {
  process : 1.5 ;
  temperature : 70 ;
  voltage : 2.75 ; …}
 voltage_map(VDD1, 2.75);
 default_operating_conditions: typical;
 …
  timing_based_variation() {
   va_parameters(voltage);
   nominal_va_values(2.00);
   /* This is okay because "voltage" is an user-defined parameter. */
 …
```

Example 12-8 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 12-9 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 */
```

13. Composite Current Source Signal Integrity Modeling

This chapter provides an overview of composite current source modeling (CCS) to support noise (signal integrity) modeling for advanced technologies. The following information is covered:

- Composite Current Source Signal Integrity Noise Model
- CCS Noise Modeling for Unbuffered Cells With a Pass Gate

13.1 Composite Current Source Signal Integrity Noise Model

CCS noise modeling can capture essential noise properties of digital circuits using a compact library representation. It enables fast and accurate gate-level noise analysis while maintaining a relatively simple library characterization. CCS noise modeling supports noise combination and driver weakening.

CCS noise is characterization data that provides information for noise failure detection on cell inputs, calculation of noise bumps on cell outputs, and noise propagation through the cell. For the best accuracy, you must add CCS timing data to the library in addition to the CCS noise data. The CCS noise data includes the following:

- Channel-connected block parameters
- DC current tables
- Timing tables for rising and falling transitions
- Timing tables for low and high propagated noise

13.1.1 Syntax

The CCS noise syntax is as follows:

```
library (name) {
...
 cell (name) {
  pin (name) {
    ...
   ccsn_first_stage () {
    is_needed : <boolean>;
      is_inverting : <boolean>;
      stage_type : <stage_type_value>;
      miller_cap_rise : <float>;
      miller_cap_fall : <float>;
      dc_current (<dc_current_template>)
        index_1("<float>, ...");
        index_2("<float>, ...");
```

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

      }

      output_voltage_rise ( )
        vector (<output_voltage_template_name>) {
        index_1(<float>);
        index_2(<float>);
        index_3("<float>, ...");
        values("<float>, ...");
      }
      ...
    }
      output_voltage_fall ( ) {
        vector (<output_voltage_template_name>) {
        index_1(<float>);
        index_2(<float>);
        index_3("<float>, ...");
        values("<float>, ...");
      }
      ...

    }
      propagated_noise_low ( ) {
        vector (<propagated_noise_template_name>) {
        index_1(<float>);
        index_2(<float>);
        index_3(<float>);
         index_4("<float>, ...");
        values("<float>, ...");
      }
    ...
  }
      propagated_noise_high ( ) {
        vector (<propagated_noise_template_name>) {
        index_1(<float>);
        index_2(<float>);
        index_3(<float>);
        index_4("<float>, ...");
        values("<float>, ...");
      }
      ...
    }
      when : <boolean expression>;
  }
 }
}
}
```

13.1.2 CCS Noise Library Example

The following is a sample CCS noise library.

Example 13-1 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 *
```

13.1.3 Conditional Data Modeling in CCS Noise Models

Liberty 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.

Liberty provides mode support for pin-based CCS noise data modeling, as shown in the following syntax.

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 */
```

13.2 CCS Noise Modeling for Unbuffered Cells With a Pass Gate

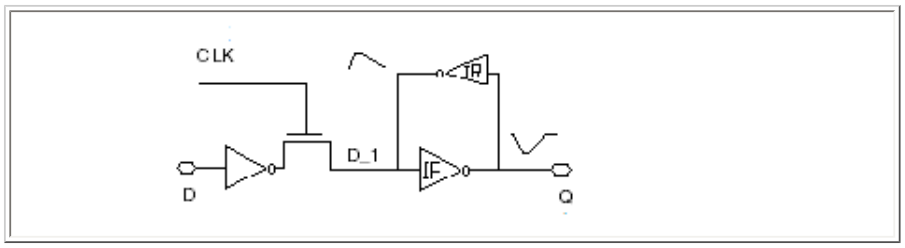
Unbuffered input and output latches are a special type of cell that has an internal memory node connected to an input or output pin. In order to increase the speed of the design and lower power consumption, these cells do not use inverters.

Figure 13-1 and Figure 13-2 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 13-1, Q is connected to internal node D_1 through the IR inverter.

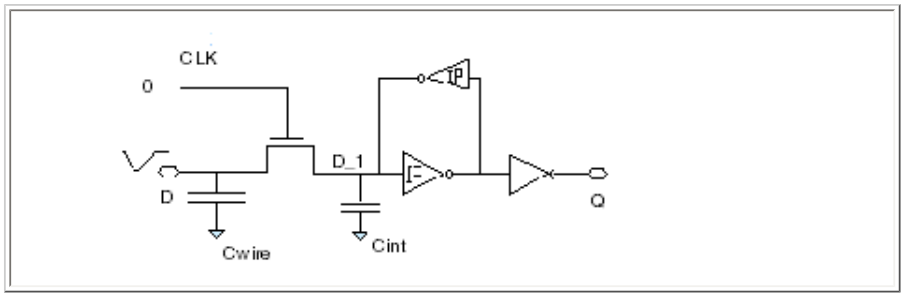
  Figure 13-1 Unbuffered Output Latch



- 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 13-2, D is connected to internal node D_1 through a pass gate.

  Figure 13-2 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.

13.2.1 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;
     …
    }
    …
  }

}
```

13.2.2 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.

14. Composite Current Source Power Modeling

This chapter provides an overview of composite current source (CCS) modeling to support advanced technologies. It covers the syntax for CCS power modeling in the following sections:

- Composite Current Source Power Modeling
- Compact CCS Power Modeling
- Composite Current Source Dynamic Power Examples

14.1 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.

14.1.1 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”. The leakage current syntax is as follows:

Example 14-1 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 14-2 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 */
  ...
 }
}
```

14.1.2 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" .

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.

14.1.3 Intrinsic Parasitic

You can use the syntax in Example 14-3 for intrinsic parasitic models. The syntax consists of two parts: one is intrinsic resistance and the other is intrinsic capacitance.

Example 14-3 Intrinsic Parasitic Model

```
cell (<cell_name> ) {
 ...
 intrinsic_parasitic() {
  when : <boolean expression>;
  intrinsic_resistance(<pg_pin_name>) {
   related_output : <output_pin_name>;
   value : < float>;
  }
  intrinsic_capacitance(<pg_pin_name>) {
   value : <float>;
  }
 }

 intrinsic_parasitic() {
without when statement */
  /* default state */

 }
}
```

14.1.4 Parasitics Modeling in Macro Cells

For macro cells, the `total_capacitance` group is provided within the `intrinsic_parasitic` group.

total_capacitance Group

The `total_capacitance` group specifies the macro cell's total capacitance on a power or ground net within the `intrinsic_parasitic` group.

- This group can be placed in any order if there is more than one `total_capacitance` group within an `intrinsic_parasitic` group.
- If the `total_capacitance` group is not defined for a certain power and ground pin, the value of capacitance defaults to 0.0. The default value is provided by the tool.
- The parasitics modeling of total capacitance in macros cells is not state dependent. This means that there is no state condition specified in `intrinsic_parasitic`.

Syntax

```
cell (<cell_name> ) {
 power_cell_type : <enum(stdcell, macro)>;
 ...
 intrinsic_parasitic() {
  total_capacitance(<a pg pin name>) {
   value : <float>;
  }
  ...
 }
 ...
}
```

14.1.5 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.

Syntax

The following syntax for dynamic power format provides more accurate models for macro cells:

```
…
cell(<cell_name>) {
 …
 power_cell_type : <enum(stdcell, macro)>
 dynamic_current() {
   when : <boolean expression>
   related_inputs : <input_pin_name>;
   switching_group() {
     min_input_switching_count : <integer>;
     max_input_switching_count :<integer>;
     pg_current(<pg_pin_name>) {
       vector(<template_name>) {
         reference_time : <float>;
         index_1(<float>);
         index_2("<float>,…");
         values("<float>, …");
       }  /* end vector group*/
       …
     }  /end pg_current group */
     …
   } /* end switching_group */
   …
 } /* end dynamic_current group*/
 …
} /* end cell group*/
…
```

The `min_input_switching_count` and `max_input_switching_count` attributes specify the number of bits in the input bus that are switching simultaneously while an asynchronous event occurs.

A single switching bit can be defined by setting the same value in both attributes. The following example shows that any three bits specified in `related_inputs` are switching simultaneously.

```
...
min_input_switching_count : 3;
max_input_switching_count : 3;
...
```

A range of switching bits can be defined by setting the minimum and maximum value. The following example shows

that any 2, 3, 4 or 5 bits specified in `related_inputs` are switching simultaneously.

```
...
min_input_switching_count : 2;
max_input_switching_count : 5;
...
```

min_input_switching_count Attribute

This attribute specifies the minimum number of bits in an input bus that are switching simultaneously.

- The count must be integer.
- The count must greater than 0 and less than `max_input_switching_count`.

max_input_switching_count Attribute

This attribute specifies the maximum number of bits in an input bus that are switching simultaneously.

- The count must be integer.
- The count must greater than `min_input_switching_count`.
- The count must be less than the total number of bits listed in `related_inputs`.

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 */
   ...
```

14.1.6 Dynamic Current Syntax

The following syntax is used for instantaneous power data, which is captured at the cell level.

Example 14-4 Dynamic Current Syntax

```
cell(<cell_name>) {

 power_cell_type : <enum(stdcell, macro)>
 dynamic_current() {
   when : <boolean expression>
   related_inputs : <input_pin_name>;
   related_outputs : <output_pin_name>;
   typical_capacitances(<float>, );/* applied for cross type;/*
   switching_group() {
     input_switching_condition(<enum(rise, fall)>);
     output_switching_condition(<enum(rise, fall)>);
     pg_current(<pg_pin_name>) {
       vector(<template_name>) {
         reference_time : <float>;
          index_output : <output_pin_name>; /* applied for cross type;/*
         index_1(<float>);

         index_n(<float>);
         index_n+1(<float>, );
         values(<float>, );
       }  /* vector */

     }  /pg_current */

    } /* switching_group */

 } /* dynamic_current */

} /* cell */
```

14.2 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.

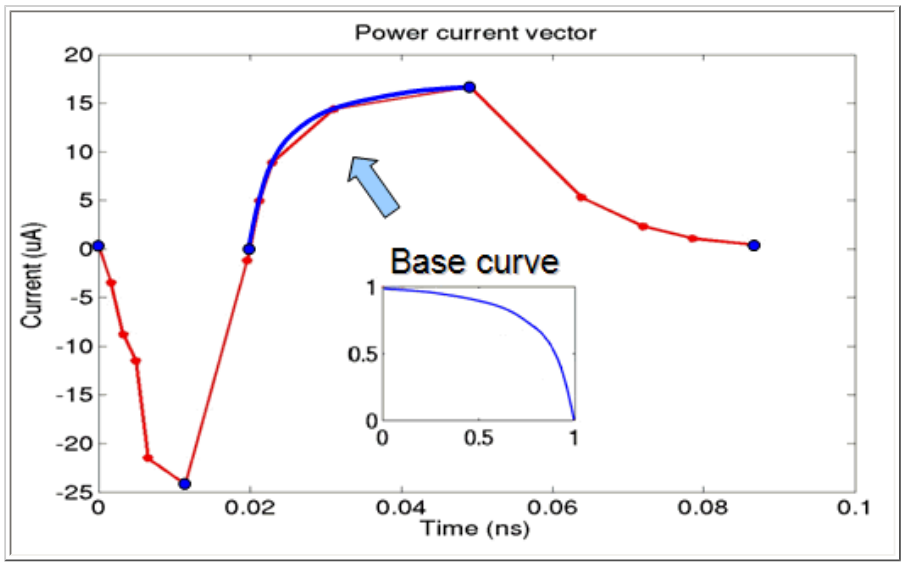
Base curve technology was first introduced for compact CCS timing. Each timing current waveform is split into two segments in the I-V domain, and the shape of each segment is modeled by a base curve that has a similar shape. This segmentation prevents direct modeling with the piecewise linear data points.

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 14-1, 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 14-1 Power Current Vector



Segmenting the waveform and using base curve technology for each segment provides greater accuracy. In Figure 14-1, 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.

14.2.1 Syntax

The expanded, or dynamic, CCS power model syntax provides an important reference and criteria for compact CCS power modeling. See "Dynamic Current Syntax" for the `dynamic_current` syntax and see "Dynamic Power and Ground Current Table Syntax" 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*/
```

14.2.2 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.

14.2.3 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 the `compact_ccs_power` group. For more information, see “compact_lut_template Group”.

The `index_output` attribute is also optional. It is used only on cross type tables. For more information about the `index_output` attribute, see “”.

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 */
}
```

14.3 Composite Current Source Dynamic Power Examples

This section provides the following CCS dynamic power examples:

- Design Cell With a Single Output Example
- Dense Table With Two Output Pins Example
- Cross Type With More Than One Output Pin Example
- Diagonal Type With More Than One Output Pin Example

For more information about the syntax in the following examples, see the Liberty Reference Manual.

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_capacitance(10.0 10.0 10.0 10.0);
    switching_group ( ) {
     input_switching_condition(rise);
     output_switching_condition(rise, fall, fall,
        fall);
     pg_current (VSS) {
      vector ( CCS_power_1 ) {
       index_output : Q;
       reference_time : 0.01;
       index_1 ( 0.01 )
       index_2 ( 5.0)
       index_3 ( 0.000, 0.0873, 0.135, 0.764)
       values ( 0.002, 0.009, 0.134, 0.546)
      }

      vector ( CCS_power_1 ) {
       index_output : QN;
       reference_time : 0.01;
       index_1 ( 0.01 )
       index_2 ( 1.0 )
       index_3 ( 0.000, 0.0873, 0.135, 0.764)
       values ( 0.002, 0.009, 0.134, 0.546)
      }
      vector ( CCS_power_1 ) {
       index_output : QN;
       reference_time : 0.01;
       index_1 ( 0.01 )
       index_2 ( 5.0 )
       index_3 ( 0.000, 0.0873, 0.135, 0.764)
       values ( 0.002, 0.009, 0.134, 0.546)
      }
```

Diagonal Type With More Than One Output Pin Example

```
pg_current_template ( CCS_power_1 ) {
 variable_1 : input_net_transition ;
```

```
 variable_2 : total_output_net_capacitance ;
 variable_3 : time ;
}

   cell ( example )
      dynamic_current() {
       when: D ;
       related_inputs : CP;
       related_outputs : Q QN QN1 QN2;
         switching_group ( ) {
            input_switching_condition(rise);
            output_switching_condition(rise,
fall, fall, fall);
          }
        }
      pg_current (VSS) {
       vector ( CCS_power_1 ) {
          reference_time : 0.01;
        index_1 ( 0.01 )
        index_2 ( 1.0 )
       index_3 ( 0.000, 0.0873, 0.135, 0.764)
       values ( 0.002, 0.009, 0.134, 0.546)
        }
       }
```

15. Modeling Noise

This chapter provides an overview of modeling noise to support gate-level static noise analysis. It covers various topics on modeling noise for calculation, detection, and propagation, in the following sections:

15.1 Modeling Noise Terminology

A net can be either an aggressor or a victim:

- An aggressor net is a net that injects noise onto a victim net.
- A victim net is a net onto which noise is injected by one or more neighboring nets through the cross-coupling capacitors between the nets.

Noise effect can be categorized in two ways:

- Delay noise
- Functional noise

Delay noise occurs when victim and aggressor nets switch simultaneously. This activity alters the delay and slew of the victim net.

Functional noise occurs when a victim net is intended to be at a stable value and the noise injected onto this net causes it to glitch. The glitch might propagate to a state element, such as a latch, altering the circuit state and causing a functional failure.

To compute and detect any delay or functional noise failure, the following are calculated:

- Noise calculation
- Noise immunity

- Noise propagation

15.1.1 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.

15.1.2 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.

15.1.3 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.

15.2 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

15.2.1 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 15-1 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 15-1 Noise Glitch and Steady-State Drive Resistance

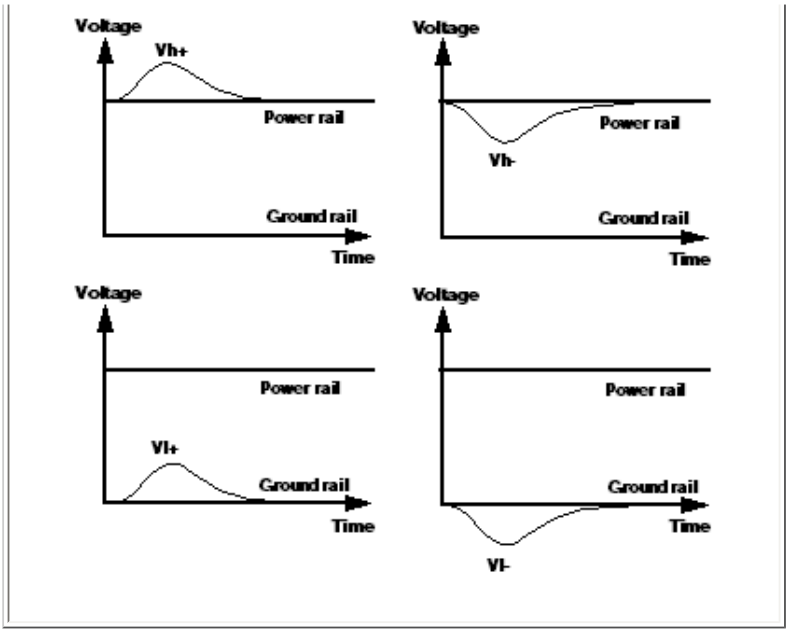
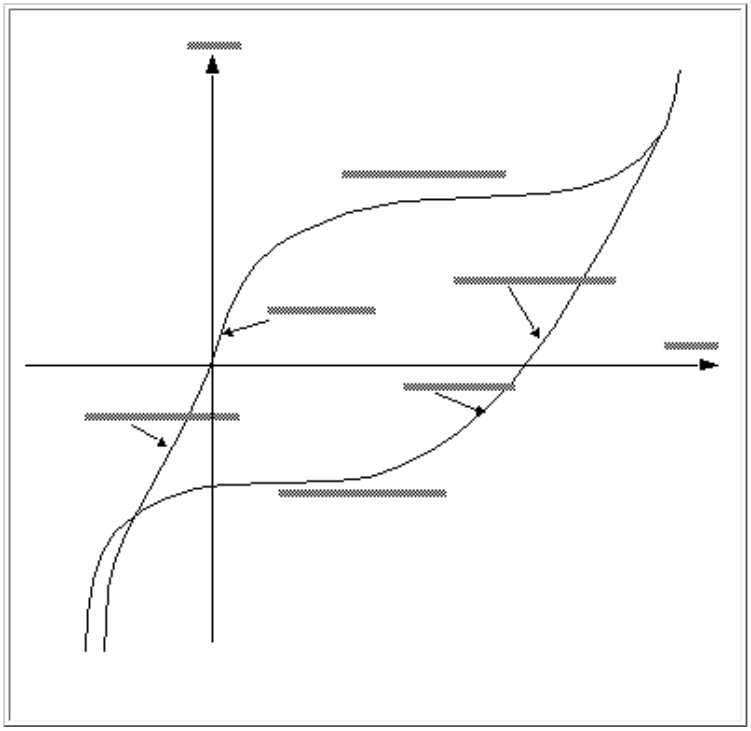


Figure 15-2 is an example of two I-V curves and the steady-state resistance value.

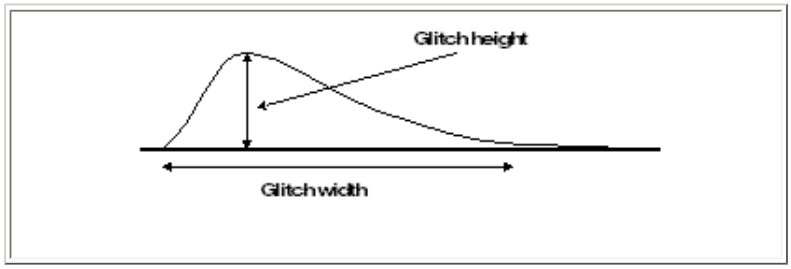
Figure 15-2 I-V Characteristics and Steady-State Drive Resistance



15.2.2 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 15-3 shows a glitch noise model.

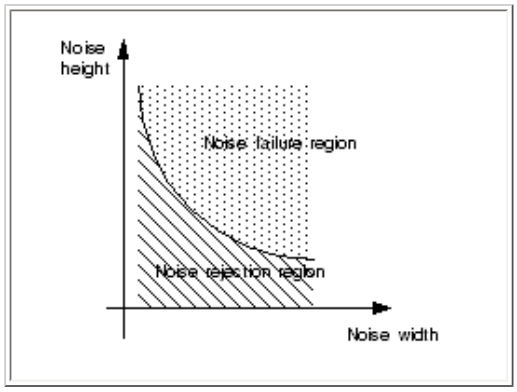
Figure 15-3 Glitch Noise Modeling



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 15-4 shows an example of a noise immunity curve.

Figure 15-4 Noise Immunity Curve

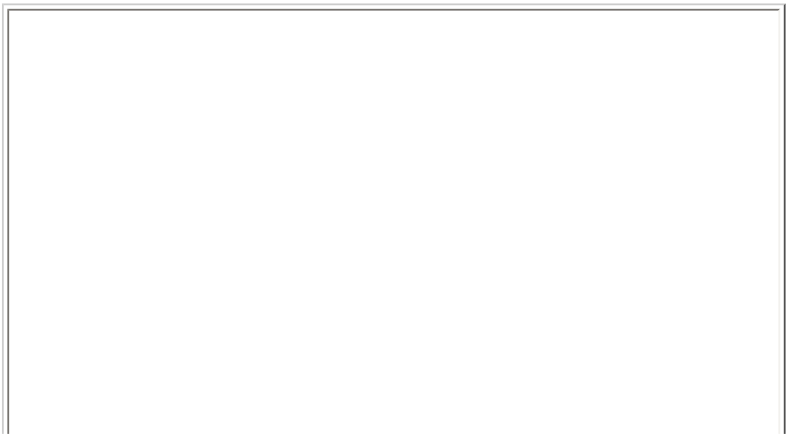


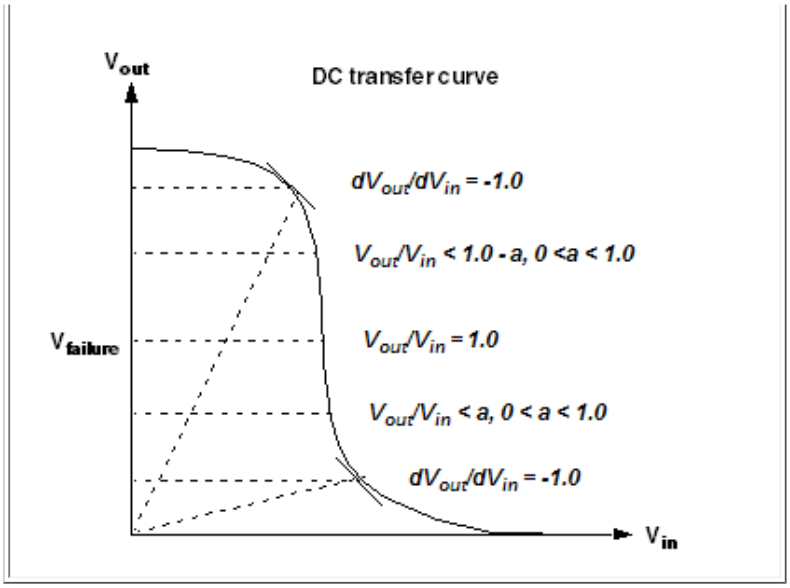
As shown in Figure 15-4, any noise width and height combination that falls above the noise rejection curve causes functional failure.

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 15-5 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.

15.2.3 Using the Hyperbolic Model

The noise immunity curve resembles a hyperbola, because the area of different noise along the hyperbola is constant. Therefore noise immunity can be defined as a hyperbolic function with only three coefficients for every input on an I/O library pin. The formula for the height based on these three coefficients is as follows:

height = height_coefficient + area_coefficient / (width – width_coefficient);

Your tool gets these coefficients from the library and applies the calculated height and width to determine whether the noise can cause functional failure. Any point above the hyperbolic curve signifies a functional failure.

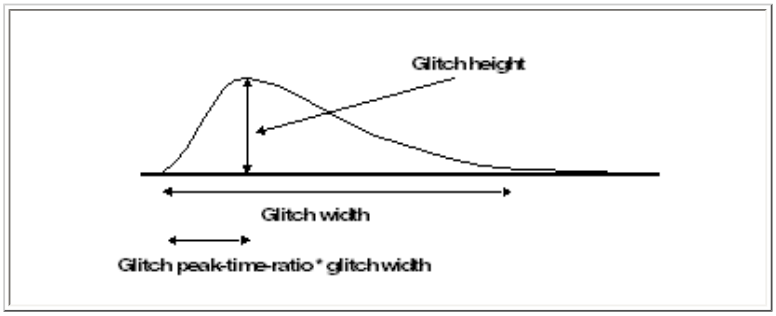
15.2.4 Noise Propagation

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 15-6 illustrates basic noise characteristics.

Figure 15-6 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 15-1 shows a summary of the syntax used to model cases when the cell is not switching.

Table 15-1 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

15.3 Representing Noise Calculation Information

You can represent coupled noise information withan 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

15.3.1 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");
}
```

15.3.2 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");
  }
}
```

15.3.3 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.

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);
  }
}
```

15.3.4 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 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");
  }
  ...
}
```

15.3.5 Using Steady-State Resistance Simple Attributes

To represent steady-state drive resistance values, use the following attributes to define the four regions:

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

These attributes are defined within the `timing` group to represent the steady-state drive resistance. If one of these attributes is missing, the model 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
```

15.3.6 Using I-V Curves and Steady-State Resistance for tied_off Cells

In tied-off cells, the output pins are tied to either high or low and there is no need to define timing information for related pins. The tied-off cells have been enhanced to accept I-V curve and steady-state resistance in the `timing` group. To specify only the noise data (I-V curves and steady-state resistance) in the `timing` group, you must specify a new Boolean attribute, `tied_off`, and set it to true.

15.3.7 Defining tied_off Attribute Usage

You can specify the I-V characteristics and steady-state drive resistance values on tied-off cells by using the `tied_off` attribute in the `timing` group.

Syntax

```
pin(name) {
  ...
  timing() {
```

```
    ...
    tied_off : boolean;
  /* timing type is not defined */
  /* steady-state resistance */
```

The following rules apply to `tied_off` cells:

- Steady-state resistance and I-V curves can coexist in the same timing arc of a `tied_off` output pin.
- If the output pin is tied to low (function : "0") and its timing arc specifies the `steady_state_current_high` group, an error message is generated. Similarly if the output pin is tied to high (function : "1") and its timing arc specifies the `steady_state_current_low` group, an error message is generated.
- If noise immunity and noise propagation are specified in the timing arcs of a `tied_off` pin, an error message is generated.
- If the `related_pin` attribute is specified on a `tied_off` output pin, an error message is generated.

Example

```
pin (high) {
  direction : output;
  capacitance : 0;
  function : "1";

  /* noise information */
  timing() {
    tied_off : true;
    steady_state_resistance_high : 1.22;
    steady_state_resistance_above_high : 1.00;
    steady_state_current_high(iv1x5){
       index_1("0.3,0.75,1.0,1.2,2");
       values("-513.2,-447.9,-359.3,-245.7,497.3");
    }
  }
}
```

15.4 Representing Noise Immunity Information

In the Liberty syntax, you can represent noise immunity information with a

- Lookup table or a polynomial model at the timing level
- Input noise width range at the pin level
- Hyperbolic model at the pin level

15.4.1 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");
}
```

15.4.2 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, your tool might use extrapolation.

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");
     }
    }
  }
}
```

15.4.3 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.

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) {
      ...
   }
   ...
}
```

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");
    }
  }
}
```

15.4.4 Input Noise Width Ranges at the Pin Level

To specify whether a noise immunity or propagation table is referenced within the noise range indexes, the Liberty syntax allows you to specify the minimum and maximum values of the input noise width.

Defining the input_noise_width Range Limits

You can specify two `float` attributes, `min_input_noise_width` and `max_input_noise_width`, at the pin level. These attributes are optional and specify the minimum and maximum values of the input noise width.

Syntax

```
pin(name_string)
{
   ...
   /* used for noise immunity or propagation */       min_input_noise_width
: float;
      max_input_noise_width: float;
   ...
}
```

`float`

The values of `min_input_noise_width` and `max_input_noise_width` are the minimum and maximum input noise width, in library time units.

The following rules apply to `input_noise_width` range limits:

- The `min_input_noise_width` and `max_input_noise_width` attributes can be defined only on input or inout pins. Otherwise, an error message is generated.
- The `min_input_noise_width` and `max_input_noise_width` attributes must both be defined. Otherwise, an error message is generated.
- A check determines whether the `min_input_noise_width <= max_input_noise_width` constraints are met. If the constraints aren't met, an error message is generated.
- Checks do not determine 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 */
```

15.4.5 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;
}
```

15.5 Representing Propagated Noise Information

In the Liberty syntax, you can represent propagated noise information at the timing level by using a

- Propagated Noise Lookup Table Model
- Propagated Noise Polynomial Model

15.5.1 Propagated Noise Lookup Table Model

You can represent propagated noise in your libraries by using lookup tables. To define your lookup tables, use the following groups and attributes:

- `propagation_lut_template` group in the `library` group
- `propagated_noise_height_above_high propagated_noise_height_below_low propagated_noise_height_high propagated_noise_height_low propagated_noise_width_above_high propagated_noise_width_below_low propagated_noise_width_high propagated_noise_width_low groups` in the `timing` group

propagation_lut_template Group

Use this library-level group to create templates of common information that multiple propagation lookup tables can use.

A table template specifies the propagated noise width, height, and output load and their corresponding breakpoints for the axis. Assign each template a name. Make the template name the group name of a propagated noise group.

Syntax

```
library(name_string) {
   ...
   propagation_lut_template(template_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 (tempname_string)          {
   ...
   }
   propagated_noise_height_high(template_name_string) {
      ...
   }
   propagated_noise_height_low(template_name_string) {
      ...
   }
   propagated_noise_width_high(template_name_string)          {
      ...
   }
   propagated_noise_width_low(template_name_string) {
      ...
   }
   ...
}
```

The following rules apply to the propagation noise groups:

- Each of the three pairs of tables is optional; the assumption is that if one pair is missing, the corresponding region does not propagate any noise.
- If a pair of tables for a particular region (high, low, above-high, or below-low) is specified, both width and height must be specified.

- Each propagated noise table has an associated name for the `propatation_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, your tool might use extrapolation.

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");
}
```

15.5.2 Propagated Noise Polynomial Model

As with the lookup table model, you can describe propagated noise in your libraries by using polynomial representation. To define your polynomial, use the following groups:

- The `poly_template` group in the `library` group
- The `propagated_noise_height_above_high`, `propagated_noise_height_below_low`, `propagated_noise_height_high`, `propagated_noise_height_low`, `propagated_noise_width_above_high`, `propagated_noise_width_below_low`, `propagated_noise_width_high`, `propagated_noise_width_low`, `propagated_noise_peak_time_ratio_above_high`, `propagated_noise_peak_time_ratio_below_low`, `propagated_noise_peak_time_ratio_high`, and `propagated_noise_peak_time_ratio_low` groups in the `timing` group

15.5.3 poly_template Group

You can define a `poly_template` group at the library level to specify the equation variables, the variable ranges, the voltage mapping, and the piecewise data. The valid values for the variables are extended to include `input_noise_width`, `input_noise_height`, `input_peak_time_ratio`, `total_output_net_capacitance`, `temperature`, and the related rail voltages.

Syntax

```
library(name_string) {
  ...
  poly_template(template_name_string) {
    variables(variable_i_enum,..., variable_n_enum);
    variable_i_range: (float, float);
    ...
    variable_n_range: (float, float);
    mapping(voltage_enum, power_rail_id);
    domain(domain_name_string) {
      variable_i_range: (float, float);
      ...
      variable_n_range: (float, float);      }
  }
  ...
```

```
}
```

Template Variables

The syntax of the `poly_template` group is the same as that of the delay model, except that the variables used in the format are

- `input_noise_width, input_noise_height, input_peak_time_ratio`
- `total_output_net_capacitance`
- `voltage, voltage` *i* `, temperature`

The piecewise model through the `domain` group is also supported.

The `input_peak_time_ratio` is always specified as a ratio of width, so it is a value between 0.0 and 1.0.

Example

```
poly_template(my_propagated_noise) {
  variables (input_noise_width, input_noise_height, input_peak_time_ratio,
       total_output_net_capacitance);
  variable_1_range (0.01, 2);
  variable_2_range (0, 0.8);
  variable_3_range (0.0, 1.0);
  variable_4_range (0, 2);
} /* poly_template(propagated_noise) */
```

Defining Propagated Noise Groups for Polynomial Representation

To specify polynomial representation, use the `propagated_noise_height_above_high`, `propagated_noise_height_below_low`, `propagated_noise_height_high`, `propagated_noise_height_low`, `propagated_noise_width_above_high`, `propagated_noise_width_below_low`, `propagated_noise_width_high`, `propagated_noise_width_low`, `propagated_noise_peak_time_ratio_above_high`, `propagated_noise_peak_time_ratio_below_low`, `propagated_noise_peak_time_ratio_high`, and `propagated_noise_peak_time_ratio_low` groups within the `timing` group to define the polynomial.

The `peak_time_ratio` groups are supported only in the polynomial model.

Syntax for Polynomial

```
timing() {
    ...
  propagated_noise_height_above_high (temp_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 (tem_name_string)        {
        ...
    }
    propagated_noise_peak_time_ratio_low (temp_name_string) {
    ...
    }
    ...
  }
```

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 ");
}
```

15.6 Examples of Modeling Noise

The examples in this section model libraries for noise extension for scalable polynomials and nonlinear lookup table model libraries.

15.6.1 Scalable Polynomial Model Noise Sample

A scalable polynomial delay library allows you to describe how noise parameters vary with rail voltage and temperature.

```
library (my_noise_lib) {
  delay_model : "polynomial";
  time_unit : "1ns";
  voltage_unit : "1V";
```

```
current_unit : "1mA";
capacitive_load_unit (1,pf);
pulling_resistance_unit : 1kohm;
power_supply() {
  default_power_rail : VDD1;
  power_rail(VDD1, 1.6);
  power_rail(VDD2, 1.3);
}
nom_voltage : 1.0;
nom_temperature : 40;
nom_process : 1.0;
/* Templates of DC noise margins and output levels */
input_voltage(MY_CMOS_IN) {
  vil : 0.3;
  vih : 1.1;
  vimin : -0.3;
  vimax : VDD + 0.3;
}
output_voltage(MY_CMOS_OUT) {
  vol : 0.1;
  voh : 1.4;
  vomin : -0.3;
  vomax : VDD + 0.3;
}
/* Template definitions for noise immunity. Variable:
* input_noise_width */
poly_template ( my_noise_reject ) {
    temperature,total_output_net_capacitance);
  variables ( input_noise_width, voltage, voltage1, \
         temperature,total_output_net_capacitance);
  mapping(voltage, VDD1);
  mapping(voltage1, VDD2);
  variable_1_range (0, 2);
  variable_2_range (1.4, 1.8);
  variable_3_range (1.1, 1.5);
  variable_4_range (-40, 125);
  variable_5_range (0.0, 1.0);
  domain (typ) {
    variables ( input_noise_width, voltage, voltage1,\
        temperature, total_output_net_capacitance);
    variable_1_range (0, 2);
    variable_2_range (1.5, 1.7);
    variable_3_range (1.2, 1.4);
    variable_4_range (25, 25);
    variable_5_range (0.0, 1.0);
    mapping(voltage, VDD1);
    mapping(voltage1, VDD2);
  }
  domain (min) {
    variables ( input_noise_width, voltage, voltage1, \
           temperature );
    variable_1_range (0, 2);
    variable_2_range (1.7, 1.8);
    variable_3_range (1.4, 1.5);
    variable_4_range (-40, -40);
    mapping(voltage, VDD1);
    mapping(voltage1, VDD2);
  }
  domain (max) {
    variables ( input_noise_width, voltage, voltage1, \
           temperature );
    variable_1_range (0, 2);
    variable_2_range (1.6, 1.7);
    variable_3_range (1.1, 1.2);
    variable_4_range (125, 125);
    mapping(voltage, VDD1);
    mapping(voltage1, VDD2);
  }
} /* end poly_template (my_noise_reject) */
poly_template ( my_noise_reject_outside_rail ) {
  variables ( input_noise_width, voltage, voltage1, \
```

```
          temperature );
  mapping(voltage, VDD1);
  mapping(voltage1, VDD2);
  variable_1_range (0, 2);
  variable_2_range (1.4, 1.8);
  variable_3_range (1.1, 1.5);
  variable_4_range (-40, 125);
} /* end poly_template ( my_noise_reject_outside_rail ) */
/* Template definitions for I-V characteristics. Variable:
* iv_output_voltage */
poly_template ( my_current_low ) {
  variables ( iv_output_voltage, voltage, voltage1, \
          temperature );
  mapping(voltage, VDD1);
  mapping(voltage1, VDD2);
  variable_1_range (-1, 2);
  variable_2_range (1.4, 1.8);
  variable_3_range (1.1, 1.5);
  variable_4_range (-40, 125);
} /* end poly_template ( my_current_low ) */
poly_template ( my_current_high ) {
  variables ( iv_output_voltage, voltage, voltage1, \
          temperature );
  mapping(voltage, VDD1);
  mapping(voltage1, VDD2);
  variable_1_range (-1, 2);
  variable_2_range (1.4, 1.8);
  variable_3_range (1.1, 1.5);
  variable_4_range (-40, 125);
} /* end poly_template ( my_current_high ) */
/* Template definitions for propagated noise. Variables:
* input_noise_width
* input_noise_height
* input_peak_time_ratio
* total_output_net_capacitance */
poly_template(my_propagated_noise) {
  variables ( input_noise_width, input_noise_height, \
          input_peak_time_ratio \
          total_output_net_capacitance, voltage, \
          voltage1, temperature );
  mapping(voltage, VDD1);
  mapping(voltage1, VDD2);
   variable_1_range (0.01, 2);
  variable_2_range (0, 0.8);
  variable_3_range (0.0, 1.0);
  variable_4_range (0, 2);
  variable_5_range (1.4, 1.8);
  variable_6_range (1.1, 1.5);
  variable_7_range (-40, 125);
} /* end poly_template (my_propagated_noise) */
/* INVERTER */
cell ( INV ) {
  area : 1 ;
  pin ( A ) {
    direction : input ;
    capacitance : 1 ;
    fanout_load : 1 ;
      /* DC noise margins.
      * These are used for compatibility of level shifters.
      * In noise analysis, they are the least accurate way
      * to define noise margins.
      * Compatible: can coexist in the pin group with any
      * other noise margin definition. */
    input_voltage : MY_CMOS_IN ;
      /* Noise group defines what is acceptable noise on input
      * pins. */
      /* Hyperbolic noise immunity.
      * Another way to specify noise immunity.
      * Mutually exclusive: noise_immunity_low cannot be
      * together with
      *hyperbolic_noise_immunity_low, and so on.
```

```
    * Defines pulse_height = height_coefficient +
    * area_coefficient / (width - width_coefficient)
    * Characterization recommendation: Use
    * hyperbolic_noise_immunity_*
    * if it can fit the curve, otherwise use
    * table noise_immunity_* */
  hyperbolic_noise_low() {
    height_coefficient : 0.4;
    area_coefficient : 1.1;
    width_coefficient : 0.1;
  }
  hyperbolic_noise_high() {
    height_coefficient : 0.3;
    area_coefficient : 0.9;
    width_coefficient : 0.1;
  }
  hyperbolic_noise_below_low() {
    height_coefficient : 0.1;
    area_coefficient : 0.3;
    width_coefficient : 0.01;
  }
  hyperbolic_noise_above_high() {
    height_coefficient : 0.1;
    area_coefficient : 0.3;
    width_coefficient : 0.01;
  }
} /* end pin (A) */
pin ( Y ) {
  direction : output ;
  max_fanout : 10 ;
  function : " !A ";
  output_voltage : MY_CMOS_OUT ;
  timing () {
    related_pin : A ;
    /* Steady state drive resistance */
    steady_state_resistance_high : 1500;
    steady_state_resistance_low : 1100;
    steady_state_resistance_above_high : 200;
    steady_state_resistance_below_low : 100;
    /* I-V curve.
    * Describes how much current the pin can deliver in a given state for
    * a given voltage on the pin.
    * Voltage is measured from the pin to ground, current is measured
    * flowing into the cell (both can be either positive or negative). */
    steady_state_current_low(my_current_low) {
      orders ("3, 3, 0, 0");
      coefs ("8.4165, 0.3198, -0.0004, 0.0000, \
      1133.8274, 8.7287, -0.0054, 0.0000, \
      139.8645, -60.3898, 0.0589, -0.0000, \
      -167.4473, 95.7112, -0.1018, 0.0000");
    }
    steady_state_current_high(my_current_high) {
      orders ("3, 3, 0, 0");
      coefs ("10.9165, 0.2198, -0.0003, 0.0000, \
      1433.8274, 8.7287, -0.0054, 0.0000, \
      128.8645, -60.3898, 0.0589, -0.0000, \
      -167.4473, 95.7112, -0.1018, 0.0000");
    }
    /* Noise immunity.
    * Defines maximum allowed noise height for given pulse width.
    * Pulse height is absolute value from the signal level.
    * Any of the 4 tables below are optional. */
    noise_immunity_low (my_noise_reject) {
      domain (typ) {
        orders ("3, 3, 0, 0, 0");
        coefs ("11.4165, 0.2198, -0.0003, 0.0000, \
        1353.8274, 8.7287, -0.0054, 0.0000, \
        149.8645, -60.3898, 0.0589, -0.0000, \
        -167.4473, 95.7112, -0.1018, 0.0000");
      }
      domain (min) {
```

```
        orders ("3, 3, 0, 0");
        coefs ("6.964065, 0.134078, -0.000183, 0.0000, \
        825.834714, 5.324507, -0.003294, 0.0000, \
        91.417345, -36.837778, .035929, -0.0000, \
        -102.142853, 58.383832, -.062098, 0.0000");
      }
      domain (max) {
        orders ("3, 3, 0, 0");
        coefs ("19.065555, 0.367066, -0.000501, 0.0000, \
        2260.891758, 14.576929, -0.009018, 0.0000, \
        250.273715, -100.850966, 0.098363, -0.0000, \
        -279.636991, 159.837704, -0.170006, 0.0000");
      }
    }
    noise_immunity_high (my_noise_reject) {
      domain (typ) {
        orders ("3, 3, 0, 0, 0");
        coefs ("12.4165, 0.2198, -0.0003, 0.0000, \
        1353.8274, 8.7287, -0.0054, 0.0000, \
        129.8645, -60.3898, 0.0589, -0.0000, \
        -147.4473, 95.7112, -0.1018, 0.0000");
      }
      domain (min) {
        orders ("3, 3, 0, 0");
        coefs ("6.364065, 0.134078, -0.000183, 0.0000, \
        845.834714, 5.324507, -0.003294, 0.0000, \
        91.417345, -36.837778, .035929, -0.0000, \
        -103.142853, 58.383832, -.062098, 0.0000");
      }
      domain (max) {
        orders ("3, 3, 0, 0");
        coefs ("19.265555, 0.367066, -0.000601, 0.0000, \
        2460.891758, 14.576929, -0.009018, 0.0000, \
        250.273715, -130.850966, 0.098363, -0.0000, \
        -279.636991, 159.837704, -0.170006, 0.0000");
      }
    }
    noise_immunity_below_low (my_noise_reject_outside_rail) {
      orders ("3, 3, 0, 0");
      coefs ("10.4165, 0.1198, -0.0003, 0.0000, \
      1333.8274, 8.7287, -0.0054, 0.0000, \
      149.8645, -60.3898, 0.0589, -0.0000, \
      -167.4473, 95.7112, -0.1018, 0.0000");
    }
    noise_immunity_above_high (my_noise_reject_outside_rail) {
      orders ("3, 3, 0, 0");
      coefs ("12.4165, 0.2298, -0.0003, 0.0000, \
      1253.8274, 8.7287, -0.0054, 0.0000, \
      149.8645, -60.3898, 0.0589, -0.0000, \
      -167.4473, 95.7112, -0.1018, 0.0000");
    }
  /* Propagated noise.
    * It is a function of input noise width and height and output
    * capacitance. Width and height are in separate tables. */
    propagated_noise_width_high(my_propagated_noise) {
      orders ("1, 2, 1, 0, 0, 0, 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)
```

15.6.2 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)
```

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X