Library Compiler™ Methodology and Modeling Functionality in Technology Libraries User Guide

Version E-2010.12, March 2011

SYNOPSYS®

Contents

Preface

This preface includes the following sections:

What’s New in This Release

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

To see the *Library Compiler Release Notes*,

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

   https://solvnet.synopsys.com/DownloadCenter

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

About This Guide

Library Compiler captures ASIC libraries and translates them into Synopsys internal database format for physical synthesis or into VHDL format for simulation. The Library Compiler documentation includes three reference manuals and three user guides.

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

Audience

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

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

Related Publications

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

https://solvnet.synopsys.com/DocsOnWeb

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

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

Conventions

The following conventions are used in Synopsys documentation.

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

Customer Support

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

Accessing SolvNet

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

To access SolvNet, go to the following address:

https://solvnet.synopsys.com

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

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

Contacting the Synopsys Technical Support Center

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

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

1

Introduction to Library Compiler

Library Compiler reads the description of an ASIC library from a text file and compiles the description into either an internal database (.db format) or into VHDL libraries.

The compiled database supports synthesis tools. The VHDL libraries support VHDL simulation tools.

You access Library Compiler through the lc_shell command interface. For information about using the lc_shell command interface, see Chapter 3, “Using the Library Compiler Shell Interface."

Also, see the Synopsys man pages and the *Design Compiler Reference Manual* for descriptions of the lc_shell commands that affect Library Compiler.

This chapter describes:

- The Role of Library Compiler
- Technology Libraries
- Symbol Libraries
- Compiled Libraries and Design Compiler
- VHDL Simulation Libraries and the VHDL Simulator

The Role of Library Compiler

The role of Library Compiler is to generate two types of libraries that support Synopsys synthesis and simulation products. The two types of libraries are

- Synthesis libraries
- VHDL simulation ASCII libraries

Synthesis Libraries

The synthesis libraries consist of the technology libraries, physical libraries, and symbol libraries that contain the types of information required to describe ASIC components: technical and physical characteristics and schematic symbols.

- Technology libraries contain information about the characteristics and functions of each component in an ASIC library. Information stored in the technology libraries consists of area, timing, function, and so on. The Synopsys design tools use this information to make synthesis decisions.
- Symbol libraries contain information about the schematic symbols that represent each ASIC component, as well as information about special symbols, such as page borders and off-sheet connectors. Using the Design Analyzer tool, you can then
  - Generate schematics of designs
  - Display the schematic on the computer screen
  - Draw the designs, using your PostScript plotter or printer
- For information about physical libraries, see the *Library Compiler Physical Libraries Reference Manual* and the *Library Compiler Physical Libraries User Guide*.

VHDL Simulation ASCII Libraries

Library Compiler can generate VHDL libraries that contain the timing and functional information needed for simulation.

Creating Libraries

You can create technology libraries as well as symbol libraries by writing ASCII text descriptions.

Note:
The `include_file` attribute lets you reduce file size and improve memory management by subdividing your technology library source file into several files. For more information, see the *Library Compiler Technology and Symbol Libraries Reference Manual*.

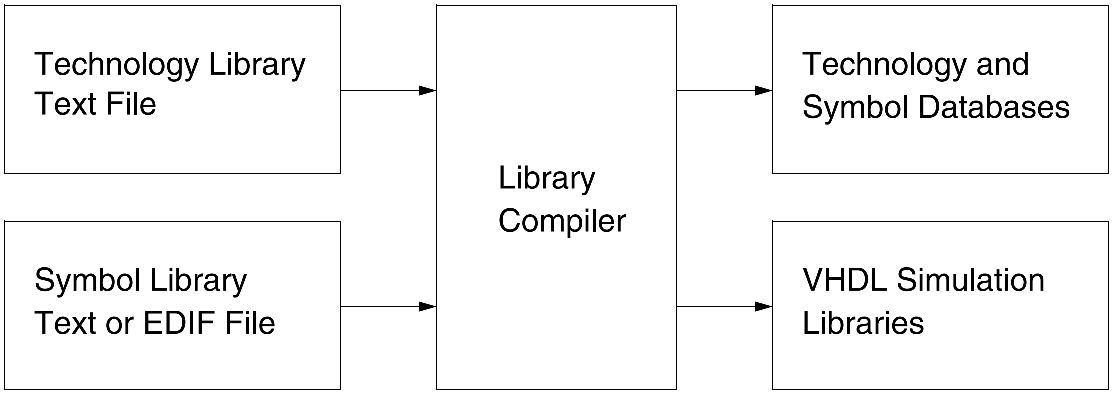
You can also create symbol libraries by transferring information from a variety of computer-aided engineering (CAE) systems that use the EDIF format.

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

You can also use Library Compiler to write out VHDL simulation libraries.

Figure 1-1 shows the role of Library Compiler.

Figure 1-1 The Role of Library Compiler


Technology Library Text File
Symbol Library Text or EDIF File
Library Compiler
Technology and Symbol Databases
VHDL Simulation Libraries

Technology Libraries

A technology library is a text file that contains four kinds of information about the ASIC technology:

Structural information

Describes each cell's connectivity to the outside world, including cell, bus, and pin descriptions.

Functional information

Describes the logical function of every output pin of every cell so that Design Compiler can map the logic of a design to the actual ASIC technology you are using. Cells that do not have a function described in the technology library are left untouched during optimization.

Timing information

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

Environmental information

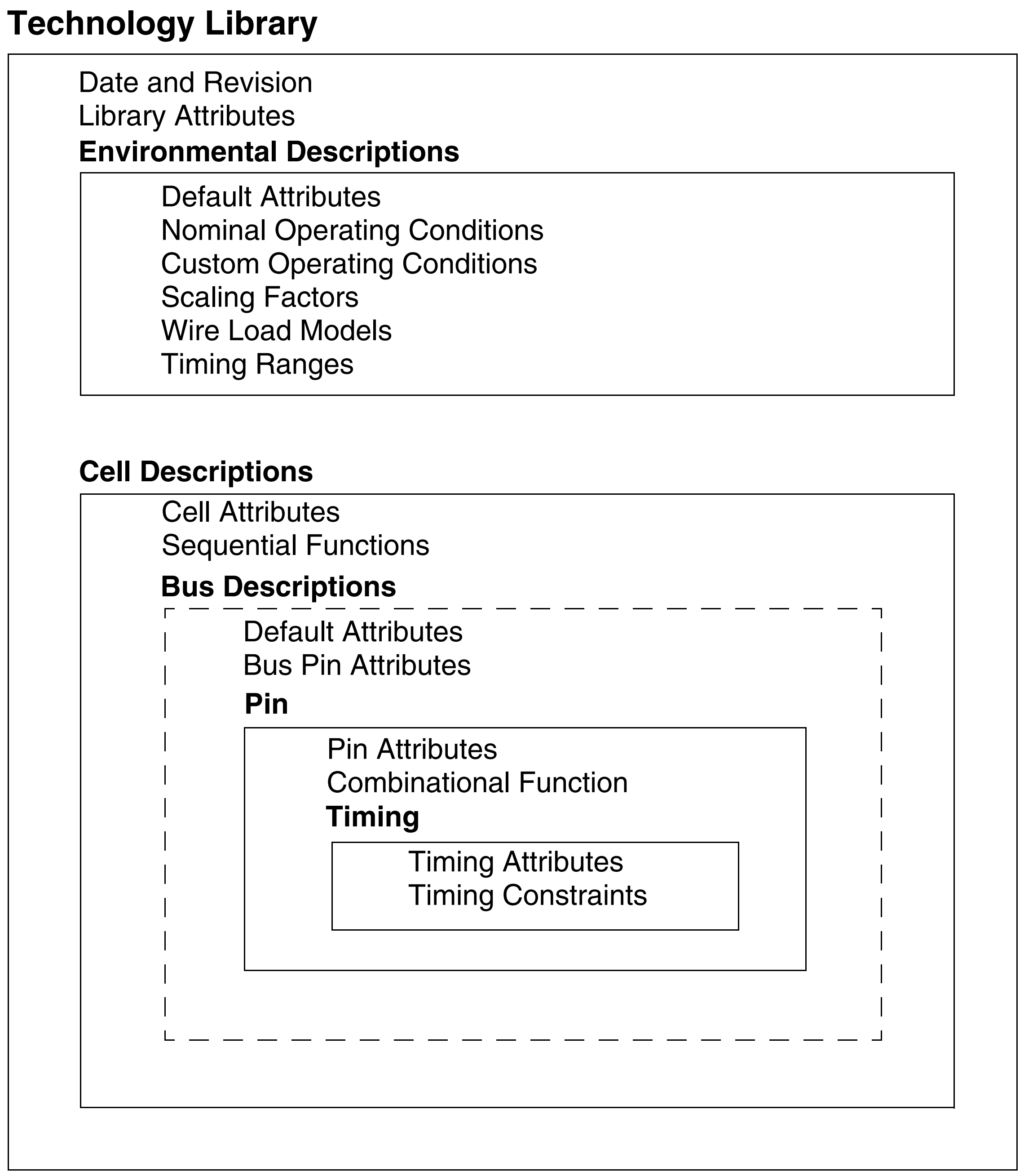
Describes the manufacturing process, operating temperature, supply voltage variations, and design layout, all of which directly affect the efficiency of every design.

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

Figure 1-2 shows the structure of a technology library. A technology library contains cell and environment descriptions:

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

Figure 1-2 Technology Library Structure



Symbol Libraries

Figure 1-3 shows the structure of a symbol library, which is similar to the structure of a technology library. For Design Analyzer to display your designs, each component of the ASIC technology must have a graphic representation—a schematic symbol in a symbol library. Also, several special symbols in a symbol library describe such things as off-sheet connectors, sheet border templates, and default logic symbols.

Figure 1-3 Symbol Library Structure

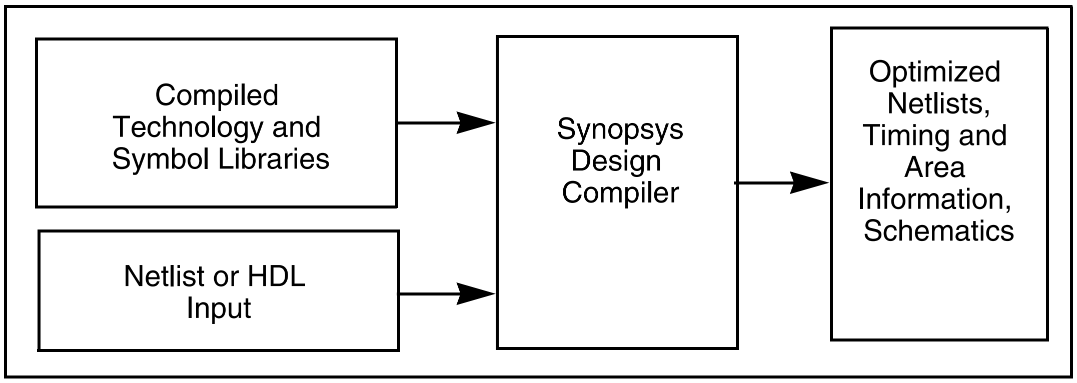
- Variables
- Default Special Symbols
- **Layers**
  - Fonts
  - Color
  - Line Widths
  - Visibility
- **Symbols**
  - Pins
  - Subsymbols
  - Drawing Instructions
  - **Templates**
    - Usable Area
    - Orientation
    - Drawing Instructions

Compiled Libraries and Design Compiler

You can use a compiled technology or symbol library with Design Compiler to generate designs and schematics, as shown in Figure 1-4.

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

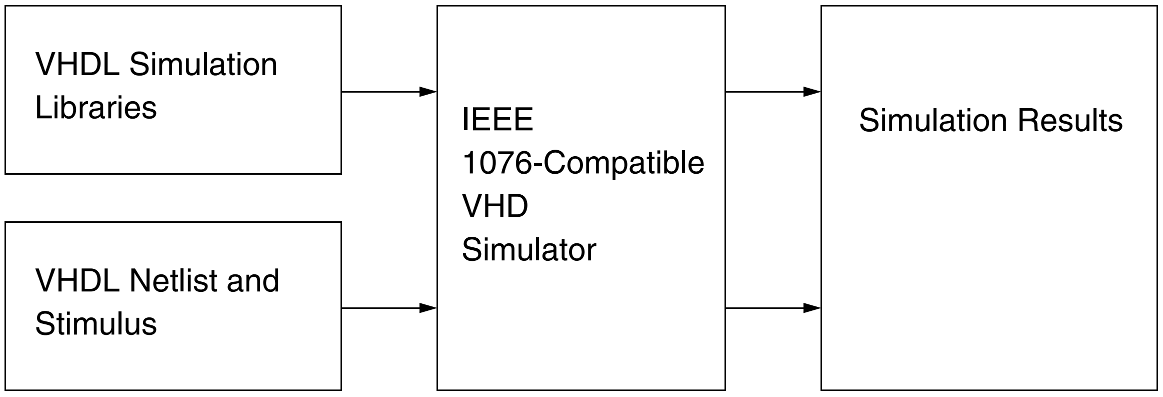
Figure 1-4 Using the Compiled Libraries With Design Compiler


Compiled Technology and Symbol Libraries
Netlist or HDL Input
Synopsys Design Compiler
Optimized Netlists, Timing and Area Information, Schematics

VHDL Simulation Libraries and the VHDL Simulator

Library Compiler generates a VHDL simulation library that a VHDL simulator uses with netlist information to simulate your design, as shown in Figure 1-5.

Figure 1-5 Using VHDL Libraries with a VHDL Simulator


VHDL Simulation Libraries
VHDL Netlist and Stimulus
IEEE 1076-Compatible VHD Simulator
Simulation Results

2

Library Development Procedure

This chapter describes the general procedure for creating technology libraries. It provides an overview of library development tasks and refers you to chapters in this guide and to other Library Compiler manuals for specific information.

This chapter includes the following sections:

Library Development Overview

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

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

Library development consists of the following major activities:

1. Describing a library in text format (.lib or .slib). This chapter describes this activity.
2. Compiling a binary form of the library (.db or .sdb). This chapter describes this activity.
3. Using the compiled library with `dc_shell` during optimization. See the *Design Compiler Reference Manual: Optimization and Timing Analysis* for a description of this activity.

These are the major tasks in developing a library in text format and compiling the library into binary format; each task is described in a separate section in this chapter.

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

For specific information about the library syntax, see the *Library Compiler Technology and Symbol Libraries Reference Manual*.

For information about VHDL libraries and simulation, see the *Library Compiler VHDL Libraries Reference Manual*.

Task 1: Understanding Language Concepts

Information in a library is contained in the Library Compiler language statements. Library statements consist of these types: group, simple attribute, and complex attribute. Each statement type has its own syntax. Chapter 4, “Using Library Compiler Syntax,” gives details about these types and define statements, which are complex attributes that you can use to create user-defined simple attributes.

References

For more information about the Library Compiler language, see

- Chapter 4, “Using Library Compiler Syntax”
- Chapter 5, “Building a Symbol Library”
- Chapter 6, “Building a Technology Library”
- *Library Compiler Technology and Symbol Libraries Reference Manual*

Basic Syntax Rules

These are the Library Compiler syntax rules:

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

Group Statements

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

This is the syntax of a group statement:

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

Attribute Statements

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

This is the syntax of a simple attribute:

```
attribute_name : attribute_value
```

This is the syntax of a complex attribute:

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

Task 2: Creating a Library

The examples in this chapter describe the development of a technology library. To create a symbol library, see Chapter 5, “Building a Symbol Library.”

References

For more information about creating a library, see

- Chapter 4, “Using Library Compiler Syntax”
- Chapter 6, “Building a Technology Library”
- *Library Compiler Technology and Symbol Libraries Reference Manual*

Procedure for Creating a Library

These are the steps for creating a library:

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

Determining the Technology

The technology can be CMOS (default) or FPGA. Note that an FPGA library requires a special license during compilation.

Selecting a Delay Model

Select the model to be used in delay calculations. The supported delay models are

generic_cmos

This delay model is also referred to as the standard delay model or the CMOS linear delay model. It is the default for CMOS technology.

table_lookup

This is the CMOS nonlinear delay model.

polynomial

This is the CMOS scalable polynomial delay model.

piecewise_cmos

This is the CMOS piecewise linear model.

dcm

If you select the dcm (Delay Calculation Module) delay model, Library Compiler does not save any time-related groups or attributes in the .db library. Application tools use the DCM libraries generated by the Delay Calculation Language program to obtain timing-related information.

Determining Where This Library Is Used

Determine whether this library will be used for synthesis or for simulation. This chapter describes a synthesis library. See the *Library Compiler VHDL Libraries Reference Manual* for information about creating a simulation library.

Describing the Library

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

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

Naming the Library

Use these extensions for library file names:

.lib	Technology library source files
.plib	Physical library source files
.pplib	Pseudo-physical library source files
.slib	Symbol library source files
.db	Compiled technology libraries in Synopsys database format
.pdb	Compiled physical libraries in Synopsys database format
.sdb	Compiled symbol libraries in Synopsys database format
.vhd	Generated VHDL simulation libraries

Defining the library Group

The `library` group statement is required and must be the first executable line in your library source file. Although it’s a good idea to use the same name for the library that the vendor uses, the names do not have to be the same.

Example

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

```
library(example) {
      ...
}
```

Specifying library Group Attributes

Some simple attributes at the library level define features of the library, such as units for time, voltage, current, capacitive load, delay model, bus naming style, and documentation. For information on bus naming style, see “bus_naming_style Attribute” on page 6-5.

A complex attribute for the `library` group defines the technology, unit for capacitive load, piecewise linear delay, or routing layer. For information on the routing layer, see "routing_layers Attribute" on page 6-5 and Task 3: Describing the Environment.

technology Attribute

The `technology` attribute in the `library` group defines the technology used for the library. If you define the technology (rather than accept the default), the `technology` attribute must be the first attribute you define.

delay_model Attribute

The `delay_model` attribute defines the delay model used in the library. If you use a delay model that is not a default, it must be the first attribute you define after the `technology` attribute.

Examples

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

```
library(example) {
   technology(cmos);
   delay_model : generic_cmos;
}
```

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

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

Task 3: Describing the Environment

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

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

References

For more information about describing the environment, see

- Chapter 6, “Building a Technology Library”
- The “Building Environments” chapter in the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide*

Procedure

These are the steps for using the Library Compiler syntax to describe the environment:

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

Modeling Delay Calculation Scaling Factors

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

The scaling factors you define for your library depend on the timing delay model you use. See the “Delay Models” and “Timing Arcs” chapters in the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide* for information about the delay analysis equations that Design Compiler uses.

The k-factors consist of the following:

- Calculating delay factors
- Intrinsic delay factors
- Slope-sensitivity factors (standard delay model)
- Drive capability factors (standard delay model)
- Pin and wire capacitance factors
- Wire-resistance factors
- Pin-resistance factors (piecewise linear delay model)
- Intercept delay factors (piecewise linear delay model)

- Power scaling factors
- Timing constraint factors

Example

```
library(example) {
   k_process_drive_fall :  1.0;
   k_process_drive_rise :  1.0;
}
```

See the “Building Environments” chapter in the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide* for a description of delay scaling attributes.

Defining Default Attributes

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

You can set global default attributes at the library level for pins, timing, wire load, chip utilization, routability, operating conditions group, and cell leakage power. See the “Building Environments” chapter in the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide* for a description of library-level default attributes.

Example

```
library(example) {
   default_intrinsic_fall :  1.0;
   default_intrinsic_rise :  1.0;
}
```

Defining Operating Conditions

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

Example

You define operating conditions in an `operating_conditions` group within the `library` group.

```
library (example) {
   operating_conditions (WCCOM) {
      process : 1.5 ;
      temperature : 70 ;
```

```
        voltage : 4.75 ;
        tree_type : worst_case_tree ;
        power_rail(v1, 2.5);
        power_rail(v2, 3.9) ;
    }
}
```

name

The name (WCCOM in the example) identifies the set of operating conditions.

process

The scaling factor accounts for variations in the outcome of the actual semiconductor manufacturing steps. This factor is typically 1.0 for normal operating conditions.

temperature

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

voltage

The operating voltage of the design.

tree_type

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

power_rail

The voltage value for a power supply.

At optimization, Design Compiler selects the operating conditions to use, in this order of precedence:

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

Modeling the Wire Load

To provide the information that Design Compiler needs to estimate interconnect wiring delays, use the `wire_load` group and `wire_load_selection` group. These groups define the estimated wire length as a function of fanout and the scaling factors when determining wire resistance, capacitance, and area for a given length of wire.

Specifying Values for the 3-D Wire Delay Lookup Table

The following is the value set that you can assign for `variable_1`, `variable_2`, and `variable_3` to the templates for wire delay tables:

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

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

Example

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

name

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

resistance

The wire resistance per unit length of interconnect wire.

capacitance

The capacitance per unit length of interconnect wire.

area

The area per unit length of interconnect wire.

slope

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

fanout_length

The values that represent fanout and length. The first value is an integer specifying the total number of pins minus one on the net being driven by the given output. The second value is the estimated amount of metal that is statistically found on a network with the given number of pins.

interconnect_delay

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

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

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

See the “Building Environments” chapter in the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide* for more information about defining wire loads and selecting `wire_load` groups automatically.

Specifying the 2-D Net Delay Lookup Table

The net delay tables that use the template given above should be given on library level. Their names are

- rise_net_delay
- fall_net_delay

Index overwrite is also feasible by providing the indexes before specifying the net delay values.

Example

```
lu_table_template(net_delay_table_template) {
   variable_1 : output_pin_transition;
   variable_2 : rc_product;
   index_1 ("1.0 , 3.0");
   index_2 ("0.12, 4.24");
}
lu_table_template(trans_template) {
   variable_1 : total_output_net_capacitance;
   index_1 ("0.0, 1.5", "2.0, 2.5");
}
rise_net_delay(net_delay_table_template) {
   values("0.00 , 0.21", "0.11 , 0.23");
}
fall_net_delay(net_delay_table_template) {
   values("0.00 , 0.57", "0.10 , 0.48");
}
```

Task 4: Defining the Cells

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

Structure

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

Function

The logical function of every output pin of each cell that Design Compiler uses to map the logic of a design to the actual ASIC technology.

Timing

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

Power

Modeling for state-dependent and path-dependent power (for information, see the “Modeling Power and Electromigration” chapter in the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide*).

Other synthesis parameters

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

Note:

Technology libraries are processed without regard to explicitly specified units of measure for delay, capacitance, resistance, voltage, temperature, or area. Choose consistent units of measure (nanoseconds, picofarads, kilohms, volts, and degrees centigrade) throughout the library.

The units used to describe area depend on the ASIC technology. For gate arrays, the value of area for a given cell is probably the number of gates it uses (physical gates or gate equivalents). Standard cells and cell-based custom designs generally use a measure of surface area such as square micrometers. Pads can be specified as having zero area, because they do not occupy area in the logic core of the chip.

References

See these sources for more information about defining cells:

- Chapter 4, “Using Library Compiler Syntax”
- Chapter 7, “Defining Core Cells”
- Chapter 8, “Defining Sequential Cells”
- Chapter 9, “Defining I/O Pads”
- The “Modeling Power,” “Timing Arcs,” and “Interface Timing of Complex Sequential Blocks” chapters in the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide*
- *Library Compiler Technology and Symbol Libraries Reference Manual*

Procedure

Defining Cells

You can divide cells into two types: core cells and input/output pads. These are the activities in defining cells:

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

Defining Core Cells

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

The minimum cell set for a CMOS technology library is:

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

The example libraries contain these cells:

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

To define core cells

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

Referring to the Datasheet

Although each company has its own way of describing components in databooks, most datasheets contain information about area, capacitance, intrinsic delays, and extrinsic delays based on loading.

The unit of measure for capacitance does not matter to Design Compiler, but you should make sure that the units you select are consistent in the library and with the timing equation factors used to describe the library. The units used to measure capacitance are usually standard load units or picofarads.

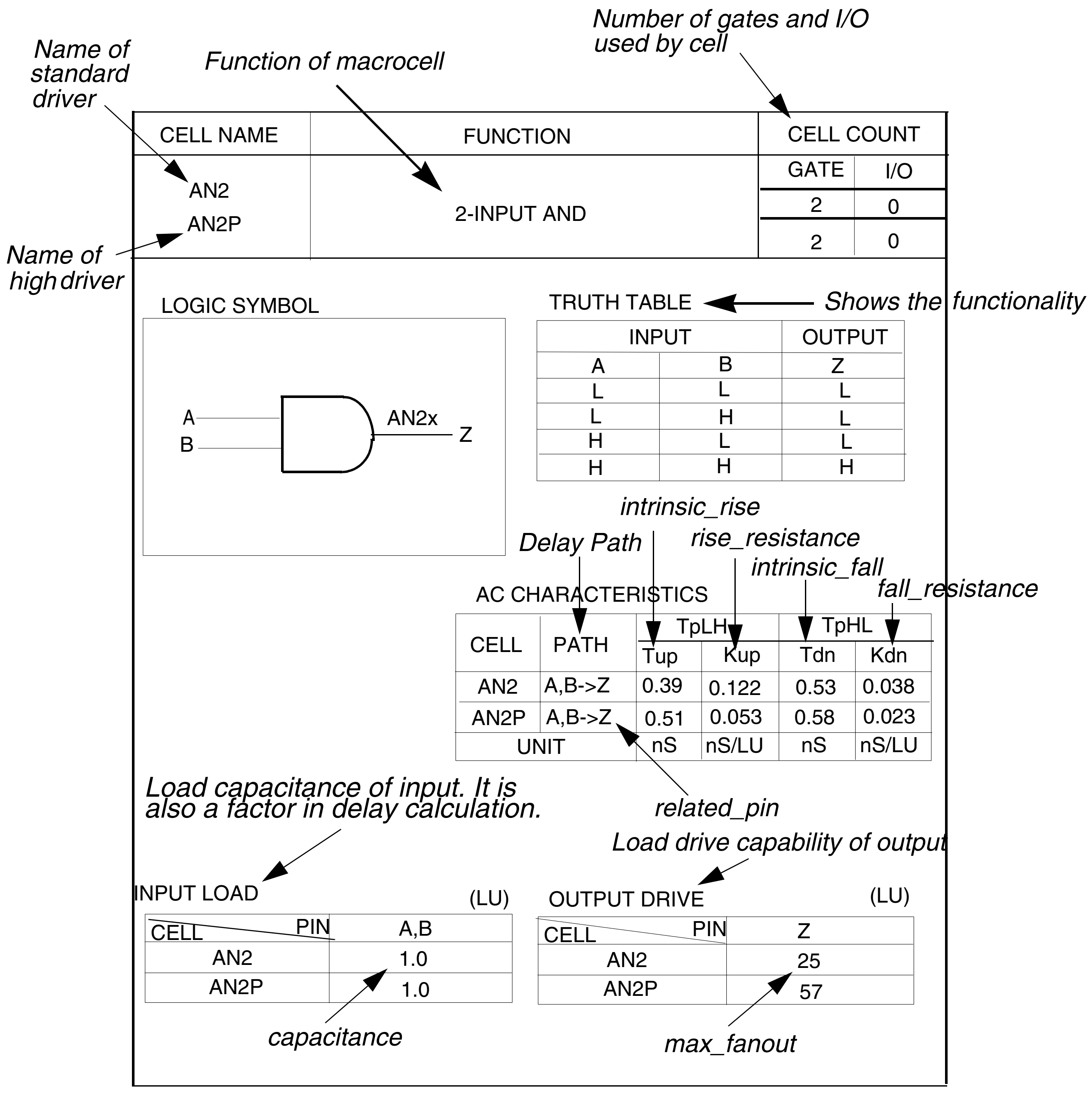
The intrinsic rise and intrinsic fall of a component are the rise and fall delays through a macrocell, regardless of delays caused by layout, loading, and environment.

Rise resistance and fall resistance are factors that calculate the extrinsic delay based on loading when they are multiplied by the capacitive load on the output pin. The units for rise and fall resistance are a value of time per a unit of load.

Figure 2-1 shows an example datasheet for a 2-input AND gate. The information on this datasheet can be described in a library that uses a generic_cmos delay model. If you described this cell in a library that used the nonlinear delay model, you would use the values that were characterized by a SPICE tool.

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

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



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

TRUTH TABLE

INPUT		OUTPUT
A	B	Z
L	L	L
L	H	L
H	L	L
H	H	H

AC CHARACTERISTICS

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

INPUT LOAD (LU)

CELL \ PIN	A,B
AN2	1.0
AN2P	1.0

OUTPUT DRIVE (LU)

CELL \ PIN	Z
AN2	25
AN2P	57

Specifying the Cell Name

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

Example

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

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

area

Defines the cell area. The default area is zero. You should specify an area for the cell if you want to use this library for area optimization. The exception is pad cells, which have zero area, because they are not used as internal gates.

Listing the Pins

A `cell` group must contain descriptions of all pins in the cell. You define pins in `pin` groups within a `cell` group.

Example

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

```
  pin(A,B) {
    direction : input ;
    capacitance : 1 ;
  }
  pin(Z) {
    direction : output ;
    function : "A * B" ;
    timing() {
      intrinsic_rise : 0.39 ;
      intrinsic_fall : 0.53 ;
      rise_resistance : 0.122 ;
      fall_resistance : 0.038 ;
      related_pin : "A B" ;
    }
  }
}
```

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

direction

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

capacitance

Defines the input pin load (input capacitance) placed on the network. Load units should be consistent with other capacitance specifications throughout the library. Typical units of measure for capacitance are picofarads and standardized loads.

function

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

timing

Describes `timing` groups. The `timing` groups describe the following:

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

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

See "Task 3: Describing the Environment" on page 2-7 for more information about timing.

Determining Pin Functions

The function of a cell is the logical (Boolean) operation of a cell's output pin in terms of its input pins and state. A cell's function can be either combinational or sequential.

Combinational cell logic depends only on the input states. An example of a combinational cell is a NAND gate.

In contrast, sequential cell logic is made up of storage or register elements. Examples of storage elements are flip-flops and latches.

In combinational cells, each output pin has its function specified by a function attribute containing a Boolean expression.

Sequential cell behavior is defined with one of these groups:

- `ff`—for a flip-flop
- `latch`—for a latch
- `statetable`—for general sequential behavior

Both the `ff` and `latch` groups create two state-variables that represent noninverted output and inverted output. You can use these state variables in the `function` attribute when you define an output pin's function.

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

Example 2-1 uses an `internal_node` attribute statement for the function of the output pins.

Example 2-1 D Flip-Flop With statetable Group

```
cell(flipflop1) {
  area : 7 ;
  pin(D) {
    direction : input ;
    capacitance : 1.3 ;
    timing() {
      timing_type : setup_rising ;
      intrinsic_rise : 0.9 ;
      intrinsic_fall : 0.9 ;
      related_pin : "CP" ;
    }
    timing() {
      timing_type : hold_rising ;
      intrinsic_rise : 0.5 ;
      intrinsic_fall : 0.5 ;
      related_pin : "CP" ;
    }
  }
  pin(CP) {
    direction : input ;
    capacitance : 1.3 ;
    min_pulse_width_high : 1.5 ;
    min_pulse_width_low  : 1.5 ;
  }
  statetable (" D    CP", "IQ   IQN") {
    table :    " L/H  R  : - - : L/H  H/L,\
                  -   ~R : - - : N    N" ;
  }

pin(Q) {
    direction : output ;
    internal_node : "IQ" ;
    timing() {
      timing_type : rising_edge ;
      intrinsic_rise : 1.11 ;
      intrinsic_fall : 1.43 ;
      rise_resistance : 0.1513 ;
      fall_resistance : 0.0544 ;
      related_pin : "CP" ;
    }
  }
  pin(QN) {
    direction : output ;
    internal_node : "IQN"
    timing() {
      timing_type : rising_edge ;
      intrinsic_rise : 1.58 ;
      intrinsic_fall : 1.56 ;
      rise_resistance : 0.1513 ;
      fall_resistance : 0.0544 ;
      related_pin : "CP" ;
    }
  }
}
```

In the cell in Example 2-2, all the definitions are the same as in Example 2-1, except that

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

The D flip-flop defines two variables, IQ and IQN. The next_state equation determines the value of IQ after the next clocked_on transition. In Example 2-2, IQ is assigned the value of the D input.

Example 2-2 D Flip-Flop With ff Group

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

Describing Timing

Timing is divided into two major areas: describing delay (the actual circuit timing) and specifying constraints (boundaries). The “Timing Arcs” chapter in the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide* describes timing arcs, which are, with netlist interconnect information, the paths followed by the path tracer during path analysis. Timing arcs can be delay arcs or constraint arcs. Each timing arc has a startpoint and an endpoint. The startpoint can be an input, output, or inout pin. The endpoint is always an output pin or an inout pin. The only exception is a constraint timing arc, such as a setup or hold constraint between two input pins.

You describe timing delay and constraints in the `timing` group in a `bundle`, `bus`, or `pin` group within a cell. Timing groups contain the information that Design Compiler needs to model timing arcs and trace paths. The `timing` groups define the timing arcs through a cell and the relationships between clock and data input signals.

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

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

For naming details, see the section on using the `timing` group in the “Timing Arcs” chapter of the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide*.

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

related_pin

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

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

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

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

intrinsic delay

The fixed delay from input to output pins.

transition delay

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

slope sensitivity

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

The delay model you use determines which set of attributes for delay calculation you specify in a pin's `timing` group. For example, Table 2-1 shows which attributes you can specify to describe the following types of delay in the CMOS linear delay model.

Table 2-1 Delay Types and Valid Attributes

Delay type	**Valid attributes**
Intrinsic	`intrinsic_rise`, `intrinsic_fall`
Transition	`rise_resistance`, `fall_resistance`
Slope sensitivity	`slope_rise`, `slope_fall`

You can set various timing constraints, such as these:

setup and hold arcs

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

recovery and removal arcs

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

skew

This is another constraint that the VHDL library generator uses for simulation.

You can also set state-dependent and conditional constraints.

See the "Timing Arcs" chapter in the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide* for information about the `timing` group attributes for setting constraints and defining delay with the different delay models. The "Timing and Power Report Format" chapter in the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide* describes how to get information about timing in cells by using the Library Compiler `report_lib -timing` command.

Defining I/O Pads

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

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

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

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

The example libraries do not contain examples of I/O pads, which are often defined in a separate library.

These are the steps in defining I/O pads:

1. Identify pad cells
2. Describe multicell pads
3. Define units for pad cells
4. Identify area resources
5. Describe input pads
6. Describe output pads
7. Model wire load

Identifying Pad Cells

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

These attributes identify pads:

pad_cell and auxiliary_pad_cell

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

pad_type : clock

Attribute that identifies a clock driver pad cell.

is_pad

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

direction

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

Describing Multicell Pads

Pads can have multiple cells—a pad cell and a driver cell, for example. You can describe I/O pads in one of two ways:

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

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

These are attributes that describe multiple cells:

auxiliary_pad_cell

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

multicell_pad_pin

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

connection_class

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

Defining Units for Pad Cells

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

capacitive_load_unit

Defines the capacitance associated with a standard load.

pulling_resistance_unit

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

voltage_unit

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

current_unit

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

Identifying Area Resources

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

area

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

Describing Input Pads

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

`input_voltage` group

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

`hysteresis` attribute

Identifies whether a pad has hysteresis. Pads with hysteresis sometimes have derating factors that are different from the factors for the cells in the core. You describe the timing of cells with hysteresis with a `scaled_cell` group, as described in "hysteresis Attribute" on page 7-53.

Note:
You can also use the `input_voltage` group to specify voltages ranges for standard cells. For more information, see the *Library Compiler User Guide: Modeling Power and Timing Technology Libraries*.

Describing Output Pads

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

output_voltage group

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

drive_current attribute

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

slew_control attribute

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

Note:
You can also use the `output_voltage` group to specify voltages ranges for standard cells. For more information, see the *Library Compiler User Guide: Modeling Power and Timing Technology Libraries*.

Modeling Wire Load

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

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

See “Modeling the Wire Load” on page 2-10 for an example of the `wire_load` group.

Task 5: Describing Application-Specific Data

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

This chapter does not describe creating VHDL simulation libraries, but you can use the Library Compiler `write_lib -f vhdl` command to create a VHDL simulation library that matches your technology library. From an existing technology library, the Library Compiler VHDL library generator can create VITAL (VHDL initiative toward ASIC libraries) simulation model VHDL simulation libraries.

To understand the structure and contents of VHDL simulation libraries, see the *Library Compiler VHDL Libraries Reference Manual*.

References

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

- Chapter 6, “Building a Technology Library”
- Chapter 7, “Defining Core Cells”
- Chapter 10, “Defining Test Cells”
- The “Modeling Power and Electromigration” and “Building Environments” chapters in the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide*.

Procedure for Describing Application-Specific Data

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

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

Setting Design Rules

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

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

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

To specify load values, use the following attributes:

fanout_load

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

max_fanout

Specifies the maximum number of loads a pin can drive.

max_transition

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

max_capacitance

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

min_fanout

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

min_capacitance

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

See “Describing Design Rule Checks” on page 7-37 for more information about the design rule attributes.

Modeling for Testable Circuits

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

To use DFT Compiler, add test-specific details of scannable cells to your technology libraries. Identify scannable flip-flops and latches and select the types of cells that are not scannable which they replace for a given scan methodology. See Chapter 10, “Defining Test Cells,” for information about scan cells.

Modeling for Power

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

Leakage Power

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

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

```
library(power_example) {
   leakage_power_unit : 1nW ;
   default_cell_leakage_power : 0.1 ;
   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" ;
         values : (2.0) ;
      }
   }
}
```

Short-Circuit Power

Short-circuit or internal power is the power dissipated whenever a pin makes a transition. Library developers can choose one of the following options:

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

Switching Power

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

See Chapter 6, "Building a Technology Library," and the "Building Environments" and "Modeling Power and Electromigration" chapters in the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide* for information about the attributes used in modeling static and dynamic power.

Adding Synthesis-to-Layout Features

In-place optimization, wire load model creation, and routability are synthesis-to-layout features you can add to your design. See "Modeling Wire Load" on page 2-26 for information about creating wire load models.

In-Place Optimization

You can use in-place optimization, an incremental optimization step, to improve performance or to fix design rule violations introduced by layout.

In-place optimization occurs after placement and routing, when Design Compiler optimizes a critical path by replacing individual cells with similar ones that improve the speed or power of the circuit.

For in-place optimization, use the following attributes:

in_place_swap_mode

Library-level processing attribute that specifies whether in-place optimization is allowed with this library. It describes the criteria that Design Compiler uses in cell swapping during in-place optimization.

cell_footprint

Cell-level attribute that specifies which cells can be swapped.

Routability

Factors that affect a design's routability are the number of feedthroughs available, the maximum fanout, and the high number of inputs to cells. You can put attributes in the library that provide Design Compiler with feed-through information. These are the attributes that affect routability:

default_min_porosity

A `library` group attribute that specifies the constraint value for Design Compiler when there is no porosity specification at the dc_shell command line. Porosity is the total feed-through area available, divided by the cell area.

routing_layers

A `library` group attribute that declares the routing layers available for place and route for the library. This attribute 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 library can have only one `routing_layers` complex attribute.

Technology Library Source File Example

This is the source file for a technology library using a CMOS linear delay model. This technology library contains examples of core cells only.

```
library(example) {
technology (cmos) ;                     /* technology */
delay_model : generic_cmos;             /* delay model */
default_inout_pin_cap    :       1.0;   /* default attributes */
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"; /* library units */
pulling_resistance_unit      : "100ohm";
voltage_unit                 : "1V";
current_unit                 : "1mA";
capacitive_load_unit(1,pf);
nom_process                  :  1.0;
nom_temperature              : 25.0;
nom_voltage                  : 5.0;
operating_conditions(WCCOM) {
/* operating condition models */
   process : 1.5 ;
   temperature : 70 ;
   voltage : 4.75 ;
   tree_type : "worst_case_tree" ;
}
operating_conditions(WCIND) {
   process : 1.5 ;
   temperature : 85 ;
   voltage : 4.75 ;
   tree_type : "worst_case_tree" ;
}
operating_conditions(WCMIL) {
   process : 1.5 ;
   temperature : 125 ;
   voltage : 4.5 ;
   tree_type : "worst_case_tree" ;
}
operating_conditions(BCCOM) {
   process : 0.6 ;
   temperature : 0 ;
   voltage : 5.25 ;
```

```
      tree_type : "best_case_tree" ;
   }
   operating_conditions(BCIND) {
      process : 0.6 ;
      temperature : -40 ;
      voltage : 5.25 ;
      tree_type : "best_case_tree" ;
   }
   operating_conditions(BCMIL) {
      process : 0.6 ;
      temperature : -55 ;
      voltage : 5.5 ;
      tree_type : "best_case_tree" ;
   }
   wire_load("05x05") {                /* wire load models */
      resistance : 0 ;
      capacitance : 1 ;
      area : 0 ;
      slope : 0.186 ;
      fanout_length(1,0.39) ;
   }
   wire_load("10x10") {
      resistance : 0 ;
      capacitance : 1 ;
      area : 0 ;
      slope : 0.311 ;
      fanout_length(1,0.53) ;
   }

   wire_load("20x20") {
      resistance : 0 ;
      capacitance : 1 ;
      area : 0 ;
      slope : 0.547 ;
      fanout_length(1,0.86) ;
   }
   wire_load("30x30") {
      resistance : 0 ;
      capacitance : 1 ;
      area : 0 ;
      slope : 0.782 ;
      fanout_length(1,1.40) ;
   }
   wire_load("40x40") {
      resistance : 0 ;
      capacitance : 1 ;
      area : 0 ;
      slope : 1.007 ;
      fanout_length(1,1.90) ;
   }
   wire_load("50x50") {
      resistance : 0 ;
      capacitance : 1 ;
      area : 0 ;
      slope : 1.218 ;
      fanout_length(1,1.80) ;
   }
   wire_load("60x60") {
      resistance : 0 ;
      capacitance : 1 ;
      area : 0 ;
      slope : 1.391 ;
      fanout_length(1,1.70) ;
   }
   wire_load("70x70") {
      resistance : 0 ;
      capacitance : 1 ;
      area : 0 ;
      slope : 1.517 ;
      fanout_length(1,1.80) ;
   }
   wire_load("80x80") {
      resistance : 0 ;
      capacitance : 1 ;
      area : 0 ;
      slope : 1.590 ;
      fanout_length(1,1.80) ;
   }
   wire_load("90x90") {
      resistance : 0 ;
      capacitance : 1 ;
      area : 0 ;
      slope : 1.64 ;
      fanout_length(1,1.9) ;
   }
   cell(AN2) {                     /* 2-input AND gate */
      area : 2 ;
      pin(A,B) {
         direction : input ;
         capacitance : 1 ;
      }
      pin(Z) {
         direction : output ;
         function : "A * B" ;
         timing() {
            intrinsic_rise : 0.39 ;
            intrinsic_fall : 0.53 ;
            rise_resistance : 0.122 ;
            fall_resistance : 0.038 ;
            related_pin : "A B" ;
         }
      }
   }
   cell(OR2) {                      /* 2-input OR gate */
      area : 2 ;
      pin(A B) {
         direction : input ;
         capacitance : 1 ;
      }
      pin(Z) {
         direction : output ;
         function : "A + B" ;
         timing() {
            intrinsic_rise : 0.38 ;
            intrinsic_fall : 0.85 ;
            rise_resistance : 0.1443 ;
            fall_resistance : 0.0589 ;
            related_pin : "A B" ;
         }
      }
   }
   cell(TRI_INV2) {              /* three-state INVERTER */
      area : 3 ;
      pin(A) {
         direction : input ;
         capacitance : 2 ;
      }
      pin(E) {
         direction : input ;
         capacitance : 2 ;
      }
      pin(Z) {
         direction : output ;
         function : "A'" ;
         three_state : "E'" ;
         timing() {
            intrinsic_rise : 0.39 ;
            intrinsic_fall : 0.75 ;
            rise_resistance : 0.15 ;
            fall_resistance : 0.06 ;
            related_pin : "A" ;
         }
         timing() {
            timing_type : three_state_enable;
            intrinsic_rise : 0.39 ;
            intrinsic_fall : 0.75 ;
            rise_resistance : 0.15 ;
            fall_resistance : 0.06 ;
            related_pin : "E" ;
         }
         timing() {
            timing_type : three_state_disable ;
            intrinsic_rise : 0.25 ;
            intrinsic_fall : 0.35 ;
            rise_resistance : 0.15 ;
            fall_resistance : 0.06 ;
            related_pin : "E" ;
         }
      }
```

```
}
cell(FF1) {                            /* D FLIP-FLOP */
   area : 7 ;
   pin(D) {
      direction : input ;
      capacitance : 1.3 ;
      timing() {
         timing_type : setup_rising ;
         intrinsic_rise : 0.9 ;
         intrinsic_fall : 0.9 ;
         related_pin : "CP" ;
      }
      timing() {
         timing_type : hold_rising ;
         intrinsic_rise : 0.5 ;
         intrinsic_fall : 0.5 ;
         related_pin : "CP" ;
      }
   }
   pin(CP) {
      direction : input ;
      capacitance : 1.3 ;
      min_pulse_width_high : 1.5 ;
      min_pulse_width_low  : 1.5 ;
   }
   ff ("IQ", "IQN") {
      next_state : "D" ;
       clocked_on : "CP" ;
   }
   pin(Q) {
      direction : output ;
      function : "IQ" ;
      timing() {
         timing_type : rising_edge ;
         intrinsic_rise : 1.11 ;
         intrinsic_fall : 1.43 ;
         rise_resistance : 0.1513 ;
         fall_resistance : 0.0544 ;
         related_pin : "CP" ;
      }
   }
   pin(QN) {
      direction : output ;
      function : "IQN"
      timing() {
         timing_type : rising_edge ;
         intrinsic_rise : 1.58 ;
         intrinsic_fall : 1.56 ;
         rise_resistance : 0.1513 ;
         fall_resistance : 0.0544 ;
         related_pin : "CP" ;
      }
   }
```

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

Technology Library With Delay Example

This is the source file for a technology library that uses the nonlinear delay model.

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

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

k_process_pin_cap            :  0.0;  /* delay calculation
scaling factors */
k_process_wire_cap           :  0.0;
k_process_wire_res           :  1.0;
k_temp_pin_cap               :  0.0;
k_temp_wire_cap              :  0.0;
k_temp_wire_res              :  0.0;
k_volt_pin_cap               :  0.0;
k_volt_wire_cap              :  0.0;
k_volt_wire_res              :  0.0;

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

operating_conditions(WCCOM) {
/* operating condition models */
   process : 1.5 ;
   temperature : 70 ;
   voltage : 4.75 ;
   tree_type : "worst_case_tree" ;
}
operating_conditions(WCIND) {
   process : 1.5 ;
   temperature : 85 ;
   voltage : 4.75 ;
   tree_type : "worst_case_tree" ;
}
operating_conditions(WCMIL) {
   process : 1.5 ;
   temperature : 125 ;
```

```
      voltage : 4.5 ;
      tree_type : "worst_case_tree" ;
   }
   operating_conditions(BCCOM) {
      process : 0.6 ;
      temperature : 0 ;
      voltage : 5.25 ;
      tree_type : "best_case_tree" ;
   }
   operating_conditions(BCIND) {
      process : 0.6 ;
      temperature : -40 ;
      voltage : 5.25 ;
      tree_type : "best_case_tree" ;
   }
   operating_conditions(BCMIL) {
      process : 0.6 ;
      temperature : -55 ;
      voltage : 5.5 ;
      tree_type : "best_case_tree" ;
   }
   wire_load("05x05") {        /* wire load models */
      resistance : 0 ;
      capacitance : 1 ;
      area : 0 ;
      slope : 0.186 ;
      fanout_length(1,0.39) ;
   }
   wire_load("10x10") {
      resistance : 0 ;
      capacitance : 1 ;
      area : 0 ;
      slope : 0.311 ;
      fanout_length(1,0.53) ;
   }
   wire_load("20x20") {
      resistance : 0 ;
      capacitance : 1 ;
      area : 0 ;
      slope : 0.547 ;
      fanout_length(1,0.86) ;
   }
   wire_load("30x30") {
      resistance : 0 ;
      capacitance : 1 ;
      area : 0 ;
      slope : 0.782 ;
      fanout_length(1,1.40) ;
   }
   wire_load("40x40") {
      resistance : 0 ;
      capacitance : 1 ;
      area : 0 ;
      slope : 1.007 ;
      fanout_length(1,1.90) ;
   }
   wire_load("50x50") {
      resistance : 0 ;
      capacitance : 1 ;
      area : 0 ;
      slope : 1.218 ;
      fanout_length(1,1.80) ;
   }
   wire_load("60x60") {
      resistance : 0 ;
      capacitance : 1 ;
      area : 0 ;
      slope : 1.391 ;
      fanout_length(1,1.70) ;
   }
   wire_load("70x70") {
      resistance : 0 ;
      capacitance : 1 ;
      area : 0 ;
      slope : 1.517 ;
      fanout_length(1,1.80) ;
   }
   wire_load("80x80") {
      resistance : 0 ;
      capacitance : 1 ;
      area : 0 ;
      slope : 1.590 ;
      fanout_length(1,1.80) ;
   }
   wire_load("90x90") {
      resistance : 0 ;
      capacitance : 1 ;
      area : 0 ;
      slope : 1.64 ;
      fanout_length(1,1.9) ;
   }
   /* Define one dimensional lookup table of size 4 */
   lu_table_template(trans_template) {
      variable_1 : input_net_transition;
      index_1 ("0.0, 0.5, 1.5, 2.0");
   }
   /* Define template of size 3x3 */
   lu_table_template(constraint_template) {
      variable_1 : constrained_pin_transition;
      variable_2 : related_pin_transition;
      index_1 ("0.0, 0.5, 1.5");
      index_2 ("0.0, 2.0, 4.0");
   }
   cell(AN2) {                         /* 2-input AND gate */
      area : 2 ;
      pin(A,B) {
         direction : input ;
         capacitance : 1 ;
      }
      pin(Z) {
         direction : output ;
         function : "A * B" ;
         timing() {
            rise_propagation(scalar) {
               values ("0.12");
            }
            fall_propagation(scalar) {
               values ("0.12");
            }
            rise_transition(trans_template) {
               values ("0.1, 0.15, 0.20, 0.29");
            }
            fall_transition (trans_template) {
               values ("0.1, 0.15, 0.20, 0.29");
            }
            related_pin : "A B" ;
         }
      }
   }
   cell(OR2) {                         /* 2-input OR gate */
      area : 2 ;
      pin(A B) {
          direction : input ;
          capacitance : 1 ;
      }
      pin(Z) {
         direction : output ;
         function : "A + B" ;
         timing() {
            rise_propagation(scalar) {
               values ("0.12");
            }
            fall_propagation(scalar) {
               values ("0.12");
            }
            rise_transition(trans_template) {
               values ("0.1, 0.15, 0.20, 0.29");
            }
            fall_transition (trans_template) {
               values ("0.1, 0.15, 0.20, 0.29");
            }
            related_pin : "A B" ;
         }
      }
   }
   cell(TRI_INV2) {                    /* three-state INVERTER */
      area : 3 ;
      pin(A) {
         direction : input ;
         capacitance : 2 ;
      }
      pin(E) {
         direction : input ;
         capacitance : 2 ;
      }
      pin(Z) {
         direction : output ;
         function : "A’" ;
         three_state : "E’" ;
         timing() {
            rise_propagation(scalar) {
               values ("0.12");
            }
            fall_propagation(scalar) {
               values ("0.12");
            }
            rise_transition(trans_template) {
               values ("0.1, 0.15, 0.20, 0.29");
            }
            fall_transition (trans_template) {
               values ("0.1, 0.15, 0.20, 0.29");
            }
            related_pin : "A" ;
         }
         timing() {
            timing_type : three_state_enable;
            rise_propagation(scalar) {
               values ("0.12");
            }
            fall_propagation(scalar) {
               values ("0.12");
            }
            rise_transition(trans_template) {
               values ("0.1, 0.15, 0.20, 0.29");
            }
            fall_transition (trans_template) {
               values ("0.1, 0.15, 0.20, 0.29");
            }
            related_pin : "E" ;
         }
         timing() {
            timing_type : three_state_disable ;
            rise_propagation(scalar) {
               values ("0.12");
            }
            fall_propagation(scalar) {
               values ("0.12");
            }
            rise_transition(trans_template) {
               values ("0.1, 0.15, 0.20, 0.29");
            }
            fall_transition (trans_template) {
               values ("0.1, 0.15, 0.20, 0.29");
            }
            related_pin : "E" ;
         }
      }
   }
   cell(FF1) {                           /* D FLIP-FLOP */
      area : 7 ;
      pin(D) {
         direction : input ;
         capacitance : 1.3 ;
         timing() {
            timing_type : setup_rising ;
            related_pin : "CP" ;
            rise_constraint(constraint_template) {
               values ("0.0, 0.13, 0.19", \
                           "0.21, 0.23, 0.41", \
                           "0.33, 0.37, 0.50");
            }
            fall_constraint(constraint_template) {
               values ("0.0, 0.14, 0.20", \
                           "0.22, 0.24, 0.42", \
                           "0.34, 0.38, 0.51");
            }
         }
         timing() {
            timing_type : hold_rising ;
            related_pin : "CP" ;
            rise_constraint(constraint_template) {
               values ("0.0, 0.13, 0.19", \
                           "0.21, 0.23, 0.41", \
                           "0.33, 0.37, 0.50");
            }
            fall_constraint(constraint_template) {
               values ("0.0, 0.14, 0.20", \
                           "0.22, 0.24, 0.42", \
                           "0.34, 0.38, 0.51");
            }
         }
      }

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

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

Task 6: Compiling the Library

This section describes how to compile a library.

References

For further information about compiling a library, see Chapter 3, "Using the Library Compiler Shell Interface."

Procedure

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

Using the shell interface,

1. Read the library source file into Library Compiler.

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

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

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

Task 7: Managing the Library

This section describes how to manage a library.

References

For more information about managing a library, see Chapter 3, "Using the Library Compiler Shell Interface."

Procedure

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

read

   Loads a previously saved .db library.

update_lib

Adds new groups (cells, operating conditions, timing-range, scaling factors, bus type definitions, interconnect wire load) to the existing library.

report_lib

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

compare_lib

Compares a symbol library against the corresponding technology library for consistency.

list

Displays information (name in memory, variable values, licenses) for resident libraries.

remove_lib

Removes a library from memory.

encrypt_lib

Secures a VHDL source library file.

add_module

Adds only memory modules to an existing library.

3

Using the Library Compiler Shell Interface

You can access Library Compiler from either the shell (command-line) interface or from the graphical interface. The shell interface lets you type commands, arguments, and options at a system prompt. The graphical interface lets you use your cursor and mouse buttons, as well as your keyboard, to specify commands, options, and file names.

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

- Reporting Library Information
- Listing Resident Libraries
- Comparing Libraries
- Removing Libraries From Memory
- Encrypting VHDL Source Libraries
- Using Shell Commands
- Using Variables

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

About the Shell Interface

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

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

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

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

- Start the Library Compiler command-line interface (lc_shell)
- Use a simple sequence of lc_shell commands
- Interrupt an lc_shell command
- Exit the command-line interface

Starting the Command Interface

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

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

```
% lc_shell
```

Note:
If you cannot start lc_shell, see the*Installation Guide* for your hardware platform and install your system properly before continuing.

The `lc_shell` invocation command has several options:

-f

Executes a script file and starts the lc_shell interface:

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

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

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

-no_init

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

-no_init -f command_log.file

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

-x *command_string*

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

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

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

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

Understanding the Command Log

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

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

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

Set the `command_log_file` variable in a setup file, because the variable value does not take effect if you change it interactively during a Library Compiler session. See the next section, “Using Setup Files,” for more information.

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

Using Setup Files

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

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

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

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

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

Using lc_shell Commands to Compile a Source File

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

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

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

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

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

3. Write the library to a system file.

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

4. Request the library reports.

```
lc_shell> report_lib my_lib
```

5. Exit the Library Compiler program.

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

Support for gzip Format

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

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

Note:
Library Compiler can only read gzip compressed files. No other compression file format is supported.

Interrupting Commands

If you enter the wrong options for a command or enter the wrong command, you can interrupt command processing. Type the interrupt sequence, Control-c.

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

If Library Compiler is processing an include script file and you interrupt one of its commands, the script processing itself is interrupted. No further script commands are processed, and control returns to lc_shell.

If you enter Control-c three times before a command responds to your interruption, lc_shell itself is interrupted and exits, with the following message:

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

Leaving the Command Interface

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

```
lc_shell> exit

Thank you...
<prompt>
```

or

```
lc_shell> quit

Thank you...
<prompt>
```

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

```
lc_shell> exit
[exit_code_value]
Thank you...
<prompt>
```

or

```
lc_shell> quit
[exit_code_value]
Thank you...
<prompt>
```

On UNIX systems, you can also exit lc_shell by typing the end-of-file (EOF) sequence, Control-d:

```
lc_shell> ^d
Thank you...
<prompt>
```

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

Describing Library Files and Memory Files

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

```
path/libraries/
my_library.lib
```

When you read a library into Library Compiler,

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

  ```
  pathname/
  filename:library
  ```

pathname

The directory from which the source file was read

filename

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

library

The user-specified name of the library

The default disk file name for a library is the library name with the suffix .db for a technology library or .sdb for a symbol library. For example, for the UNIX operating system, the memory file name for a technology library named my_lib is

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

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

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

The `list -libraries` command lists all libraries in memory and their memory file names. In a UNIX environment, the list appears as follows:

```
lc_shell> list -libraries

Library   File       Path
-------   ----       ----
A         A.db       /user1/libraries
B         B.db       /home/libraries
psyn1     psyn1.pdb  /home/libraries
```

Using search_path and Path Names

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

Table 3-1 shows the UNIX OS path specifications:

Table 3-1 Path Types for UNIX OS

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

The `search_path` variable is described in the online help (help `search_path`).

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

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

lc_shell Command Syntax

An lc_shell command has the following components:

command name

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

options

Option names are prefixed by a hyphen (-). A command can have zero or more options. For example, the `read` command has no options, but the `list` command has many options.

You can abbreviate (truncate) option names, if the abbreviation is unique to the related command. For example, the `list` command has both `-libraries` and `-licenses` options. The minimum abbreviations for these options are `-lib` and `-lic`.

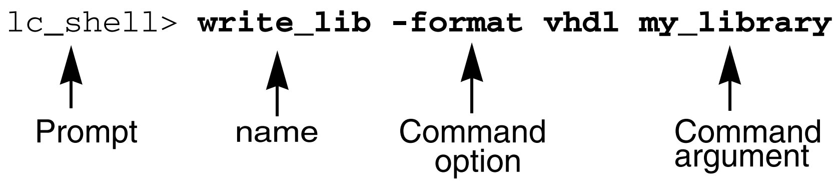
arguments

Command-specific values. Both commands and options can have arguments. For example, the `read` command has no options but has one argument (the file name). The `write_lib` command has two options: `-format` and `-output`. The `-format` and `-output` options each require an argument (namely, the format and the file name).

Note:
In the Library Compiler environment, command names are case-sensitive. For example, the following two commands are not equivalent: `compare_lib lib LIB1` and `compare_lib lib lib1`.

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

Figure 3-1 Typical Command



After processing each command, lc_shell returns a status value that is the value of the command. This status value is also set in the lc_shell variable `lc_shell_status` (see “Reexecuting Commands” on page 3-35).

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

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

See the *Library Compiler Technology and Symbol Libraries Reference Manual* for the syntax and description of all the Library Compiler commands.

String and List Operator

The lc_shell program provides a string (list) operator you can use with command arguments. Table 3-2 shows how this operator works.

Table 3-2 String and List Concatenation Operator

Operator	Example	Result
+	"string" + "string"	"string string"
+	{a b c} + "d"	{a b c d}
+	{a b c} + {d e}	{a b c d e}

String and list operators can also be used with variables:

```
lc_shell> my_lib = "my_library.db"
Warning:  Defining new variable ’my_lib’ my_library.db

lc_shell> "read path/libraries/" + my_lib
Loading db file path/libraries/my_library.db’


lc_shell> list search_path
search_path = { /home, /libraries/files }
```

In a UNIX environment, enter

```
lc_shell> search_path = /usr/libraries + search_path
{ /usr/libraries, /home, /libraries/files }
```

Redirecting Command Output

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

```
command > file
```

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

For example, to redirect a report to a file,

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

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

```
command >> file
```

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

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

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

Using Commands

You can use lc_shell commands in the following two ways:

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

Getting Help Information

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

Command Use Help

The intended use (syntax) of an lc_shell command is displayed when you use the `-help` option with a command name. For example, the `read_lib` command's use help is

```
lc_shell> read_lib -help
Usage:  read_lib
      -format(EDIF symbol format; default is Synopsys
format)
      -symbol(with EDIF, name of Synopsys library file to
create)
      <file_name>(technology or symbol library file)
      -no_warnings(disable warning messages)
      -names_file <file_list>  (one or more names files)
```

The command use help displays the command's possible options and arguments.

Topic Help

The `help` command displays information about an lc_shell command (including the `help` command itself) and a variable or variable group.

Syntax

```
help [topic]
```

topic

Names a command, variable, or variable group. If you do not name a topic, the `help` command displays its own man page.

The `help` command lets you display the man pages interactively while you are running lc_shell. Man pages for all commands, variables, and predefined variable groups and user messages (for example, DBVH-1) are included in the online help pages.

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

```
lc_shell> help
```

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

```
lc_shell> help report_lib
```

The following example returns a man page for a specified variable:

```
lc_shell> help system_variables
```

If you request help for a topic that cannot be found, Library Compiler displays the following error message:

```
lc_shell> help xxyxx_topic
Error: No manual entry for 'xxyxx_topic'
```

Reading In Libraries

The `read_lib` command loads a technology (.lib), physical (.plib), pseudo_physical_library (.pplib), or symbol library (.slib) source file into the Library Compiler program and compiles it to a Synopsys database format (.db, .pdb, or .sdb). When you start to load a .lib file, Library Compiler attempts to check out a Library-Compiler license. If the license is missing, the library is still loaded, but all the functional information is removed; that is, all cells are black boxes and optimization is disabled. If the license is present, Library Compiler releases the license when the `read_lib` command finishes loading the .lib file.

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

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

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

For example, to read the cmos.lib technology library into Library Compiler, type the following command:

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

The following example reads an EDIF symbol library, cmos.edif, into Library Compiler and creates a symbol library file named cmos.slib in Synopsys format.

```
lc_shell> read_lib -format edif -symbol cmos.slib cmos.edif
```

The following example reads a physical library file (mylib.plib). The pseudo-physical library file contains technology information in Synopsys technology entry format.

```
dc_shell> read_lib -format gdsii -plibrary mylib -pplibrary bar.plib
{mylib.plib}
```

The following example reads a list of GDSII files (mylib1 and mylib2) and the technology file (bar.plib), and then generates an intermediate pseudo-physical library file (mylib.plib). The pseudo-physical library file contains technology information in Synopsys technology entry format.

```
dc_shell> read_lib -format gdsii -plibrary mylib -pplibrary bar.lib
{mylib1.gdsii mylib2.gdsii}
```

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

Writing Library Files

The `write_lib` command saves a library memory file to a disk file in either Synopsys internal database (.db), physical database (.pdb), or VHDL format. You can also generate a Verilog model or a datasheet from a .lib file.

Syntax

```
write_lib library_name [-format format_name] \
                        [-names_file name_mapping_files] \
                        [-output file] [-macro_only] \
                        [-compress compression_method]
```

`library_name`

Specifies the file that you want to write to disk. To write a file in .db or .pdb format, use the name of the technology, physical, pseudo physical, or symbol library for *library_name*.

By default, the names of the output files that are saved to disk are the same as the library names. However, the suffix changes depending on the file type. The technology library file has a .db suffix, the physical library has a .pdb suffix, and the symbol library has a .sdb suffix.

`-format` *`format_name`*

Defines the Library Compiler output format. If you do not use the `-format` option, Library Compiler generates a file in Synopsys database format. The following formats are valid values:

- `verilog`

  Library Compiler generates a Verilog model from a .lib file and creates a *cell*.v Verilog file for each cell in the library. You must first run Liberty NCX and generate a .lib file. Next, you generate a Verilog file from the .lib file using Library Compiler. This ensures that the Verilog file matches the .lib file exactly. Although you generate the Verilog file in Library Compiler, it requires a Liberty NCX license and will fail if a Liberty NCX license is not available. Use the following commands to generate Verilog models:

  1. To enable Verilog model generation, set the `verilog_enable` variable to true. By default, the variable is set to false, and a Verilog file is not generated.

     ```
     set verilog_enable [true | false]
     ```

  2. To create a Verilog file from a .lib file, run the `write_lib` command in lc_shell and specify the .lib library name, as shown:

     ```
     write_lib -format verilog library_name
     ```

  3. If you want the Verilog model to be written to a directory other than *library_name*_verilog, set the `veriloglib_output_dir` variable, as shown:

     ```
     set veriloglib_output_dir directory_name
     ```

  For more information about generating Verilog models, see the Appendix B, “Library Conversion Support.”

- `datasheet`

  Library Compiler can generate datasheets in text (.txt) format or in HTML (.html) format. You must first run Liberty NCX and generate a .lib file. Next, you generate a datasheet from the .lib file using Library Compiler. This ensures that the datasheet matches the .lib file exactly. Although datasheet generation is performed in Library Compiler, it requires a Liberty NCX license and will fail if a Liberty NCX license is not available. Use the following commands to generate datasheets:

1. To enable datasheet generation, set the `datasheet_enable` variable to `true`. By default, the variable is set to `false`, and a datasheet is not generated.

   ```
   set datasheet_enable [true | false]
   ```

2. To create a datasheet from a .lib file, run the `write_lib` command in lc_shell and specify the .lib library name as shown:

   ```
   write_lib -format datasheet library_name
   ```

3. If you want the datasheet to be written to a directory other than *library_name*_datasheet, set the `datasheet_output_dir` variable, as shown:

   ```
   set datasheet_output_dir directory_name
   ```

For more information about generating datasheets, see the the Appendix B, "Library Conversion Support."

- `vhdl`

  To write a file in VHDL format, use the name of the technology library used to generate the VHDL library for *library_name*. To generate a VHDL-formatted library, use `vhdl` for *`format_name`*. When you select VHDL format, Library Compiler uses the compiled technology library specified by *library_name* to generate and write a VHDL library to disk. The VHDL library consists of the following files:

  - *_VITAL.vhd
  - *_Vtables.vhd
  - *_components.vhd
  - *_Vcomponents.vhd

`-names_file` *`name_mapping_files`*

Changes the names of symbols and symbol pins. The *name_mapping_files* value follows the same format as the name-mapping files in the `change_names` command. See `change_names` in the Synopsys man pages for details.

`-output` *`file`*

Specifies an output file name that is different from *library_name*. If you use the `-output` option, Library Compiler writes a disk file with the file name exactly as defined.

`-macro_only`

Applies only to physical libraries. Specifies to write out all macros and their contents. Other information from the original .plib file (such as, the resource and design rule information) is not written out in the .pdb file.

`-compress`

Writes a compressed library file to a disk file in either Synopsys internal database (.db) or physical database (.pdb) format. The only valid compression method is gzip.

Examples

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

```
lc_shell> write_lib cmos
```

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

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

To write a compressed library called cmos to a file named my_cmos.db.gz in the current directory, enter

```
lc_shell> write_lib cmos -compress gzip my_cmos.db.gz
```

To write a physical library to a file named psym1.pdb in the current directory, enter

```
lc_shell> write_lib psym1 -o psym1.pdb
```

To write compiled VHDL libraries to files whose prefix is cmos and that reside in the current working directory, enter

```
lc_shell> write_lib -f vhdl cmos
```

To write compiled VHDL libraries to files whose prefix is cmos and that reside in the /libraries/work directory, enter

```
lc_shell> write_lib -f vhdl cmos -o path/libraries/work
```

Note:
To create a VHDL library, read the source file into the Library Compiler program in the same session that you generate the library.

You can specify many modeling features before you generate the VHDL libraries. See the *Library Compiler VHDL Libraries Reference Manual* for more information.

Model Adaptation System

You can convert existing libraries from one format to another by using the Model Adaptation System, which converts or merges one or more existing library files in Liberty (.lib) format to create a new library that is written out in Liberty format. For more information, see Appendix B, “Library Conversion Support.”

Note:
Although library conversion by using the Model Adaptation System is performed in Library Compiler, it requires a Liberty NCX license and will fail if it is not available.

Generating Hierarchical .db Designs

Library Compiler generates hierarchical .db designs with the `model` group, which can accept all the groups and attributes that a `cell` group accepts, plus these additional two attributes:

- `cell_name` simple attribute
- `short` complex attribute

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

Syntax

```
model(model_name)
{
       area : float;
       ...
       cell_name :
cellnameid ; /*
optional */
       ...

pin(name1string) {
...}

pin(name2string) {
...}
       short
(name1,name2
);
}
```

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

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

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

Verifying Hierarchical .db Designs

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

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

The following example commands produce the .db files described:

Commands

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

lc_shell> write_lib mylib
```

Files Produced

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


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

Loading Compiled Libraries

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

Syntax

```
read file_list
```

file_list

A list of one or more file names containing libraries.

For example, to read in a compiled technology library file named lib.db, enter

```
lc_shell> read lib.db

Loading db file 'path/libraries/ex/lib.db'
{}
lc_shell> list -libraries

Library  File    Path
------     ----     ----
Lib_1    lib.db  path/libraries/ex
```

Note that the lib.db file contains a technology library called Lib_1.

To read in a compiled physical library file named lib.pdb, enter

```
lc_shell> read lib.pdb

Loading pdb file 'path/
libraries/ex/
lib.pdb'
{}
lc_shell> list -libraries

Library  File    Path
------     ----     ----
Lib_2    lib.pdb  path/libraries/ex
```

Note that the lib.pdb file contains a physical library called Lib_2.

To load multiple libraries, enter

```
lc_shell> read {my_lib1.db my_lib2.db}

Loading db file 'path/libraries/ex/my_lib1.db'
Loading db file 'path/libraries/ex/my_lib2.db'
{}

lc_shell> list -libraries

Library  File         Path
------     ----       ----
 Lib_1     lib.db path/libraries/ex
 lib1      my_lib1.db path/libraries/ex
 lib2      my_lib2.db/libraries/ex
```

Note that library lib.db was loaded before the latest `read` command.

Adding Library-Level Information to Existing Libraries

The `update_lib` command lets you add library-level information to an existing library. The kind of information you can add includes cells, operating conditions, timing ranges, scaling factors, bus type definitions, interconnect wire loads, and other group statements. Updated groups are added to the library as if they were typed at the end of the original text of the library file.

You can also use the `update_lib` command to change groups by overwriting them with a subsequent `update_lib` command. However, you cannot overwrite the original data in a .db file.

Note:
You cannot overwrite existing `type` groups, because their definitions might already have affected subsequent group declarations in the library. You can, however, add new `type` groups.

For a complete listing of the groups and attributes you can specify in technology library, see Chapter 1, “Library-Level Attributes and Groups,” in the *Library Compiler Technology and Symbol Libraries Reference Manual*.

Using the update_lib Command

This is the syntax for using the `update_lib` command to add new groups to an existing library:

```
update_lib [-overwrite] [-permanent] library_name filename [-no_warnings]
```

-overwrite

Indicates that the groups declared in *filename* overwrite existing groups with the same name in the *library_name* library. This command overwrites only those groups that were added using a previous `update_lib` command; you cannot use this command to overwrite groups included in the original library.

-permanent

Indicates that the group declared in *filename*, after being written into *library_name* library, cannot be overwritten anymore by a subsequent `update_lib` command.

-force

Overwrites a limited set of library-level groups and attributes included in the original library. The groups that you can overwrite are the `operating_conditions` group, the `wire_load` group, and the `power_supply` group. The attributes that you can overwrite are the simple attributes of a library object, which are described in the source files.

library_name

Name of the existing library to which you add the new groups. The library name is the name assigned with the library (*library_name*) statement in the library .lib file.

filename

Name of the file where one or more new groups is found. The syntax of this file is similar to a normal Library Compiler library description, except it contains only library-level groups without the `library` group definition.

-no_warnings

Suppresses all warning messages. Use this option sparingly; warnings can identify serious library problems.

An example of the command is

```
lc_shell> update_lib -overwrite -permanent cmos add_ms.lib
```

This command adds the groups in `add_ms.lib` to the cmos library and overwrites similarly named groups in cmos that were added by a previous `update_lib` command. These new groups cannot be overwritten by subsequent `update_lib` commands.

Updating Attribute Values

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

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

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

Note:
You can apply the `set_lib_attribute` command only to libraries in which the library vendor has set the value of the `library_features` attribute to `allow_update_attribute`. For more information about using the `library_features` attribute, see the *Library Compiler Technology and Symbol Libraries Reference Manual*.

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

Example

```
lc_shell> set_lib_attribute [example/lib_udg1, example/lib/
lib_udg2] lib_udg_attr
new_value
Performing set_lib_attribute on library object ‘example/
lib_udg1’
Performing set_lib_attribute on library object ‘example/lib/
lib_udg2’
{"example/lib_udg1",
"example/lib/lib_udg2"
```

Adding Memory Modules

To add a memory module to an existing library, use the `update_lib` command. However, if you do not have a Library Compiler license, you can add groups with module functionality by using the `add_module` command. The `add_module` command performs the same function as `update_lib`, except that it can add memory modules only.

Syntax

This is the syntax for adding groups with memory to an existing library without a Library Compiler license:

```
add_module [-overwrite] [-permanent] filename library_name
[-no_warnings]
```

-overwrite

Indicates that the group declared in *filename* overwrites an existing group with the same name in the *library_name* library. You can overwrite only groups that were added by a previous `update_lib` command; you cannot overwrite groups included in the original library.

-permanent

Indicates that the groups declared in *filename*, after being written into *library_name* library, cannot be overwritten anymore by a subsequent `update_lib` command.

filename

Name of the file where one or more new groups is found. The syntax of this file is similar to a normal Library Compiler library description, except it contains only library-level groups without the `library` group definition.

library_name

Name of the existing library to which you add the new groups. The library name is the name assigned with the library (*library_name*) statement in the library .lib file.

-no_warnings

Suppresses all warning messages. Use this option sparingly; warnings can identify serious library problems.

In this example, the library file add_ram.lib, which contains module information (that is, a RAM group), is added to the library memory.

```
lc_shell> add_module add_ram.lib memory
```

Processing

If Library Compiler finds an error while processing a group, it does not add that group to the library; the `update_lib` and `add_module` operations return 0 if illegal groups are present.

The `report_lib` command places an asterisk (*) after the names of all cells added to the library with the `update_lib` and `add_module` commands. For additional information, see “Reporting Library Information” on page 3-25.

Accessing Library Information

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

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

Accessing Simple Attribute Values

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

```
get_lib_attribute {object_name, ...} attribute_name
```

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

Example

```
dc_shell> get_lib_attribute {example/lib_udg1, example/lib/
lib_udg2] lib_udg_attr
Performing get_lib_attribute on library object ‘example/
lib_udg1’
Performing get_lib_attribute on library object ‘example/
lib_udg2’
```

```
{"value1",
"value_2"}
```

Reporting Library Information

The `report_lib` command displays information about the technology libraries, physical libraries, and symbol libraries compiled in the Library Compiler database.

- For technology libraries, the `report_lib` command displays timing, power, electromigration, functionality, and FPGA information upon request. Cells are annotated according to the attributes specified for them.
- For physical libraries, the `report_lib` command lists the site, layer, routing wire model, macro, and process resource information.
- For symbol libraries, the `report_lib` command lists the names of all symbols defined in the library.

Syntax

```
report_lib library
[-timing] [-timing_arcs]
[-timing_label][-power]
[-power_label]
[-em] [-table] [-full_table]
[-vhdl_name]
[-fpga] [-user_defined_data]
[-all]
```

library

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

-timing

Prints a report of timing information for verification. You can use the `-timing` option only when you read the .lib file into the Library Compiler program during the same session. See the “Generating Library Reports” chapter in the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide* for information about the timing report format.

-timing_arcs

Prints a list of the timing arcs defined for each cell. Example 3-2 shows an example of a per-cell timing arc report.

-timing_label

Prints all the timing arc label information.

-power

Prints a report of power information for each cell that has power modeling, as shown in Example 3-1.

-power _label

Prints all the power label information.

-em

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

-table

Prints compact statetable information for technology library cells.

-full_table

Prints complete statetable and memory information for technology library cells.

-vhdl_name

Prints a report of proposed VHDL names that differ from the Library Compiler names.

-fpga

Prints the library defaults and, if they exist, the part group and the I/O attributes.

-user_defined_data

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

-all

Prints all library information. This option applies only to technology libraries.

Example 3-1 Power Information Report for Cells With Power Modeling

```
Attributes:

a - average power specification
i - internal power
l - leakage power
rf - rise and fall power specification

                                  Power

Cell           #     Attr        Toggling pin      Source of path
when
_________________________________________________________________
_________
EO             0     i,a         Z                 B                 A'
               1     i,a         Z                 B                 A
               2     i,a         Z                 B
               3     i,a         Z                 A                 B'
               4     i,a         Z                 A                 B
```

```
              5     i,a        Z               A
E01           0     i,a        Z               B               A’
              1     i,a        Z               B               A
              2     i,a        Z               A
              3     i,a        Z               B
              4     i,a        Z               A               B’
              5     i,a        Z               A               B
```

Example 3-2 Per-Cell Timing Arc Report

```
Cell    Attributes   #     Sense/Type                From    To
-----------------------------------------------------------------
dff                  0     rising edge               CLK     Q
                     1     preset neg unate          PRE     Q
                     2     clear pos unate           CLR     Q
                     3     rising edge               CLK     QN
                     4     clear pos unate           PRE     QN
                     5     preset neg unate          CLR     QN
```

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

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

The following example displays information for the cmos library:

```
lc_shell> report_lib cmos
```

When reporting on cells, Library Compiler might indicate they have the following attributes:

active_falling and active_rising

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

active_high and active_low

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

black box

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

clock_enable

This is the clock-enabling mechanism.

dont_touch

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

map_only

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

preferred

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

removable

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

sa0, sa1, sa01

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

statetable

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

dont_use

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

test_cell

This is a test cell.

If a library cell has footprint information and the library is enabled for in-place optimization with the `in_place_swap_mode` attribute, the `report_lib` command prints the cell footprint for each cell, as shown in the following example. See Chapter 6, “Building a Technology Library,” for information about this attribute.

```
   Cell                 Footprint


----------------------------------------------
   M21H                 "FP_F"
   M21L                 "FP_F"
   M41H                 "FP_H"
   M41L                 "FP_H"
   R01H                 "FP_A"
   R01L
"FP_A"
```

Listing Resident Libraries

The `list` command displays all information about various lc_shell items, such as the value of a variable, the names of the libraries in memory, the names of all lc_shell commands, the names and values of variables in different variable groups, and any licenses currently being used.

Syntax

```
list [-commands] [-licenses] [variable_name]      [-libraries]
[-variables variable_group]
```

-commands

Lists all lc_shell commands.

-licenses

Lists the licenses you have checked out. Use the lc_shell command `license_users` to list all licenses in use.

variable_name

Lists the value of the specified lc_shell variable.

-libraries

Lists all libraries residing in memory.

-variables *variable_group*

Lists the names and values of variables defined in one of the following groups. The available `variable` groups are configurable and can be changed, or new `variable` groups can be created with the `group_variable` command.

- all
  All lc_shell variables
- system
  Global system variables
- vhdlio
  Variables that affect writing VHDL files and libraries

For information on predefined `variable` groups, use the `help` command with *variable_group*. For example,

```
lc_shell> help system_variables
```

This example lists the value of the designer variable:

```
lc_shell> list designer
```

This example lists the values of the variables in the vhdlio `variable` group:

```
lc_shell> list -variables vhdlio
```

After executing `read_lib cmos.lib` in this example, the `list -libraries` command displays the following:

```
lc_shell> list -libraries

Library    File      Path
-------    ----      ----
cmos       cmos.db   /path/
libraries
```

After executing `read_lib -format edif -symbol cmos.slib cmos.edif`, the libraries listing shows

```
lc_shell> list -libraries

Library    File      Path
-------    ----      ----
cmos       cmos.sdb  /path/
libraries
cmos       cmos.db   /path/
libraries
```

Comparing Libraries

The `compare_lib` command compares a technology library with either a physical library or a symbol library for consistency. This command

- Verifies that each cell in the technology library has a corresponding definition in a physical library or a symbol library.
- Checks that the pin names for each cell in the technology library match the pin names defined for that cell's corresponding symbol.

Syntax

compare_lib *filename1 filename2*

filename1, *filename2*

Names the libraries that you want to compare. One library must be a technology library; the other can be either a physical library or a symbol library. Each argument is a string. The order in which the file names appear is not important.

The libraries must exist before you can use the `compare_lib` command. Use the `list` command to display the libraries that reside in memory.

In the following example, the `list` command displays the existing libraries that reside in memory; the `compare_lib` command compares the `cmos.db` technology library with the `cmos.sdb` symbol library and then with the cmos.pdb physical library.

```
lc_shell> list -libraries


Library    File      Path
-------    ----      ----
cmos       cmos.db   /path/libraries
cmos       cmos.sdb  /path/libraries
cmos       cmos.pdb  /path/libraries

lc_shell> compare_lib cmos.db:cmos cmos.sdb:cmos
lc_shell> compare_lib cmos.db:cmos cmos.pdb:cmos
```

Library Compiler reports any discrepancies between these libraries.

For a complete description of the `compare_lib` command, see the Synopsys man pages.

Removing Libraries From Memory

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

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

Syntax

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

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

lib_list

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

-all

Removes all libraries in memory.

-quiet

Prevents informational messages from being displayed.

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

Encrypting VHDL Source Libraries

Use the `encrypt_lib` command to secure a VHDL source library file. The `encrypt_lib` command lets you protect the security of VHDL models that contain proprietary information. The encrypted VHDL file can be parsed by the Synopsys VHDL System Simulator (VSS).

Syntax

```
encrypt_lib file_name [-output encrypted_file]
```

file_name

Name of the VHDL source library file to be encrypted.

-output *encrypted_file*

Writes the encrypted output to a file name different from the default *current_directory/file name*.

If you do not use the `-output` option, your encrypted output file has the name *current_directory/file name*.

This is an example of the command:

```
lc_shell> encrypt_lib my_file.vhd
```

Using Shell Commands

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

Command Aliases

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

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

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

The `unalias` command removes alias definitions.

alias Command

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

Syntax

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

identifier

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

definition

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

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

Example

```
lc_shell> alias my_writelib "write_lib -format vhdl"
```

After creating the `my_writelib` alias, you could write a library in VHDL format by simply entering the alias, as shown in the next example. (The example assumes that you have a library called lib1 already loaded into lc_shell.)

Example

```
lc_shell> my_writelib lib1
```

The lc_shell program does not echo the entered alias definition.

unalias Command

The `unalias` command removes alias definitions.

Syntax

```
unalias [-all | alias ...]
```

-all

Removes all alias definitions.

alias ...

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

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

Example

```
lc_shell> unalias my_writelib
```

Listing Previously Entered Commands

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

"Reexecuting Commands" on page 3-35 explains how to reexecute a previous command.

Syntax

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

n

Lists up to *n* previously entered commands.

-r

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

-h

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

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

```
lc_shell> history
```

To list only the previous ten commands, enter

```
lc_shell> history 10
```

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

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

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

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

```
lc_shell> include history.script
```

Reexecuting Commands

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

!!

Reexecutes the last command.

!-*n*

Reexecutes the *n*th command from the last.

!*n*

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

!*text*

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

!?*text*

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

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

```
lc_shell> !!
```

last_command

If you enter the following commands,

```
lc_shell> read_lib -format edif -symbol cmos.slib cmos.edif
lc_shell> write_lib cmos
```

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

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

To list the command history, enter

```
lc_shell> history
     1 read _lib -format edif -symbol cmos.sdb cmos.edif
     2 write_lib cmos
     3 write_lib cmos -output cmos1.db
     4 history
```

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

```
lc_shell> !1
read_lib -format edif -symbol cmos.sdb cmos.edif
```

To reexecute the previous `read_lib` command (the last command starting with re), enter

```
lc_shell> !re
read_lib -format edif -symbol cmos.sdb cmos.edif
```

To reexecute the last `read_lib` command (the last command containing edif), enter

```
lc_shell> !?edif
read_lib -format edif -symbol cmos.sdb cmos.edif
```

Shell Commands

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

pwd

Lists the Library Compiler working directory.

cd *directory*

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

ls *directory*

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

sh *"command"*

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

Example 3-3 shows how to use the four shell commands, starting by invoking the lc_shell program from a system prompt.

Example 3-3 Using Four Shell Commands

```
<prompt> cd
<prompt> pwd
/path
<prompt> lc_shell
Library Compiler (TM)
...
Initializing...
lc_shell> pwd
/path
lc_shell> cd edif_designs
lc_shell> pwd
/path/edif_designs
lc_shell> ls
my_lib1.edif
my_lib2.edif
my_libs.script

lc_shell> sh "more my_designs.script"
read /path/edif_designs/
my_design1.edif
set_system_variable ...
...
list -files
...
pwd
...
lc_shell> cd
lc_shell> pwd
/path
```

Command Scripts

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

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

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

Using the include Command

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

Syntax

```
include file
```

file

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

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

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

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

```
lc_shell> search_path = {., my_dir, /path/libraries}
{., my_dir, /path/libraries}
lc_shell> include example.script
command
command /
*comment*/
. . .
```

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

Command Status

After lc_shell executes a command, it sets a status value in the `lc_shell_status` system variable. This value is the same as the value lc_shell displays after processing a command.

```
lc_shell> alias myread "read_lib -format edif -no_warnings"
1
```

The command status value returned for the previous `alias` command is 1, indicating successful command completion. You can also use the `list` command to show the value of the `lc_shell_status` variable.

```
lc_shell> list lc_shell_status
lc_shell_status = 1
```

The `lc_shell_status` variable is the only variable that changes type. The `lc_shell_status` type depends on the type of the return value from the last lc_shell command.

If a command (such as a `read_lib` command that tries to read a library file containing an error) cannot execute properly, its return value is an error status value. The error status value is 0 for most commands and a null list ({}) for commands that return a list.

```
lc_shell> list no_such_var
Error: Variable 'no_such_var' is undefined. (UI-1)
0

lc_shell> read_lib -format edif -symbol bad_lib.slib bad_lib.edif
Error . . .
{}
```

Following is an example of checking for a null list:

```
if (lc_shell_status == null) {
     quit
}
```

Using Control Flow Commands

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

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

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

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

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

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

The condition expression is evaluated as a Boolean variable. Table 3-3 shows how each non-Boolean variable type is evaluated. For example, the integer 0 is evaluated as a Boolean false. A nonzero integer is evaluated as a Boolean true. Use condition expressions to compare two variables of the same type or a single variable of any type. All variable types have Boolean evaluations.

Table 3-3 Boolean Evaluations of Non-Boolean Types

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

if Command

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

Syntax

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

This is how the `if` command is executed:

1. When `if_expression` is true, execute all `if_commands`.

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

Each expression is evaluated as a Boolean variable (see Table 3-3).

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

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

```
vendor_library = my_lib
if ((vendor_library != Xlib) && (vendor_library
!= Ylib)) {
  vendor_library = Xlib
}
if (vendor_library == Xlib) {
  link_library = Xlib.db
  font_library = 1_25.font
} else {
  link_library = Ylib.db
  font_library = 1_30.font
}
```

In this example, Library Compiler loads a library and checks it for errors; if it finds errors, the script quits. Otherwise, the value of the `lc_shell_status` variable controls the read of the library.

```
read_lib my_lib.lib
if (lc_shell_status != 1) {
  quit
}
```

while Command

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

Syntax

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

If *expression* is true, Library Compiler executes all `while` commands repeatedly. Library Compiler evaluates the expression as a Boolean variable (see Table 3-3).

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

If Library Compiler encounters the `break` command, command execution jumps to the next command after the end of the `while` loop.

An expression can test the return value of a statement by checking the `lc_shell_status` system variable. In this way, continued command execution depends on successful completion of the previous command.

The following example displays 0 to 19, exiting the `while` loop after the count is greater than 20 (the `if` command returns 0 to lc_shell_status):

```
count = 0
while (lc_shell_status != 0) {
  count = count + 1
  if (count < 20 ) {
    list count
  }
}
```

foreach Command

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

Syntax

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

variable_name

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

list

Any valid expression containing variables or constants.

foreach_command

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

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

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

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

The result of this command is

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

continue Command

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

Syntax

```
continue
```

The following example is similar to the previous `while` example, but this script skips the display of the first three elements in the list.

```
count = 0
while (lc_shell_status != 0) {
  count = count + 1
  if (count = 3) {
    continue
  }
  if (count < 20) {

    list count
  }
}
```

break Command

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

Syntax

```
break
```

The following example displays commands 1 to 19 but breaks out of the `while` loop when the count is greater than 19 (the `if` command returns 0 to lc_shell_status):

```
count = 1
while (count > 0) {
  if (count < 20) {
    list count
}
if (lc_shell_status != 1) {
  break
}
count = count + 1
}
```

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

Using the Wildcard Character

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

u*

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

u*z

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

*

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

Using Variables

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

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

name

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

value

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

A variable can be separate from, or part of, a variable group. For example, the `system` variable contains the `designer` and `company` variables.

For information on predefined variable groups, use the `help` command with the name of the variable group. For example,

```
lc_shell> help system_variables
```

The following sections explain how to set, create, list, and remove variables.

Setting Variable Values

To set the value of a variable, enter the variable name, an equal sign (=), and the appropriate value. Library Compiler echoes the new value.

```
lc_shell> designer = "brian"
brian

lc_shell> vhdllib_architecture = {"VITAL"}
{"VITAL"}

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

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

Creating Variables

To create a new variable, use the same syntax you use to set an existing variable. The lc_shell program issues a warning message when a new variable is created (in case you misspelled the name of an existing variable), then echoes the value of the new variable. A value can be a number, a string, or a path as shown in the examples.

```
lc_shell> numeric_var = 107.3
Warning: Defining new variable ’numeric_var’.
107.3

lc_shell> my_var = "my_value"
Warning: Defining new variable ’my_var’.
my_value


lc_shell> list_var = {/home/frank, /usr/libraries, /tmp}
Warning: Defining new variable ’list_var’.
{/home/frank, /usr/libraries, /tmp}
```

Listing Variable Values

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

```
lc_shell> list designer
designer = "brian"

lc_shell> list numeric_var
numeric_var = 107.3
```

Removing Variables

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

```
lc_shell> remove_variable my_var
```

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

Creating Variable Groups

You can combine related variables in variable groups with the `group_variable` command. Most predefined variables are part of a group, such as the system variable group or the vhdlio variable group.

Variable groups are a convenience for relating variables. The only operations you can perform on variable groups are adding and listing the member variables.

Syntax

```
group_variable  group_name  "variable_name"
```

group_name

New or existing variable group name.

"*variable_name*"

Variable added to *group_name*. The *variable_name* must be enclosed in quotation marks; otherwise, its value is used as the variable name.

For example, to create a new variable group called `my_var_group`, enter

```
lc_shell> group_variable my_var_group "my_var"
lc_shell> group_variable my_var_group numeric_var
```

After you create a variable group, you cannot delete it. However, you can remove a variable from the group with the `remove_variable` command.

Listing Variable Groups

You can list variable and system group members and their values.

To list variable group members and their values, use the `list -variables` command with the variable group's name:

```
lc_shell> list -variables my_var_group
        my_var              my_value
        numeric_var        107.3
```

To list the Library Compiler system group members and their values, use the `list -variables` command as follows:

```
lc_shell> list -variables system
auto_link_options "-all"
command_log_file "./lc_command.log"
...
echo_include_commands "true"
verbose_messages
"true"
```

4

Using Library Compiler Syntax

Library Compiler compiles three types of libraries: technology libraries, physical libraries, and symbol libraries. All three types of libraries use a single syntactic format. For details about the syntax used in library descriptions, see the *Library Compiler Technology and Symbol Libraries Reference Manual* and *Library Compiler Physical Libraries Reference Manual*. See Appendix A in the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide* for an example of a technology library.

Group statements, attribute statements, definition statements, and comments describe a library. To build a library, you must understand the following concepts explained in this chapter:

- Library File Names
- Library, Cell, and Pin Names
- Statements
- Comments
- Units of Measure
- Library Example

Library File Names

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

.lib

Technology library source files

.plib

Physical library source files

.pplib

Pseudo-physical library source files

.slib

Symbol library source files

.db

Compiled technology libraries in Synopsys database format

.pdb

Compiled physical libraries in Synopsys database format

.sdb

Compiled symbol libraries in Synopsys database format

.vhd

Generated VHDL simulation libraries

Library, Cell, and Pin Names

The following rules apply to naming conventions for libraries:

- Use the same name for libraries, cells, and pins as the vendor uses in its libraries.
- Names are case-sensitive. For example, the name AND2 is not the same as And2.
- Library Compiler accepts names of any length.
- Enclose names that do not begin with an alphabetic character in quotation marks (" "). For example, express a cell named 2NAND as "2NAND". This rule has exceptions; see the sections on the `function` attribute and the `related_pin` attribute in the *Library Compiler Technology and Symbol Libraries Reference Manual*.

- Each identifier must be unique within its own context. Library Compiler flags each duplicate cell name in a library as a warning and flags each duplicate pin name in a cell as an error.

Statements

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

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

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

Group Statements

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

This is the general syntax of a group statement:

Syntax

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

name

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

Example 4-1 defines `pin` group A.

Example 4-1 Group Statement Specification

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

Attribute Statements

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

In this 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, but complex attributes have differing syntactic requirements.

Unless the manual indicates otherwise, you can use a particular attribute only one time within a group statement. If an attribute is used more than once, Library Compiler recognizes only the last attribute statement.

Simple Attributes

Simple attributes have the following syntax:

Syntax

```
attribute_name : attribute_value ;
```

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

Example 4-2 adds a `direction` attribute to the `pin` group in Example 4-1.

Example 4-2 Simple Attribute Specification

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

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

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

For certain simple attributes, you must enclose the attribute value in quotation marks. Example 4-3 adds the function X + Y to the pin example.

Example 4-3 Defining the Function of a Pin

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

Complex Attributes

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

Syntax

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

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

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

Define Statements

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

Defining New Groups

The `define_group` statement lets you create new groups.

Syntax

define_group (*group_name*$_{id}$, *parent_name*$_{id}$,);

group_name

The name of the new group you are creating.

parent_name

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

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

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

Defining New Attributes

The `define` statement lets you create new simple attributes. The syntax for `define` statements is similar to the syntax for complex attributes.

Syntax

define (*attribute_name*$_{id}$, *group_name*$_{id}$, *attribute_type*$_{enum}$);

attribute_name

The name of the new attribute you are creating.

group_name

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

attribute_type

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

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

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

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

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

Comments

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

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

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

Library Compiler ignores all characters between the delimiters.

Units of Measure

Design Compiler requires that a design use the same units of measure as those used in the ASIC vendor's library. For example, if the unit of time used in the vendor's library is nanoseconds, then you must enter all timing values in nanoseconds. If picofarads are the unit of capacitance in the vendor's library, then you must enter all capacitance values in picofarads. All units and entries in an ASIC vendor's library must be consistent to ensure accurate timing calculation.

You must assign the appropriate physical time units when you use Library Compiler to generate VHDL simulation libraries. Library Compiler supports four discrete time units: 1 ns, 100 ps, 10 ps, and 1 ps. See the section "library Group" on page 6-3 for more time unit information.

You can optionally supply a voltage unit, current unit, resistance unit for pull ups and pull downs, capacitive load unit, and power unit. See "Defining Units" on page 6-13 for additional information.

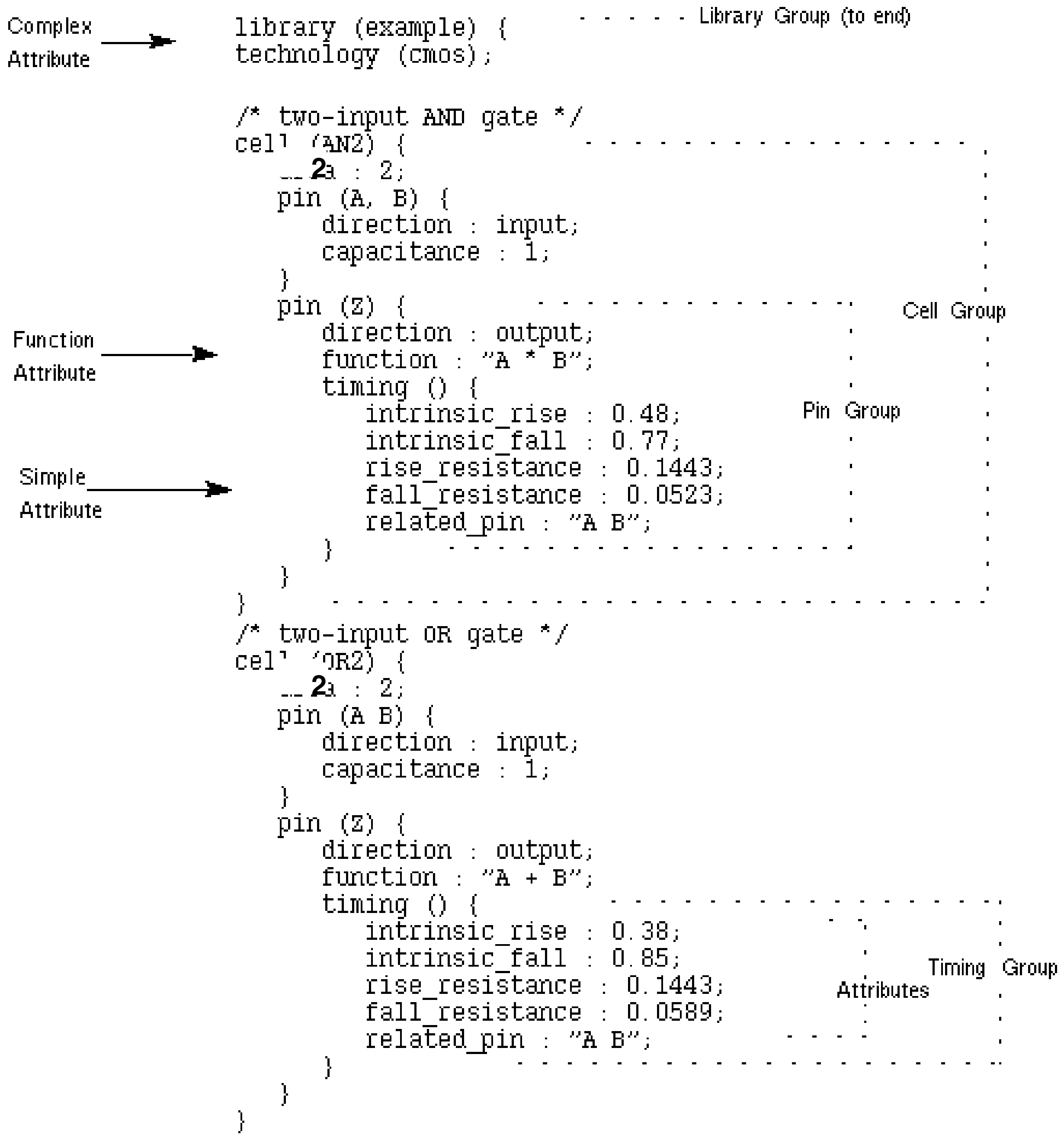
The ASIC vendor determines the measure of area used. You should express area in equivalent gates or in some other internal metric. Take care to use the same unit of measure throughout the library. This area consistency requirement also applies to the estimation of wire area. Table 4-1 shows commonly used units of measure.

Table 4-1 Commonly Used Units of Measure

Component	**Unit of measure**
Area	Equivalent gates
	Square microns for standard cells
	Cell-based custom designs
Capacitance	Picofarads
	Standardized loads
Delay	Nanoseconds
Resistance	Kilohms
Temperature	Degrees centigrade
Voltage	Volts

Library Example

The following example shows a CMOS technology library describing three cells: AN2, OR2, and IV1.



```
library (example) {
technology (cmos);

/* two-input AND gate */
cell (AN2) {
   area : 2;
   pin (A, B) {
      direction : input;
      capacitance : 1;
   }
   pin (Z) {
      direction : output;
      function : "A * B";
      timing () {
         intrinsic_rise : 0.48;
         intrinsic_fall : 0.77;
         rise_resistance : 0.1443;
         fall_resistance : 0.0523;
         related_pin : "A B";
      }
   }
}
/* two-input OR gate */
cell (OR2) {
   area : 2;
   pin (A B) {
      direction : input;
      capacitance : 1;
   }
   pin (Z) {
      direction : output;
      function : "A + B";
      timing () {
         intrinsic_rise : 0.38;
         intrinsic_fall : 0.85;
         rise_resistance : 0.1443;
         fall_resistance : 0.0589;
         related_pin : "A B";
      }
   }
}
```

5

Building a Symbol Library

The symbol libraries contain the information that the Design Vision tools use to generate and display the graphic representation of a design. The construction of symbol libraries is technology-independent.

To build a symbol library, you must know about the following concepts described in this chapter:

- Symbol Libraries
- Symbol Library Group
- layer Group
- symbol Group
- Special Symbols
- Library Compiler Checks

Symbol Libraries

Symbol libraries share the same fundamental structure and syntax as technology libraries. Typically, a symbol library is built for use with a particular technology library, although the actual construction of the library is technology-independent.

Each cell in the technology library must have a cell with the same name in the symbol library; however, the symbol library might include cells that have no corresponding cell in the technology library.

A symbol library also contains special symbols that have no corresponding cell in the technology library. These special symbols are used to draw the parts of the schematic that are not cells, such as connectors, template borders, and text.

Creating Symbol Libraries

You can create a new symbol library in one of these ways:

- Using the Mentor Graphics CAE interface
- Using the EDIF interface
- Using the Synopsys Integrator for the Mentor Falcon Framework
- Writing a text description

Using one of the automated interfaces, such as a CAE-specific or EDIF interface, is faster and more accurate than writing a text description.

Mentor Graphics CAE Interface

The Mentor Graphics CAE interface lets you extract symbol library information from the Mentor Graphics system. This interface appears at the schematic level. The schematic includes the bounding box and pin locations of each symbol, so schematics transmitted to the Mentor system are compatible with the symbols used to display the schematic on that system.

Schematics generated with symbols extracted from the Mentor system contain only rectangle symbols when the Design Analyzer tool displays them. However, when you use Mentor EDIF, the symbols are fully displayed. For more information, see the *Mentor Interface Application Note* and the *EDIF 2 0 0 Interface User Guide*.

EDIF Interface

Design Compiler supports the EDIF 2 0 0 design interface to and from Synopsys synthesis tools. This interface provides a way to transfer symbol information between Design Compiler and many CAE systems. You can create an almost complete symbol library from the data in an EDIF schematic view file.

Synopsys Integrator for the Mentor Falcon Framework

The Mentor Falcon Framework is a system that lets you use a variety of software tools and data types within one environment. Mentor's Design Manager tool is the graphical user interface (GUI) to the framework. Other tools that you can use include Design Vision and Design Analyzer, the Synopsys graphical environment tools, and Synopsys VHDL Simulation tools.

Text File Symbol Libraries

If you cannot use a CAE interface or the EDIF interface to generate a symbol library, you can create a text file description of the symbol graphics, and then read it into `lc_shell`. You can then use Library Compiler to generate a compiled symbol library.

Note:
You can have only one symbol library in a file. Create the technology and schematic descriptions in separate files.

Symbol Library File Names

There are no predefined file naming conventions. Synopsys uses and recommends the following suffixes for symbol library file names:

.slib

Symbol library files

.sdb

Compiled symbol libraries in Synopsys internal database format

.edif

Symbol libraries derived from EDIF systems

Synopsys Generic Symbol Library

Design Compiler comes with a symbol library containing more than 170 of the most common cell functions. This symbol library is in a file called generic.sdb. The symbols defined in this generic symbol library include most Boolean logic symbols.

The Design Analyzer tool automatically searches generic.sdb for symbols unless you use another file name for the `generic_symbol_library` variable, as described in the section "Defining a Symbol Library for Design Analyzer" on page 5-5.

If the Design Analyzer tool does not find a symbol for an element (such as a flip-flop, latch, or MUX) in generic.sdb during schematic generation, the program draws a rectangle.

Note:
 You can find the text version of generic.sdb in the $SYNOPSYS/libraries directory.

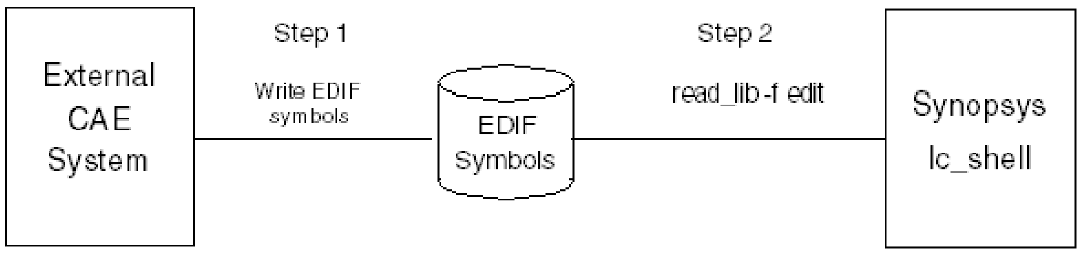
Schematics generated with the generic symbol library cannot be written to an external CAE system. To allow transfer back to the external system, the symbols must come from a symbol library generated from the information in the external system's libraries.

Transferring a Symbol Library From a CAE System

Figure 5-1 shows the data and control flow of reading CAE symbols into Design Compiler. The flow follows these steps:

1. The external CAE system creates the EDIF library file containing the system symbols.
2. The Library Compiler `read_lib -f edif` command creates a symbol library from the EDIF library file containing the CAE system symbols.

Figure 5-1 Reading Symbols



Defining a Symbol Library for Design Analyzer

These two variables defined in lc_shell name the symbol libraries to use when creating schematics:

generic_symbol_library

Names the default symbol library. This variable is required. The value is usually generic.sdb, the Synopsys generic symbol library.

symbol_library

Names a user-defined symbol library. Usually the library named by symbol_library contains the symbol descriptions for a third-party technology library.

To use a custom library, set the `symbol_library` variable to the name of the library you want to use.

```
lc_shell> symbol_library = libname
```

The Design Analyzer tool searches for the symbols required to draw your schematic in the library named by the `symbol_library` variable. If Design Analyzer cannot find the library, it searches the library named in `generic_symbol_library` that supplies most of your symbols.

Note:

You can use only one symbol library with EDIF descriptions. A custom symbol library must include all the symbols in your technology library, plus the special symbols listed in "Special Symbol Attributes" on page 5-9. If you use a symbol from the Synopsys generic library, you cannot export it back to the originating system. Generic symbols do not transfer back to EDIF systems.

Fonts in Symbol Libraries

The fonts used with the Design Analyzer tool are stored in the /libraries subdirectory where the Synopsys tools are installed. Files that end with .flib contain the information you need for creating and using fonts.

To use specific fonts, prepare them for the Design Analyzer tool by following these steps:

1. Preprocess the .flib file with the C preprocessor in your environment. On a UNIX system, at the system prompt, type a line that uses the following syntax:

```
usr/lib/cpp <font_name.flib | grep -v "^#" >font.cpp
```

2. Read the preprocessed file into lc_shell.

```
lc_shell> read_lib font.cpp
```

3. Write it out to a file.

```
lc_shell> write_lib "font_name.font"
```

You can now use this file as a font in your symbol libraries.

Symbol Library Group

The symbol library text file syntax includes the symbol library group, library variables, symbol dimensioning, and the `set_route_grid` attribute.

The `library` group statement names the library being described. This statement must be the first executable line in your library.

```
library ( library_name ) {
   ... library group description ...
}
```

Use a library name that matches the symbol library file name, but do not include the suffix. For example, to name a library example.slib:

```
library ( example ) {
  ... symbol descriptions ...
}
```

The `library` group of a symbol library contains

- Variables
- The `set_route_grid` and `set_meter_scale` attributes
- Special symbol attributes
- Layer groups
- Symbol groups

Library Variables

Define variables at the beginning of symbol libraries.

Syntax

```
variable_name = variable_value;
```

For example, to define true and false variables, place these lines at the beginning of the library:

```
true = 1;
false = 0;
```

Symbol Dimensioning Using Variables

You can use sizing variables one time to define a symbol and then have Library Compiler generate the symbol in a size appropriate to your sheet size.

The example symbol library in generic.slib defines two sets of sizing variables: one based on the ANSI standards and one based on MIL-STD-8086. You can use these two standards as models for other sizing standards you want to use.

Example 5-1 shows the definition of a set of sizing variables for AND gates. The variables are defined according to ANSI standards.

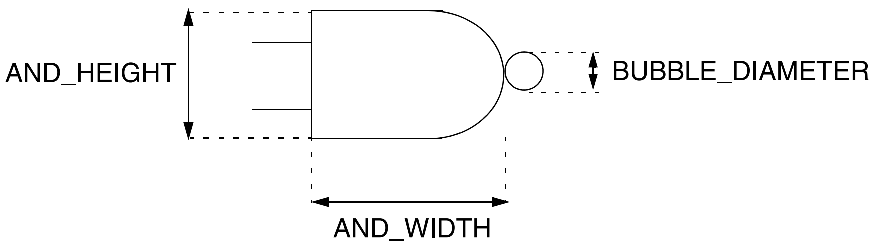
Example 5-1 Defining Sizing Variables

```
/* REQUIRED SIZES: */
AND_HEIGHT = 4 ;

/* ANSI Dimensions: */
ANSI_AND_HEIGHT = 26.0 ;
SCALE = AND_HEIGHT / ANSI_AND_HEIGHT ;

AND_WIDTH = 32 * SCALE ;
BUBBLE_DIAMETER = 4 * SCALE ;
```

Only the first two sizes are defined by a number. All other sizes are defined by the scale determined by the two variables.

Figure 5-2 shows an AND gate defined by these sizing variables. The dimensions of the symbols are labeled.

Figure 5-2 Symbol Height and Width Variables



set_route_grid Attribute

Schematic symbols are defined with floating-point numbers but are stored as an integer representation. The `set_route_grid` complex attribute controls floating-point number conversion to integers.

Syntax

```
set_route_grid ( route_grid ) ;
```

route_grid

An integer that determines the resolution of the grid by defining the number of values between each set of two whole floating-point values. The Design Analyzer tool routes wires only on the grid, so the pin representing the starting point of a wire must be on the grid. The `route_grid` value must be a power of 2; the suggested value is 1024. The font libraries and the generic symbol library use 1024 as the `route_grid` value.

set_meter_scale Attribute

The EDIF interface creates the `set_meter_scale` complex attribute whenever you use the EDIF interface to read in an EDIF design. The EDIF interface uses the attribute to correct the schematic when it is written back to the originating system. Do not add this attribute to your library.

Special Symbol Attributes

The special symbol attributes define the symbols for many of the schematic elements. These special symbols do not represent cells or have corresponding cells in a technology library. Define special symbols by assigning the following attributes in the `library` group:

logic_1_symbol : *"symbol_name"* ;

Names the symbol for logic 1 in schematics.

logic_0_symbol : *"symbol_name"* ;

Names the symbol for logic 0 in schematics.

in_port_symbol : *"symbol_name"* ;

Names the symbol for input ports in schematics.

out_port_symbol : *"symbol_name"* ;

Names the symbol for output ports in schematics.

inout_port_symbol : *"symbol_name"* ;

Names the symbol for bidirectional ports in schematics.

in_osc_symbol : *"symbol_name"* ;

Names the symbol for input off-sheet connectors in multiple-page schematics.

out_osc_symbol : *"symbol_name"* ;

Names the symbol for output off-sheet connectors in multiple-page schematics.

inout_osc_symbol : *"symbol_name"*;

Names the symbol for bidirectional off-sheet connectors in multiple-page schematics.

The following example assigns symbols defined in the generic symbol library to the special symbols.

```
logic_1_symbol : "logic_1" ;
logic_0_symbol : "logic_0" ;
in_port_symbol : "in_port" ;
out_port_symbol : "out_port" ;
inout_port_symbol : "io_port" ;
in_osc_symbol : "in_osc" ;
out_osc_symbol : "out_osc" ;
inout_osc_symbol : "out_osc" ;
```

Caution:

If you use generic symbols, you cannot move the resulting schematics to an outside system.

layer Group

The layers in the Design Analyzer tool define the appearance of objects in a schematic. These characteristics include color, visibility, font, and line width. You can think of layers as transparent sheets, each with different information drawn on it, placed one on top of another.

Syntax

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

name

Name of one of the predefined text layers described in Table 5-1. These are the only valid layers. Each layer contains a specific type of information that gives you control over what is shown in your schematics.

Table 5-1 Layers in Symbol Libraries

Layer	Contents
bus_net_layer	Bused nets
bus_net_name_layer	Bused net names
bus_net_type_layer	Bus types
bus_osc_layer	Bus off-sheet connectors
bus_osc_name_layer	Bus off-sheet connector name
bus_port_layer	Bus ports
bus_port_name_layer	Bus port names
bus_port_width_layer	Widths of bus ports
bus_ripper_layer	Bus ripper symbols
bus_ripper_name_layer	Bus ripper symbol names

Table 5-1 Layers in Symbol Libraries (Continued)

Layer	Contents
bus_ripper_type_layer	Bus ripper types
cell_layer	Cell symbols
cell_name_layer	Cell names
cell_ref_name_layer	Corresponding technology cell names
clock_layer	Clock icons
constraint_layer	Constraint flags
hierarchy_layer	Hierarchy boxes and symbols
hierarchy_name_layer	Hierarchy boxes and symbol names
highlight_layer	Highlighted objects
net_layer	Net lines
net_name_layer	Net names
osc_layer	Off-sheet connector symbols
osc_name_layer	Off-sheet connector names
pin_layer	Pin symbols
pin_name_layer	Pin names
port_name_layer	Port names
port_layer	Port symbols
template_layer	Template
template_text_layer	Text associated with the template
text_layer	Text not assigned to other layers
variable_layer	Text on a template, defined as a variable

layer Group Attributes

To define the visual characteristics of a layer, use the following attributes in a `layer` group:

set_font (*"font_file"*) ;

Names the font to be used on the layer. Each layer has a defined font, even if the layer does not normally have text on it.

visible : true | false ;

Determines whether the Design Analyzer tool shows the layer by default.

line_width : *width* ;

Determines the width of the lines used to draw the objects on the layer. The width is defined by an integer.

scalable_lines : true | false ;

Determines whether lines are scaled in a zoomed view. When true, line width is directly proportional to the zoom factor. When false, the line width does not scale at all. In a drawing where scaling is allowed, lines in a zoomed view are much thicker than they are in an unzoomed view. For example, `scalable_lines` on the `net_layer` has a default of false, so nets are shown as 1 at all zoom levels.

red : *value* ;

Sets the red saturation of objects on the layer for the X Window System. The value is an integer from 0 to 65,536.

blue : *value* ;

Sets the blue saturation of objects on the layer for the X Window System. The value is an integer from 0 to 65,536.

green : *value* ;

Sets the green saturation of objects on the layer for the X Window System. The value is an integer from 0 to 65,536.

layer Group Example

Example 5-2 shows two layer definitions from the Synopsys generic symbol library.

Example 5-2 Layer Definitions in a Symbol Library

```
layer(cell_layer) {
  set_font ("1_25.font") ;
  visible : true ;
  line_width : 256 ;
  scalable_lines : true ;
  red : 65000 ;
```

```
    green : 65000 ;
    blue : 0 ;
  }

  layer(cell_name_layer) {
    set_font ("1_25.font") ;
    visible : true ;
    line_width : 1 ;
    red : 65000 ;
    green : 33000 ;
    blue : 0 ;
  }
```

Both layers are visible and both use the 1_25 font, although no text appears on `cell_layer` in this example. The lines in `cell_layer` are scaled in proportion to the zoom factor. The text in `cell_name_layer` does not scale by default. The `cell_layer` is light green, and the `cell_name_layer` is orange.

symbol Group

The `symbol` group defines everything that is drawn. The group defines the cell symbols, which represent the components in your technology library, and the special symbols, which represent all other graphics used in a schematic, such as template borders and off-sheet connectors. The `symbol` group is described in the `library` group as follows:

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

name

Name of the corresponding component in the technology library, or the name of a special symbol.

The `symbol` group contains the attributes that define the cell symbol. These include graphic attributes defining the shape, pins, and text associated with the symbol and those defining other cell characteristics. The `symbol` group includes the following types of attributes:

- Graphic attributes
- Cell attributes
- General attributes
- Template attributes

Graphic Attributes

The following graphic attributes are complex attributes used to draw cell symbols. They are defined in a `symbol` group.

line (*x1*, *y1*, *x2*, *y2*) ;

Creates a line from (*x1*, *y1*) to (*x2*, *y2*). The *x1*, *x2*, *y1*, and *y2* values are floating-point numbers.

arc (*x1*, *y1*, *x2*, *y2*, *x3*, *y3*) ;

Creates a clockwise arc starting at (*x1*, *y1*), ending at (*x2*, *y2*), and centered on (*x3*, *y3*). The *x1*, *y1*, *x2*, *y2*, *x3*, and *y3* values are floating-point numbers.

circle (*x*, *y*, *R*) ;

Creates a circle centered at (*x*, *y*) with radius *R*. The *x*, *y*, and *R* values are floating-point numbers.

pin (*name*, *x*, *y*, *direction*) ;

Creates a pin named *name* with the connection points (*x* and *y*) on a symbol with the approach named *direction*. The direction is LEFT, RIGHT, UP, DOWN or ANY_ROTATION. The *x* and *y* values are the following types when *direction* has these values:

- LEFT or RIGHT, *x* is floating-point and *y* is integer.
- UP or DOWN, *x* is integer and *y* is floating-point.
- ANY_ROTATION, both *x* and *y* are floating-point. With this value, Library Compiler determines the rotation of the pin, using appropriate algorithms.

The number of pins defined for a symbol in the symbol library must match the number pins defined in the technology library. You can check this correspondence while verifying your libraries. See the “Verifying CMOS Libraries" chapter in the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide*.

Note:
Include only the base name of a bused pin as the name of the symbol library pin, so that one symbol represents cells of the same type but with variable-width input.

See Table 5-2 for a comparison of bused and single-pin formats.

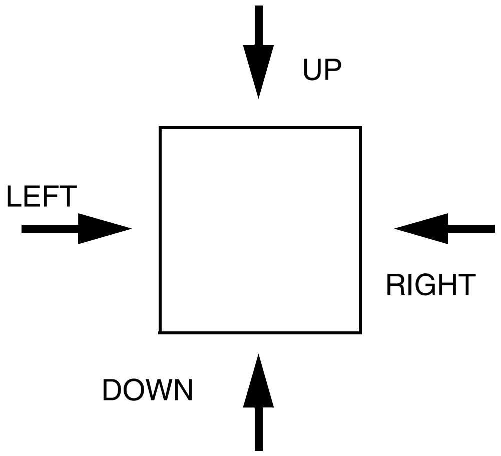
Table 5-2 Comparison of Bused and Single-Pin Formats

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

Pin Approach Directions

The four approach directions for defining pins are LEFT, RIGHT, UP, and DOWN, as illustrated in Figure 5-3. The schematic generator uses these approach directions to determine how to bring wires into the pin.

Figure 5-3 Pin Approach Directions



LEFT and RIGHT must have integer y coordinates, and UP and DOWN must have integer x coordinates.

The following pin declarations are valid:

```
pin( a, 2, 3, LEFT ) ;
pin( b, 2.25, 4.0, RIGHT ) ;
pin( enable, -1, 1.234, UP ) ;
pin( count, 5, 5.5, DOWN ) ;
```

The following pin declarations are not valid:

```
pin( a, 3.1, 3.4,LEFT ) ;    /* fractional y */
pin( b, 1.5, 1, UP ) ;       /* fractional x */
```

Cell Attributes

The following cell attributes are defined in a `symbol` group:

set_minimum_boundary (*min_x*, *min_y*, *max_x*, *max_y*) ;

Defines a minimum symbol size. The symbol's bounding box is guaranteed to be at least this large, even if the symbol design itself is not. A minimum boundary is created for the undefined symbols. The *min_x*, *min_y*, *max_x*, and *max_y* values are integers.

ripped_pin : *pin_name* ;

Identifies a symbol as a bus ripper and denotes which port is the bus that is broken up. A ripper symbol has two ports, both of undefined width: one bus port and one output port.

The `pin_name` is the name of the bus port. It must have a corresponding `pin` attribute in the symbol definition. See the next section for a complete ripper symbol description.

ripped_bits_property : *property_name*;

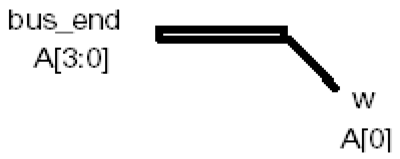
Uses optional attributes to attach an EDIF property name to a ripper.

Bus Ripper

A bus ripper is a symbol that extracts one wire from a group of wires in a bus. One end of the ripper is attached to the bus and has the width of the bus; the other end is a single bit. This is an example of a ripper symbol definition.

```
symbol (ripper1x2) {
  ripped_pin : "bused_port" ;
  ripped_bits_property : "port" ;
  pin (bused_port,1,1,LEFT);
  pin (wire1,2,1,RIGHT) ;
  line (1,1,3,1) ;
}
```

In the following illustration, bus_end is connected to A[3:0] and w represents A[0]:



Unlike off-sheet connectors and power or ground symbols, the bus ripper symbol has no special symbol keyword. The schematic generator identifies a cell with a `ripped_pin` attribute as a bus ripper. A ripper has one bused pin and one single-bit pin. An example is

```
symbol (ripper) {
   ripped_pin : "bus_end" ;
   ripped_bits_property : "port" ;
   pin (bus_end,-.25 * OFF_SHEET_HEIGHT,0,LEFT);
   pin (w,-.5 * OFF_SHEET_HEIGHT,0,RIGHT)
   line (1,1,3,1) ;
}
```

The schematic generator looks for a bus ripper in the symbol library named by the Design Compiler `symbol_library` variable. If it does not find one, it uses the ripper provided by Synopsys in the generic symbol library generic.slib.

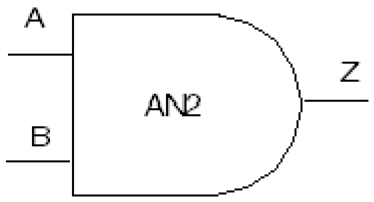
You can use symbols from your own symbol library and the bus ripper from the generic symbol library. However, if the scaling or `route_grid` value in the generic symbol library is different from that of your own symbol library, you cannot generate an EDIF schematic netlist. Design Compiler does not allow you to write a schematic netlist that contains cells of different scaling factors.

If you do not want buses or if your target CAE system does not support buses, use the `-no_bus` option with the `create_schematic` command.

Symbol Definition Example

Figure 5-4 shows the graphic cell symbol for a cell called AN2.

Figure 5-4 Cell Symbol AN2



Example 5-3 is the complete definition of the cell symbol in Figure 5-4.

Example 5-3 Cell Symbol Definition

```
symbol(AN2) {

   AND_LEFT_X = - AND_X_ORIGIN ;
   AND_BOTTOM_Y = - AND_Y_ORIGIN ;
   AND_TOP_Y = AND_BOTTOM_Y + AND_HEIGHT ;
   X_START_OF_ARC = AND_LEFT_X + AND_WIDTH - AND_HEIGHT / 2.0 ;
   AND_MIDDLE_Y = AND_BOTTOM_Y + AND_HEIGHT / 2.0 ;
   AND_RIGHT_X = AND_LEFT_X + AND_WIDTH ;

   line(AND_LEFT_X, AND_TOP_Y, X_START_OF_ARC, AND_TOP_Y) ;
   line(AND_LEFT_X,AND_BOTTOM_Y,X_START_OF_ARC,AND_BOTTOM_Y);
   line(AND_LEFT_X, AND_BOTTOM_Y, AND_LEFT_X, AND_TOP_Y) ;

   arc(X_START_OF_ARC, AND_TOP_Y, X_START_OF_ARC,\
     AND_BOTTOM_Y, X_START_OF_ARC, AND_MIDDLE_Y) ;

   pin(A, AND_LEFT_X, AND_BOTTOM_Y + 3, LEFT) ;
   pin(B, AND_LEFT_X, AND_BOTTOM_Y + 1, LEFT) ;
   pin(Z, AND_RIGHT_X, AND_MIDDLE_Y, RIGHT) ;
}
```

The variables at the beginning of this example define the origin and size of the symbol in terms of variables defined for the library as a whole. You can find examples of these variables in the example symbol library in /libraries/generic.slib. The line and arc graphic attribute statements follow the variables. The last three lines describe the three pins on the cell. The pin attribute statements define placement and direction.

General Attributes

You can build symbols from other symbols. To save space and time when constructing a symbol library, describe general-purpose symbols and use them to build other symbols.

Use the `sub_symbol` complex attribute to include these general-purpose symbols in the definition of another symbol.

sub_symbol (*symbol_name*, *x*, *y*, *r*) ;

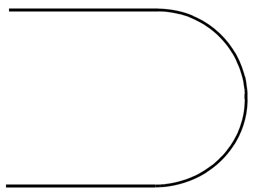
Includes another symbol in the definition of the current symbol. The origin of the subsymbol is placed at (*x*, *y*) with rotation *r*. The *symbol_name* is the name of the symbol you want to include. The *r* value is the angle of rotation, from 0 to 360 degrees. The *x* and *y* values are integers.

Note:
You must define a symbol in the symbol library before you can use it as a subsymbol.

A subsymbol is useful for describing several similar components. For example, you might use it to describe two AND gates: AN2, defined in Example 5-3, and AN3, which is similar to AN2 but has three inputs instead of two.

The symbols for these components have some parts in common. Specifically, they have the same top and bottom line and the same arc at the front, as shown in Figure 5-5. You can create a symbol of this shape and later use it to create complete cell symbols. Example 5-4 shows the symbol definition for the subsymbol of Figure 5-5.

Figure 5-5 Subsymbol Graphic



First, define the `library` group level `and_outline` subsymbol, as shown in Example 5-4.

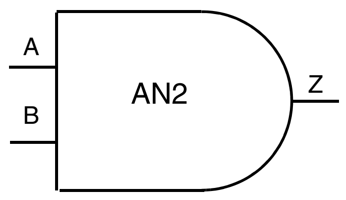
Example 5-4 Defining a Subsymbol

```
symbol(and_outline) {
AND_LEFT_X = - AND_X_ORIGIN ;
AND_BOTTOM_Y = - AND_Y_ORIGIN ;
AND_TOP_Y = AND_BOTTOM_Y + AND_HEIGHT ;
X_START_ARC = AND_LEFT_X + AND_WIDTH - AND_HEIGHT / 2.0 ;
AND_MIDDLE_Y = AND_BOTTOM_Y + AND_HEIGHT / 2.0 ;
AND_RIGHT_X = AND_LEFT_X + AND_WIDTH ;

line(AND_LEFT_X, AND_TOP_Y, X_START_ARC, AND_TOP_Y) ;
line(AND_LEFT_X, AND_BOTTOM_Y, X_START_ARC, AND_BOTTOM_Y) ;
arc(X_START_ARC, AND_TOP_Y, X_START_ARC, AND_BOTTOM_Y, \
    X_START_ARC, AND_MIDDLE_Y) ;
}
```

Only the top and bottom lines and the arc along the right side of the symbol are defined; the left side is not yet defined. The cell symbol definition can identify the left side. You can use the subsymbol in a variety of cell symbol definitions.

After the `and_outline` subsymbol is defined, you can use it in the description of the AN2 cell pictured in Figure 5-6.

Figure 5-6 Subsymbol in a Symbol Graphic (AN2)



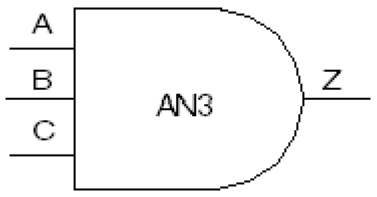
Example 5-5 describes the AN2 cell.

Example 5-5 Using a Subsymbol in a Cell Symbol Description (AN2)

```
symbol(AN2) {

  sub_symbol(and_outline, 0,0,0) ;

  line(AND_LEFT_X, AND_BOTTOM_Y, AND_LEFT_X, AND_TOP_Y) ;

  pin(A, AND_LEFT_X, AND_BOTTOM_Y + 3, LEFT) ;
  pin(B, AND_LEFT_X, AND_BOTTOM_Y + 1, LEFT) ;
  pin(Z, AND_RIGHT_X, AND_MIDDLE_Y, RIGHT) ;
}
```

To define an AND gate with three inputs (AN3), use the AN2 subsymbol and define a left side with three input pins, as shown in Figure 5-7. Example 5-6 shows a subsymbol in a symbol graphic (AN3).

Figure 5-7 Subsymbol in a Symbol Graphic (AN3)



Example 5-6 Using a Subsymbol in a Cell Symbol Description (AN3)

```
symbol(AN3) {

  sub_symbol(and_outline, 0,0,0) ;

  line(AND_LEFT_X, AND_BOTTOM_Y, AND_LEFT_X, AND_TOP_Y) ;

  pin(A, AND_LEFT_X, AND_BOTTOM_Y + 3, LEFT) ;
  pin(B, AND_LEFT_X, AND_BOTTOM_Y + 2, LEFT) ;
  pin(C, AND_LEFT_X, AND_BOTTOM_Y + 1, LEFT) ;
  pin(Z, AND_RIGHT_X, AND_MIDDLE_Y, RIGHT) ;
}
```

Note:

The `compare_lib` command in `lc_shell` flags `sub_symbol` declarations as extra symbols. Typically, a subsymbol appears only in a generic symbol library that has no corresponding technology library for comparison.

Template Attributes

Use a sheet template when you display your schematics. Templates define the border boxes and the identification block for each sheet. Build templates for each sheet size you use and include the American A-E sheets, the European A0-A4 sheets, a Mentor-specific sheet, and an infinite sheet. Each sheet has two versions: one for landscape orientation and one for portrait orientation. A landscape sheet is wider than it is tall; a portrait sheet is taller than it is wide.

Templates are built in symbol groups with the following attributes at the symbol group level:

template : *"template_name"* ;

Defines a symbol as a template. The *template_name* is the name of the template.

landscape : true | false ;

Identifies the orientation of the template: true gives a landscape template and false gives a portrait orientation.

variable (*"variable_name"*, *x*, *y*, *"layer"*) ;

Stores the text associated with the template. The *variable_name* is any variable available in Design Compiler. Some common values are `current_design`, `company`, and `sheet`.

The *x* and *y* coordinates are integers indicating the position of the text on the template. The value of *layer* determines which layer the Design Analyzer tool uses to display the text; the value is one of the layer names listed in Table 5-1 on page 5-10.

set_usable_area (*x1*, *y1*, *x2*, *y2*) ;

Indicates the portion of the sheet that can occupy the schematic drawing. The (*x1* and *y1*) coordinates are integers defining the location of the lower-left corner in `route_grids`; (*x2* and *y2*) are integers identifying the upper-right corner.

set_infinite_template_location (*x*, *y*) ;

Determines the placement of the template on a sheet of paper. An infinite template produces the entire schematic on one sheet.

The x and y values in the attribute are floating-point numbers between 0 and 1. Using (0,0) puts the lower-left corner of the schematic in the lower-left corner of the sheet. Using (0.5 and 0.5) puts the lower-left corner of the schematic in the center of the sheet. Use this attribute instead of `set_usable_area` when you describe an infinite template. Example 5-7 is an example of an infinite template.

Example 5-7 Describing an Infinite Template

```
symbol(template_infinite) {
    template : "infinite" ;
    landscape : true ;
    sub_symbol(key_box, - KEY_BOX_WIDTH, \
               - KEY_BOX_HEIGHT - FONT_SIZE * 2, 0) ;
    set_infinite_template_location(0,0) ;
/* Starts the infinite template at the lower left corner */
}
```

Building the Borders

You define the borders of a template in a `symbol` group by using the attributes described previously in this chapter. Example 5-8 is an example of a template designed to fit on an A-size sheet.

Example 5-8 A-Size Sheet Template

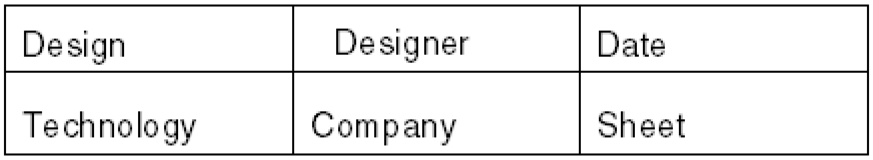
```
symbol(template_A){
 template : "A" ;
 landscape : true ;
 sub_symbol(key_box, A_SHEET_WIDTH - KEY_BOX_WIDTH -FONT_SIZE/2,\
 FONT_SIZE /2,0); set_usable_area(FONT_SIZE / 2, KEY_BOX_HEIGHT + \
 FONT_SIZE,A_SHEET_WIDTH-FONT_SIZE / 2, A_SHEET_HEIGHT-FONT_SIZE / 2);
 /* box around all */line(0, 0, A_SHEET_WIDTH, 0) ;
line(A_SHEET_WIDTH, 0,A_SHEET_WIDTH,A_SHEET_HEIGHT);
line(A_SHEET_WIDTH, A_SHEET_HEIGHT, 0, A_SHEET_HEIGHT) ;
line(0, A_SHEET_HEIGHT, 0,0) ;}
```

The template symbol is called template. The template name, A, is in the Design Analyzer tool menu. The subsymbol is `key_box`. Example 5-9 shows the definition of a key box subsymbol. The usable area of the sheet is defined as the area inside the borders but outside the key box.

Building the Key Box

A key box is a subsymbol used with the template symbol. The key box, usually placed in the lower right corner, gives information about the design and its creator. Figure 5-8 shows the key box defined in Example 5-9.

Figure 5-8 Key Box Graphic Symbol



Example 5-9 Key Box Subsymbol Definition

```
KEY_BOX_WIDTH = A_SHEET_HEIGHT - FONT_SIZE ;
KEY_BOX_HEIGHT = 4 * FONT_SIZE ;

symbol(key_box) {
  /* box around all */
   line(0, 0, KEY_BOX_WIDTH, 0) ;
   line(KEY_BOX_WIDTH, 0, KEY_BOX_WIDTH, KEY_BOX_HEIGHT) ;
   line(KEY_BOX_WIDTH, KEY_BOX_HEIGHT, 0, KEY_BOX_HEIGHT) ;
   line(0, KEY_BOX_HEIGHT, 0,0) ;

   /* inner lines */
   line(0,KEY_BOX_HEIGHT / 2, KEY_BOX_WIDTH, KEY_BOX_HEIGHT/ 2);
   line((1.0/3.0) * KEY_BOX_WIDTH,0,(1.0/3.0) * KEY_BOX_WIDTH, \
        KEY_BOX_HEIGHT) ;
   line((2.0/3.0) * KEY_BOX_WIDTH, 0, \
      (2.0/3.0) * KEY_BOX_WIDTH,KEY_BOX_HEIGHT) ;

   /* variables */
   LOW_Y = FONT_SIZE / 2 ;
   HIGH_Y = LOW_Y + KEY_BOX_HEIGHT / 2 ;

   LEFT_X = FONT_SIZE / 2 ;
   MIDDLE_X = FONT_SIZE / 2 + (1.0 / 3.0) * KEY_BOX_WIDTH ;
   RIGHT_X = FONT_SIZE / 2 + (2.0 / 3.0) * KEY_BOX_WIDTH ;

   variable("design", LEFT_X, HIGH_Y,  "template_text_layer") ;
   variable("technology", LEFT_X, LOW_Y,"template_text_layer") ;
   variable("designer",MIDDLE_X, HIGH_Y,"template_text_layer") ;
   variable("company", MIDDLE_X, LOW_Y, "template_text_layer") ;
   variable("date", RIGHT_X, HIGH_Y, "template_text_layer") ;
   variable("sheet", RIGHT_X, LOW_Y, "template_text_layer") ;
}
```

Special Symbols

In addition to the component symbols, a symbol library contains a number of special symbols. Special symbols include page borders, off-sheet connectors, a bus ripper, and the primitives that make up pin symbols. Make special symbols as elaborate or as simple as you like.

Describe special symbols the same way you describe cell symbols. Example 5-10, taken from the example symbol library in generic.slib, describes the `logic_0` and `in_osc` symbols.

Example 5-10 Defining Special Symbols

```
symbol(logic_0) {
  line(0,0,0,- 1.25) ;
  line( -.75,- 1.25,.75,- 1.25) ;
  line( -.75,- 1.27,0,- 2) ;
  line(.75,- 1.27,0,- 2) ;
  pin(a,0,0,RIGHT) ;
}

symbol(in_osc) {
  sub_symbol(osc, 0, 0, 0) ;
  pin(a,.5 * OFF_SHEET_HEIGHT, 0, RIGHT) ;
}
```

The value for `OFF_SHEET_HEIGHT` is assigned in the variable definitions at the beginning of the library. You can scale this symbol to fit any template defined in a library.

The completed library has a symbol for each of the following special symbols:

- `logic_1`
- `logic_0`
- `in_port`
- `out_port`
- `ripper`
- `inout_port`
- `in_osc`
- `out_osc`
- `inout_osc`

Library Compiler Checks

When you use the Design Compiler `read_lib` command to load a symbol library, the command performs consistency checks for the following:

- Duplicate symbols and duplicate pins on a symbol.
- Existence of special symbols in the library. Special symbols include power and ground symbols, as well as in, out, and inout ports.
- Invalid pin definitions. For example, two pins cannot have the same approach direction (LEFT) and the same coordinate values (*x*, *y*).
- Pin definitions that do not fall exactly on the grid.
- Duplicate layer definitions.
- Symbols with no name.

6

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

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.

This chapter describes these `library` group level attributes. See the “Building Environments” chapter in the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide* for information about the default attributes and scaling factors, which are other library-level attributes. See also the *Library Compiler Technology and Symbol Libraries Reference Manual* for specific information about the attributes.

The library description also contains several group statements that are defined at the `library` group level. Chapter 7, “Defining Core Cells,” explains how to define these groups.

Syntax

Example 6-1 shows the general syntax of the library. The first statement names the library. The following statements are library-level attributes that apply to the library as a whole. These statements define library features such as technology type, date, and revision, as well as definitions and defaults that apply to the library in general. Every cell in the library has a separate cell description.

For a complete list and the syntax of all the groups and attributes in a technology library, see the *Library Compiler Technology and Symbol Libraries Reference Manual*.

Example 6-1 General Syntax of the Library Description

```
library (name) {
   technology (name) ;/* library-level attributes */
   delay_model : generic_cmos | table_lookup | cmos2 | piecewise_cmos | dcm;
   bus_naming_style : string ;
   routing_layers(string);
   date : string ;
   revision : float_or_string ;
   comment : "string" ;
   time_unit : unit ;
   voltage_unit : unit ;
   current_unit : unit ;
   pulling_resistance_unit : unit ;
   capacitive_load_unit(value,unit);
   leakage_power_unit : unit ;
   piece_type : type ;
   piece_define ("list") ;
   define_cell_area(area_name,resource_type);
   in_place_swap_mode : match_footprint | no_swapping ;
   library_features (value);
   simulation : true | false ;
```

```
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
}
```

library Group

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

Example

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

Using General Library Attributes

These attributes apply generally to the technology library:

- technology
- delay_model
- bus_naming_style
- routing_layers

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, Library Compiler defaults to cmos.

Example

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

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 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;
   ...
}
```

bus_naming_style Attribute

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

Example

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

Note:
You cannot redefine the `bus_naming_style` attribute of a technology library from the lc_shell or dc_shell prompt.

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.

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.
```

Documenting Libraries

Use these library-level attributes to document the library:

- date
- revision
- comment

date Attribute

This attribute identifies the date your library was created.

Example

```
date : "September 1, 2005";
```

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

revision Attribute

This attribute defines a revision number for your library.

Example

```
revision : 2005.01;
```

comment Attribute

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

Example

```
comment : "Copyright 2005, General Silicon, Inc."
```

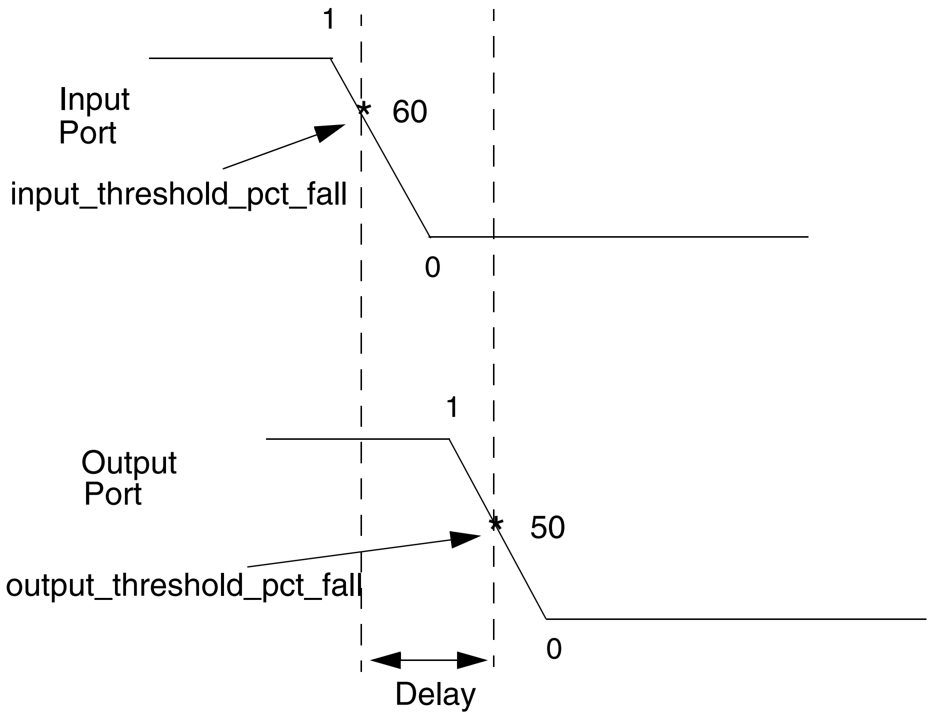
Delay and Slew Attributes

This section describes attributes used to set the values of the input and output pin threshold points that Library Compiler uses 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 6-1).

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 6-1 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:

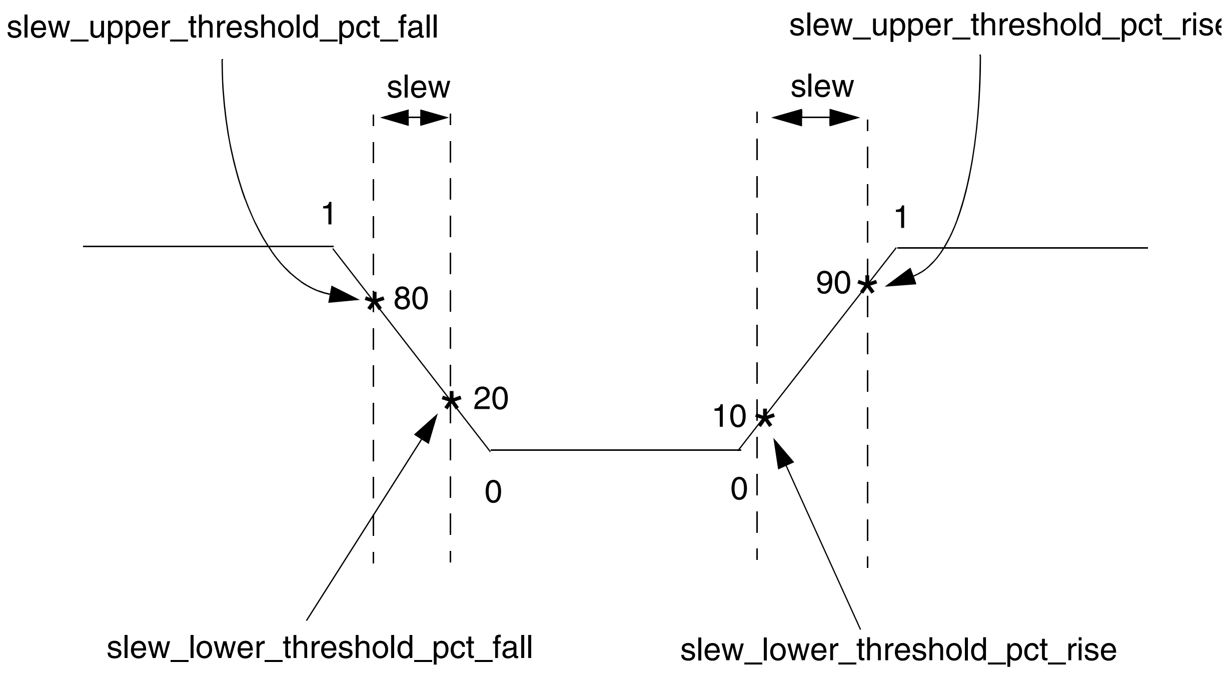
- slew_lower_threshold_pct_fall
- slew_upper_threshold_pct_fall

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 6-2 shows an example of slew modeling.

Figure 6-2 Slew Modeling


slew_upper_threshold_pct_fall
slew_upper_threshold_pct_rise
slew
slew
1
80
20
0
0
10
90
1
slew_lower_threshold_pct_fall
slew_lower_threshold_pct_rise

input_threshold_pct_fall Simple Attribute

Note:
To enable Library Compiler to model the delay of a signal going from an input pin to an output pin, you also need to set the value of the output pin. See "output_threshold_pct_fall Simple Attribute" on page 6-9 and "output_threshold_pct_rise Simple Attribute" on page 6-10 for details.

Use the `input_threshold_pct_fall` attribute to set the value of the threshold point on an input pin signal falling from 1 to 0. Library Compiler uses this value in modeling the delay of a signal transmitting from an input pin to an output pin.

Syntax

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

trip_point

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

Example

```
input_threshold_pct_fall : 60.0 ;
```

input_threshold_pct_rise Simple Attribute

Use the `input_threshold_pct_rise` attribute to set the value of the threshold point on an input pin signal rising from 0 to 1. Library Compiler uses this value in modeling the delay of a signal transmitting from an input pin to an output pin.

Note:
To enable Library Compiler to model the delay of a signal going from an input pin to an output pin, you also need to set the value of the output pin. See "output_threshold_pct_fall Simple Attribute" and "output_threshold_pct_rise Simple Attribute" on page 6-10 for details.

Syntax

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

trip_point

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

Example

```
input_threshold_pct_rise : 40.0 ;
```

output_threshold_pct_fall Simple Attribute

Use the `output_threshold_pct_fall` attribute to set the value of the threshold point on an output pin signal falling from 1 to 0. Library Compiler uses this value in modeling the delay of a signal transmitting from an input pin to an output pin.

Note:

To enable Library Compiler to model the delay of a signal going from an input pin to an output pin, you also need to set the value of the input pin. See "input_threshold_pct_rise Simple Attribute" and "input_threshold_pct_fall Simple Attribute" on page 6-8 for details.

Syntax

```
output_threshold_pct_fall : trip_point_value ;
```

trip_point

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

Example

```
output_threshold_pct_fall : 40.0 ;
```

output_threshold_pct_rise Simple Attribute

Use the `output_threshold_pct_rise` attribute to set the value of the threshold point on an output pin signal rising from 0 to 1. Library Compiler uses this value in modeling the delay of a signal transmitting from an input pin to an output pin.

Note:

To enable Library Compiler to model the delay of a signal going from an input pin to an output pin, you also need to set the value of the input pin. See "input_threshold_pct_rise Simple Attribute" on page 6-9 and "input_threshold_pct_fall Simple Attribute" on page 6-8 for details.

Syntax

```
output_threshold_pct_rise : trip_point_value ;
```

trip_point

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

Example

```
output_threshold_pct_rise : 40.0 ;
```

slew_derate_from_library Simple Attribute

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

Syntax

```
slew_derate_from_library : derate_value ;
```

derate

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

Example

```
slew_derate_from_library : 0.5 ;
```

slew_lower_threshold_pct_fall Simple Attribute

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

Note:

To enable Library Compiler to model the delay of a pin falling from 1 to 0, you also need to set the value for the upper threshold point. See "slew_lower_threshold_pct_rise Simple Attribute" for details.

Syntax

```
slew_lower_threshold_pct_fall : trip_point_value ;
```

trip_point

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

Example

```
slew_lower_threshold_pct_fall : 30.0 ;
```

slew_lower_threshold_pct_rise Simple Attribute

Use the `slew_lower_threshold_pct_rise` attribute to set the value of the lower threshold point Library Compiler uses in modeling the delay of a pin rising from 0 to 1.

Note:

To enable Library Compiler to model the delay of a pin rising from 0 to 1, you also need to set the value for the upper threshold point. See "slew_upper_threshold_pct_fall Simple Attribute" on page 6-12 for details.

Syntax

```
slew_lower_threshold_pct_rise : trip_point_value ;
```

trip_point

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

Example

```
slew_lower_threshold_pct_rise : 30.0 ;
```

slew_upper_threshold_pct_fall Simple Attribute

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

Note:
To enable Library Compiler to model the delay of a pin falling from 1 to 0, you also need to set the value for the lower threshold point. See "slew_lower_threshold_pct_fall Simple Attribute" on page 6-11 for details.

Syntax

```
slew_upper_threshold_pct_fall : trip_point_value ;
```

trip_point

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

Example

```
slew_upper_threshold_pct_fall : 70.0 ;
```

slew_upper_threshold_pct_rise Simple Attribute

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

Note:
To enable Library Compiler to model the delay of a pin rising from 0 to 1, you also need to set the value for the lower threshold point. See "slew_lower_threshold_pct_rise Simple Attribute" on page 6-11 for details.

Syntax

```
slew_upper_threshold_pct_rise : trip_point_value ;
```

trip_point

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

Example

```
slew_upper_threshold_pct_rise : 70.0 ;
```

Defining Units

Design Compiler is unitless. However, units are required to create VHDL libraries and reports.

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. Library Compiler does not do any conversions.

time_unit Attribute

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

Example

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

voltage_unit Attribute

Library Compiler uses this attribute to scale the contents of the `input_voltage` and `output_voltage` groups. Additionally, the `voltage` attribute in the `operating_conditions` group represents values in the voltage units. See the “Building Environments” chapter in the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide* for more information.

Example

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

current_unit Attribute

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

Example

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

pulling_resistance_unit Attribute

Library Compiler uses this attribute to define pulling resistance values for pull-up and pull-down devices.

Example

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

capacitive_load_unit Attribute

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

Example

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

leakage_power_unit Attribute

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

Example

```
leakage_power_unit : 100uW;
```

Using Piecewise Linear Attributes

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

- piece_type
- piece_define

piece_type Attribute

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

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.

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");
```

See the *Library Compiler Technology and Symbol Libraries Reference Manual* for more information on the `piece_define` attribute.

Describing Pads

Use these library-level groups and attributes for pad descriptions:

- input_voltage group
- output_voltage group
- preferred_output_pad_slew_rate_control attribute
- preferred_output_pad_voltage and preferred_input_pad_voltage attributes

input_voltage Group

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

Note:
You can also use the `input_voltage` group to specify voltages ranges for standard cells. For more information, see the *Library Compiler User Guide: Modeling Power and Timing Technology Libraries*.

Example

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

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

For example, you can define an `input_voltage` group called TTL with a set of high and low thresholds and minimum and maximum voltage levels. Use the following command in the `pin` group to assign those ranges to the pad cell pin.

```
input_voltage : TTL ;
```

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

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

output_voltage Group

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

Note:
You can also use the `input_voltage` group to specify voltages ranges for standard cells. For more information, see the *Library Compiler User Guide: Modeling Power and Timing Technology Libraries*.

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

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

Example

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

}
```

preferred_output_pad_slew_rate_control Attribute

This attribute directly embeds the desired slew rate control value in the library.

Example

```
preferred_output_pad_slew_rate_control : "high";
```

If defined, the value of the `preferred_output_pad_slew_rate_control` attribute in the library is used if a value for a `preferred_slew_rate_control` attribute has not been specified. The pad-mapping optimization within the `insert_pads` and `compile` dc_shell commands uses this information when selecting the candidate set of pads for mapping.

Note:

If the `preferred_output_slew_rate_control` value is set within a library, the library should supply at least one output pad, one three-state pad, and one bidirectional pad that have the `slew_rate_control` attribute within the `cell` group set to a value equivalent to the preferred value.

For more information about slew-rate control, see “Slew-Rate Control” on page 9-10.

preferred_output_pad_voltage and preferred_input_pad_voltage Attributes

These attributes embed the desired voltage values in the library. The preferred voltage values are the names of output and input voltage groups defined in the library.

Examples

```
preferred_output_pad_voltage : "out_volt_group_one" ;
preferred_input_pad_voltage : "in_volt_group_six" ;
```

Describing Power

You can describe power dissipation in libraries, using the CMOS nonlinear delay model.

To describe power,

1. Use the library-level `power_lut_template` group to define templates of common information to use with lookup tables.
2. Use the template and the cell-level `internal_power` group in the `pin` group to create lookup tables of power information (see the “Modeling Power and Electromigration” chapter in the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide*).

Lookup tables and their corresponding templates can be one-, two-, or three-dimensional.

power_lut_template Group

Use this library-level group to create templates of common information that multiple lookup tables can use. A table template specifies the table parameters and the breakpoints for each axis. Assign each template a name. So that power lookup tables can refer to the template, place its name as the group name of a `fall_power` group, `power` group, or `rise_power` group in the `internal_power` group (see the next section, “Template Variables”).

This is the syntax of the `power_lut_template` group.

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

Template Variables

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

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

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

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

total_output_net_capacitance

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

equal_or_opposite_output_net_capacitance

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

input_transition_time

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

For information about the `related_pin` attribute, see the “Timing Arcs” chapter in the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide*.

Template Breakpoints

The following index statements define the breakpoints for an axis.

- `index_1` specifies the breakpoints of the first dimension defined by `variable_1`.
- `index_2` specifies the breakpoints of the second dimension defined by `variable_2`.
- `index_3` specifies the breakpoints of the third dimension defined by `variable_3`.

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

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

The number of floating-point numbers in the indexes determines the size of each dimension, as Example 6-2 illustrates.

You must define at least one `index_1` statement in the `power_lut_template` group. For a one-dimensional table, use only `variable_1`.

Example 6-2 shows four `power_lut_template` groups that have one-, two-, and three-dimensional templates.

Example 6-2 Four power_lut_template Groups

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

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

}

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

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

Note:
There is a predefined template whose name is scalar and whose size is 1. You can refer to it by placing the string, *scalar*, as the group name of a `fall_power` group, `power` group, or `rise_power` group in the `internal_power` group.

Using Processing Attributes

Use these attributes to define processing options:

- `in_place_swap_mode`
- `library_features`
- `simulation`

in_place_swap_mode Attribute

In-place optimization occurs after placement and routing when Design Compiler optimizes a critical path by replacing individual cells with similar ones that improve the speed or power of the circuit.

Example

```
in_place_swap_mode : match_footprint;
```

Values for `in_place_swap_mode` can be `match_footprint` or `no_swapping`.

The `in_place_swap_mode` attribute specifies the criteria used by Design Compiler in cell swapping during in-place optimization. The basic criteria for cell swapping are that:

- The cells must have the same function
- The cells must have the same number of pins with the same pin names

You can define a set of interchangeable cells by using the `cell_footprint` cell attribute, described in "cell_footprint Attribute" on page 7-4." You can then use the `in_place_swap_mode` attribute to indicate how the footprints are used as criteria during in-place optimization.

Use the `report_lib` command in `lc_shell` to print the `in_place_swap_mode` value your technology library is using.

library_features Attribute

The `library_features` attribute lets other Synopsys products use the command features that you specify as attribute values.

The default for the `library_features` attribute is none (no library features are available for use by other Synopsys products).

Syntax

```
library_features (value) ;
```

value

Valid values are `report_delay_calculation`, `report_power_calculation`, `report_noise_calculation`, `report_user_data` and `allow_update_attribute`. The default is none (no library features are available to be used by other Synopsys products).

Example

```
library_features (report_delay_calculation) ;
```

simulation Attribute

When you set the `simulation` attribute to `true`, you permit simulation library files for the Synopsys VHDL Simulator to be generated with liban from the technology library's .db file. See the Synopsys VHDL Simulator manuals for more information about liban.

Example

```
simulation : true;
```

Setting Minimum Cell Requirements

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

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

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

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

Setting CMOS Default Attributes

You can set default pin and timing attribute values for a CMOS technology library in the `library` group definition. You can override defaults with attribute values set at the individual `pin` or `timing` group level.

The following tables list the default attributes that you can define within the `library` group and the attributes that override them.

- Table 6-1 lists the default attributes you can use in all the CMOS models.
- Table 6-2 lists the default attributes you can use in CMOS2 delay models.
- Table 6-2 lists the default attributes you can use in Piecewise linear delay models.
- Table 6-3 lists the default attributes you can use in CMOS linear delay models.

For descriptions of the default attributes, see the “Building Environments” chapter in the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide*.

Table 6-1 CMOS Default Attributes for All Models

Default attribute	Description	Override with
`default_cell_leakage_power`	Default leakage power	`cell_leakage_power`
`default_connection_class`	Default connection class	`connection_class`
`default_fanout_load`	Fanout load of input pins	`fanout_load`
`default_inout_pin_cap`	Capacitance of inout pins	`capacitance`
`default_input_pin_cap`	Capacitance of input pins	`capacitance`
`default_intrinsic_fall`	Intrinsic fall delay of a timing arc	`intrinsic_fall`
`default_intrinsic_rise`	Intrinsic rise delay of a timing arc	`intrinsic_rise`
`default_leakage_power_density`	Default leakage power density	`cell_leakage_power`
`default_max_capacitance`	Maximum capacitance of output pins	`max_capacitance`
`default_max_fanout`	Maximum fanout of all output pins	`max_fanout`
`default_max_transition`	Maximum transition of output pins	`max_transition`
`default_max_utilization`	Maximum limit of utilization	No override available
`default_min_porosity`	Minimum porosity constraint	`set_min_porosity` in dc_shell

Table 6-1 CMOS Default Attributes for All Models (Continued)

Default attribute	Description	Override with
`default_operating_conditions`	Default operating conditions for the library	`operating_conditions`
`default_output_pin_cap`	Capacitance of output pins	`capacitance`
`default_slope_fall`	Fall sensitivity factor of a timing arc	`slope_fall`
`default_slope_rise`	Rise sensitivity factor of a timing arc	`slope_rise`
`default_wire_load`	Wire load	No override available
`default_wire_load_area`	Wire load area	No override available
`default_wire_load_capacitance`	Wire load capacitance	No override available
`default_wire_load_mode`	Wire load mode	`set_wire_load -mode` in dc_shell
`default_wire_load_resistance`	Wire load resistance	No override available
`default_wire_load_selection`	Wire load selection	No override available

Table 6-2 CMOS2 Default Attributes for Piecewise Linear Delay Models

Default attribute	Description	Override with
`default_fall_delay_intercept`	Falling-edge intercept of a timing arc	`fall_delay_intercept`
`default_fall_pin_resistance`	Fall resistance of output pins	`fall_pin_resistance`

Table 6-2 CMOS2 Default Attributes for Piecewise Linear Delay Models (Continued)

Default attribute	Description	Override with
`default_rise_delay_intercept`	Rising-edge intercept of a timing arc	`rise_delay_intercept`
`default_rise_pin_resistance`	Rise resistance of output pins	`rise_pin_resistance`

Table 6-3 CMOS Default Attributes for Linear Delay Models

Default attribute	Description	Override with
`default_inout_pin_fall_res`	Fall resistance of inout pins	`fall_resistance`
`default_inout_pin_rise_res`	Rise resistance of inout pins	`rise_resistance`
`default_output_pin_fall_res`	Fall resistance of output pins	`fall_resistance`
`default_output_pin_rise_res`	Rise resistance of output pins	`rise_resistance`

CMOS library Group Example

Example 6-3 shows `library` group attributes for a CMOS library that contains buses and uses a piecewise linear delay model.

Example 6-3 CMOS library Group Example

```
library(example1) {
   technology (cmos) ;
   delay_model : piecewise_cmos ;
   date : "August 14, 2002" ;
   revision : 2002.05 ;
   comment : "Copyright 1988-2002 XYZ, Inc." ;

   bus_naming_style : "Bus%sPin%d" ;
   piece_type : piece_length ;
   piece_define ( " 0 10 20 " ) ;
}
```

7

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:

Defining cell Groups

A `cell` group defines a single cell in the technology library. A `model` group describes limited hierarchical interconnections. This section discusses the attributes in a `cell` group and a `model` group.

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

- "Defining pin Groups" on page 7-18
- "Defining Bused Pins" on page 7-56
- "Defining Signal Bundles" on page 7-63
- "Defining scaled_cell Groups" on page 7-67

See Chapter 10, "Defining Test Cells," for a test cell with a `test_cell` group.

See Example 7-1 on page 7-13 for an example cell description.

cell Group

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

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

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

When naming cells, remember that names are case-sensitive; cell names AND2, and2, and And2 are all different. Cell names beginning with a number must be enclosed in quotation marks. These rules apply to all user-defined names in Library Compiler.

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

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

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

- `area`
- `bundle` (See "Defining Signal Bundles" on page 7-63.)
- `bus` (See "Defining Bused Pins" on page 7-56.)
- `cell_footprint`
- `clock_gating_integrated_cell`
- `contention_condition`
- `dont_fault`
- `dont_touch`
- `dont_use`
- `handle_negative_constraint`
- `is_clock_gating_cell`
- `map_only`
- `pad_cell`
- `pad_type`
- `pin_equal`
- `pin_opposite`
- `preferred`
- `scaling_factors`
- `use_for_size_only`
- `vhdl_name`

area Attribute

This attribute specifies the cell area.

Example

```
area : 2.0;
```

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

cell_footprint Attribute

This attribute assigns a footprint class to a cell.

Example

```
cell_footprint : 5MIL ;
```

Characters in the string are case-sensitive.

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

See the section on the "in_place_swap_mode Attribute" on page 6-20 for more information.

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.

clock_gating_integrated_cell Attribute

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

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

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

Syntax

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

generic

When you specify an attribute value of generic, Library Compiler determines the actual type of clock gating integrated cell structure by accessing the function specified on the library pin. Library Compiler can determine the actual clock gating structure by accessing the latch groups along with the function attribute specified on the cell.

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

value

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

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

Example

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

Table 7-1 lists some example values for the `clock_gating_integrated_cell` attribute:

Table 7-1 Values for the clock_gating_integrated_cell Attribute

When the value is:	The integrated cell must contain
`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-triggeredregisters. 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.

Table 7-1 Values for the clock_gating_integrated_cell Attribute (Continued)

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

For a complete listing of the values you can enter for the `clock_gating_integrated_cell` attribute, the corresponding circuitry that these values represent, and examples for each value, see Appendix A, "Clock-Gating Integrated Cell Circuits."

For more details about clock-gating integrated cells, see the "Modeling Power and Electromigration" chapter in the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide* and the *Power Compiler Reference Manual*.

Setting Pin Attributes for an Integrated Cell

The clock-gating tool requires that you set the pins of your integrated cells by using the attributes listed in Table 7-2. Setting some of the pin attributes, such as those for test and observability, is optional.

Table 7-2 Pin Attributes for Integrated Clock-Gating Cells

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

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

- "clock_gate_clock_pin Attribute" on page 7-22
- "clock_gate_enable_pin Attribute" on page 7-22

- “clock_gate_obs_pin Attribute” on page 7-23
- “clock_gate_out_pin Attribute” on page 7-23
- “clock_gate_test_pin Attribute” on page 7-23

For more details about the `clock_gating_integrated_cell` attribute and the corresponding pin attributes, see the “Modeling Power and Electromigration” chapter in the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide* and the *Power Compiler Reference Manual*.

Setting Timing for an Integrated Cell

You set both the setup and hold arcs on the enable pin by setting the `clock_gate_enable_pin` attribute for the integrated cell to `true`. The setup and hold arcs for the cell are determined by the edge values you enter for the `clock_gating_integrated_cell` attribute. Table 7-3 lists the edge values and the corresponding setup and hold arcs.

Table 7-3 Values of the clock_gating_integrated_cell Attributes

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

For details about setting timing for an integrated cell, see the “Modeling Power and Electromigration” chapter in the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide*.

contention_condition Attribute

DFT Compiler uses the `contention_condition` attribute to specify the contention conditions for a cell. Contention is a clash of 0 and 1 signals. In certain cells, it can be a forbidden condition and cause circuits to short.

Example

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

dont_fault Attribute

This attribute is used by DFT Compiler. It is a string attribute that you can set on a library cell or pin.

Example

```
dont_fault : sa0;
```

If you set this attribute on a library cell to sa0, instances of the cell in the design are not modeled with stuck-at-0 faults during fault modeling. Similarly, if you set this attribute on a library cell to sa1, instances of the cell in the design are not modeled with stuck-at-1 faults. Setting this attribute on a cell to sa01 means that the cell is not modeled with either stuck-at-0 or stuck-at-1 faults. The same values are allowed on pins and mean that the pin is not modeled with the specified stuck-at fault.

dont_touch Attribute

If you do not want a cell removed during optimization, use the `dont_touch` attribute. When you set this attribute to `true`, all instances of the cell must remain in the network.

Example

```
dont_touch : true;
```

When it encounters the cell during optimization, Design Compiler works around the cell and does not replace it. You can apply the `dont_touch` attribute to a special clock-distribution cell.

In addition to defining `dont_touch` in your technology library, you can also apply the `dont_touch` attribute to a cell with the `set_dont_touch` command from dc_shell. You cannot remove a cell with the `dont_touch` attribute from dc_shell.

dont_use Attribute

If you do not want a cell added to a design during optimization, set this attribute to `true`. For example, you could specify `dont_use` for an I/O cell that you do not want Design Compiler to put into the core of your design.

Example

```
dont_use : true;
```

In addition to defining `dont_use` in your technology library, you can also apply the `dont_use` attribute to a cell with the `set_dont_use` command from dc_shell.

handle_negative_constraint Attribute

Use this attribute during generation of VITAL models to indicate whether the cell needs negative constraint handling. It is an optional Boolean attribute for timing constraints in a `cell` group.

Example

```
handle_negative_constraint : true ;
```

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

is_clock_gating_cell Attribute

The `is_clock_gating_cell` attribute identifies cells that are used for clock gating. Design Compiler uses this attribute when it compiles a design containing gated clocks that are introduced by Power Compiler.

Set this attribute only on 2-input AND, NAND, OR, and NOR gates; inverters; buffers; and 2-input D latches. Valid values for this attribute are `true` and `false`.

Example

```
is_clock_gating_cell : true;
```

To use a cell exclusively during the mapping of clock-gating circuits, set the `is_clock_gating_cell` attribute to `true` and the `dont_touch` attribute to `true`.

See “Defining pin Groups” on page 7-18 for information about designating clock and enable ports on clock gates. For additional information about clock gating, see the *Power Compiler User Guide*.

is_memory_cell Attribute

The `is_memory_cell` simple Boolean attribute identifies whether a cell is a memory cell. The valid values are `true` and `false`. Set the `is_memory_cell` attribute to `true` at the cell level to specify that the cell is a memory cell.

Example

```
is_memory_cell : true;
```

map_only Attribute

If this attribute has the value `true`, the cell is excluded from logic-level optimization during compilation.

Example

```
map_only : true;
```

In addition to defining `map_only` in a `cell` group, you can also apply the `map_only` attribute to a cell by using the `set_map_only` command from `dc_shell`.

pad_cell Attribute

The `pad_cell` attribute in a `cell` group identifies the cell as a pad. Design Compiler filters out pads during core optimization and treats them differently during technology translation.

Example

```
pad_cell : true ;
```

If the `pad_cell` attribute is included in a cell definition, at least one pin in the cell must have an `is_pad` attribute. See “is_pad Attribute” on page 7-51.

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

```
auxiliary_pad_cell : true ;
```

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

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

pad_type Attribute

This attribute identifies a pad cell or auxiliary pad cell that requires special treatment. The only type of pad cell supported is clock, which identifies a pad cell as a clock driver.

Example

```
pad_type : clock ;
```

pin_equal Attribute

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

Example

```
pin_equal : ("Y Z") ;
```

Design Compiler automatically determines the equivalent pins for cells that have `function` attributes. See "function Attribute" on page 7-45 for more information.

Note:
: Use the `pin_equal` attribute only in cells without function information or when you want to define required input values.

pin_opposite Attribute

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

Example

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

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

Design Compiler automatically determines the opposite pins for cells that have `function` attributes. See "function Attribute" on page 7-45 for more information.

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

preferred Attribute

Setting this attribute to value `true` indicates that the cell is the preferred replacement during the gate-mapping phase of optimization.

Example

```
preferred : true ;
```

Design Compiler chooses cells with the `preferred` attribute over other cells with the same function if the standard cell offers no optimization advantage.

You can apply the `preferred` attribute to a cell with preferred timing or area attributes. For example, in a set of 2-input NAND gates, you might want to use gates with higher drive strengths wherever possible. This practice is useful primarily in design translation.

scaling_factors Attribute

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

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.

For additional information on the `scaling_factors` group, see the “Scaling Factors for Individual Cells” section in the “Building Environments” chapter in the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide*.

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

use_for_size_only Attribute

You use this attribute to specify the criteria for sizing optimization. You set this attribute on a cell at the cell level. When this attribute is set on a cell, Design Compiler does not map to that cell. As a result, a designer has to instantiate those cells.

Example

```
library(lib1){
   cell(cell1){
      area : 14 ;
      use_for_size_only : true ;
      pin(A){
      ...
      }
      ...
   }
}
```

vhdl_name Attribute

This attribute defines valid VHDL object names.

Example

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

The `vhdl_name` attribute can be added to technology library `cell` and `pin` groups. This attribute lets library developers define legal VHDL object names explicitly when they translate from database format. Some .db object names can violate the more restrictive VHDL rules for identifiers. The `vhdl_name` attribute resolves conflicts of invalid object names when translating from .db to VHDL. Names in the library should be VHDL-compliant.

type Group

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

Example

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

cell Group Example

Example 7-1 shows cell definitions that include some of the CMOS cell attributes described in this section.

Example 7-1 cell Group Example

```
library (cell_example){
   date : "August 14, 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 */
```

mode_definition Group

A `mode_definition` group declares a `mode` group that contains several timing mode values. PrimeTime can enable each timing arc, based on the current mode of the design, which results in different timing for different modes. You can optionally put a condition on a `mode` value. When the condition is true, the `mode` group takes that value.

Syntax

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

Group Statement

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

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

mode_value Group

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

Syntax

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

Simple Attributes

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

when Simple Attribute

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

Syntax

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

Example

```
when: !R;
```

sdf_cond Simple Attribute

The `sdf_cond` attribute supports Standard Delay Format (SDF) file generation and condition matching during back-annotation; however, PrimeTime does not use this attribute for back-annotation.

Syntax

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

Example

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

Example 7-2 shows a `mode_definition` description.

Example 7-2 mode_definition Description

```
cell(example_cell) {
...
   mode_definition(rw) {
      mode_value(read) {
         when : "R";
         sdf_cond : "R == 1";
      }
      mode_value(write) {
         when : "!R";
         sdf_cond : "R == 0";
      }
   }
}
```

Defining Cell Routability

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

routing_track Group

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

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.

Example

```
tracks : 2 ;
```

total_track_area Attribute

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

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 7-3 shows a library that contains routability information.

Example 7-3 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.

Defining pin Groups

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

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

For more information about `timing` groups, see the "Timing Arcs" chapter in the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide*.

For more information about the `internal_power` group, see the "Modeling Power and Electromigration" chapter in the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide*.

Example 7-11 on page 7-56 shows a `pin` group specification.

pin Group

You can define a `pin` group within a `cell`, `test_cell`, `scaled_cell`, `model`, or `bus` group. You can also define a `pin` group within a `bundle` group, however, Design Compiler does not accept a `pin` group in the `bundle` group.

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

See “Defining Bused Pins” on page 7-56 for descriptions of `bus` groups. See “Defining Signal Bundles” on page 7-63 for descriptions of `bundle` groups.

The `pin` groups are also valid within `test_cell` groups. They have different requirements from `pin` groups in `cell`, `bus`, or `bundle` groups. See “Pins in the test_cell Group” on page 10-3 for specific information and restrictions on describing test pins.

All pin names within a single `cell`, `bus`, or `bundle` group must be unique. As with all names in Library Compiler, pin names are case-sensitive: pins named `A` and `a` are different pins.

Pin names beginning with a number must be enclosed in quotation marks; for the exceptions to this rule, see the “function Attribute” on page 7-45 and the information on the `related_pin` attribute in the “Timing Arcs” chapter in the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide*.

Also, pin names that include a special character, such as a greater-than symbol (>), or a lesser-than symbol (<), must be enclosed in quotation marks.

Note:

Although netlists can be written out in either implicit or explicit formats, Library Compiler supports only the implicit format.

Implicit, or order-based, netlists attach nets to components according to the order in which they are defined in the netlist. The order in which the pins are listed in the descriptions of the cells is the order in which they will be referenced when writing (and possibly reading) a netlist. Therefore, the Library Compiler cell descriptions must have output pins listed before input pins.

Explicit netlists include the pin name and the net connected to the pin, and the order is not important. Library Compiler does not support this format.

See “Netlisting Considerations” on page 7-71 for information about pin ordering within `cell` groups.

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

Do not confuse defining multiple pins in a single `pin` group with defining a bus or bundle. See “Defining Bused Pins” on page 7-56 and “Defining Signal Bundles” on page 7-63.

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 ;
   }
}
```

General pin Group Attributes

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

- `capacitance`
- `clock_gate_clock_pin`
- `clock_gate_enable_pin`
- `clock_gate_obs_pin`
- `clock_gate_out_pin`
- `clock_gate_test_pin`
- `complementary_pin`
- `connection_class`
- `direction`

- `dont_fault`
- `driver_type`
- `fall_capacitance`
- `fault_model`
- `inverted_output`
- `is_analog`
- `pin_func_type`
- `rise_capacitance`
- `steady_state_resistance`
- `test_output_only`

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

capacitance Attribute

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

Example

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

```
cell (AND) {
   area : 3 ;
   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. Design Compiler uses the `clock_gate_clock_pin` attribute when it compiles a design containing gated clocks that were introduced by Power Compiler.

Valid values for this attribute are true and false.

Example

```
clock_gate_clock_pin : true;
```

See "clock_gating_integrated_cell Attribute" on page 7-4; Appendix A, "Clock-Gating Integrated Cell Circuits"; and the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide* for more information about identifying pins on integrated clock-gating cells. For additional information about integrated clock-gating, see the *Power Compiler User Guide*.

clock_gate_enable_pin Attribute

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

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

Valid values for this attribute are `true` and `false`. A `true` value labels the input port pin connected to an enable signal for nonintegrated and integrated clock-gating cells. A `false` value labels the input port pin connected to an enable signal as *not* for nonintegrated and integrated clock-gating cells.

Example

```
clock_gate_enable_pin : true;
```

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

See "clock_gating_integrated_cell Attribute" on page 7-4; Appendix A, "Clock-Gating Integrated Cell Circuits"; and the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide* for more information about identifying pins on integrated clock-gating cells. For additional information about integrated clock gating, see the *Power Compiler User Guide*.

clock_gate_obs_pin Attribute

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

Valid values for this attribute are `true` and `false`. A `true` value labels the pin as an observability pin. A `false` value labels the pin as not an observability pin.

Example

```
clock_gate_obs_pin : true;
```

See "clock_gating_integrated_cell Attribute" on page 7-4; Appendix A, "Clock-Gating Integrated Cell Circuits"; and the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide* for more information about identifying pins on integrated clock-gating cells. For additional information about integrated clock gating, see the *Power Compiler User Guide*.

clock_gate_out_pin Attribute

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

Valid values for this attribute are `true` and `false`. A `true` value labels the pin as an out (`enable_clock`) pin. A `false` value labels the pin as *not* an out pin.

Example

```
clock_gate_out_pin : true;
```

See "clock_gating_integrated_cell Attribute" on page 7-4; Appendix A, "Clock-Gating Integrated Cell Circuits"; and the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide* for more information about identifying pins on integrated clock-gating cells. For additional information about integrated clock gating, see the *Power Compiler User Guide*.

clock_gate_test_pin Attribute

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

Valid values for this attribute are `true` and `false`. A `true` value labels the pin as a test (`test_mode` or `scan_enable`) pin. A `false` value labels the pin as not a test pin.

Example

```
clock_gate_test_pin : true;
```

See "clock_gating_integrated_cell Attribute" on page 7-4; Appendix A, "Clock-Gating Integrated Cell Circuits"; and the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide* for more information about identifying pins on integrated clock-gating cells. For additional information about integrated clock gating, see the *Power Compiler User Guide*.

complementary_pin Simple Attribute

The `complementary_pin` attribute, which works only with DFT Compiler, supports differential I/O. Differential I/O assumes the following:

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

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

Library Compiler automatically generates a connection class differential without your having to set it with the `connection_class` attribute. For information about the `connection_class` attribute, see "connection_class Simple Attribute."

Syntax

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

string

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

For details on the `fault_model` attribute used to define the value when both the complementary pin and the pin that it complements are driven to the same value, see "fault_model Simple Attribute" on page 7-32.

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.

Design Compiler recognizes two special connection classes: universal and default.

- The universal connection class matches all other connection classes, so a pin with this class can be connected to another pin.
- Design Compiler assigns the default connection class to all pins that do not have a connection class assigned to them. A default pin can be connected only to another default pin.

You can define a default connection class with the `default_connection_class` attribute in the `library` group, as described in the “Library-Level Default Attributes” section in the “Building Environments” chapter in the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide*. If you do not define a default connection class for your library, Design Compiler assigns the value to the `default_connection_class` attribute.

Nets with multiple drivers, such as buses or wired logic, follow the same connection rules as single-driver nets. All pins must have the same connection class.

Example 7-4 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 7-5 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 7-4 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 7-5 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

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

Example

```
direction : output;
```

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

dont_fault Attribute

DFT Compiler uses the `dont_fault` attribute. It is a string attribute that you can set on a library cell or pin.

Example

```
dont_fault : sa0;
```

If you set this attribute on a library cell to sa0, instances of the cell in the design will not be modeled with stuck-at-0 faults during fault modeling. Similarly, if you set this attribute on a library cell to sa1, instances of the cell in the design will not be modeled with stuck-at-1 faults. Setting this attribute on a cell to sa01 means that the cell will not be modeled with either stuck-at-0 or stuck-at-1 faults. The same values are allowed on pins and mean that the pins will not be modeled with the specified stuck-at fault.

driver_type Attribute

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

The `driver_type` attribute tells the DFT Compiler and the VHDL library generator to use a special pin-driving configuration for the pin during simulation. A pin without this attribute has normal driving capability by default.

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

pull_up

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

pull_down

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

Note:
To alert you that the output driving the input could be affected by the pull-up or pull-down driver, Library Compiler issues a warning whenever an input pin has a pull-up or pull-down driver.

bus_hold

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

open_drain

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

open_source

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

resistive

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

resistive_0

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

resistive_1

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

Except for inout pins, each pin can have only one `driver_type` attribute. Library Compiler ignores multiple statements and uses the last statement.

Inout pins can have two driver types, one for input and one for output. The only valid combinations are pull_up/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 7-4 for the signal mapping and pin types for the different driver types.

Table 7-4 Driver Types

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

Signal Mapping:

0 and 1represent strong logic 0 and logic 1 values

L represents a weak logic 0 value

H represents a weak logic 1 value

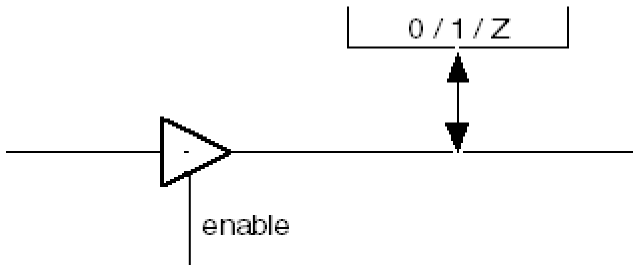
Z represents high impedance

S represents the previous state

Interpreting Bus Holder Driver Type

Figure 7-1 illustrates the 01Z to 01S signal mapping for bus holders.

Figure 7-1 Interpreting bus_hold Driver Type

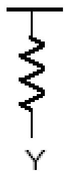


For bus_holder driver types, a three-state buffer output value of 0 changes the bus value to 0. Similarly, a three-state buffer output value of 1 changes the bus value to 1. However, when the output of the three-state buffer is Z, the bus holds its previous value (S), which can be 0, 1, or Z. In other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

Modeling Pull-Up and Pull-Down Cells

Figure 7-2 shows a pull-up resistor cell.

Figure 7-2 Pull-Up Resistor of a Cell



Example 7-6 is the description of the pull-up resistor cell in Figure 7-2. This cell has a pull-up driver type and no input. Because it is functionless, Library Compiler labels it a black box. This model requires special recognition from the tools that use it. This cell is a pad, but you can omit the pad information for an internal pull-up cell.

Example 7-6 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 7-7 describes an output pin with a pull-up resistor and the bidirectional pin on a bus holder cell.

Example 7-7 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 Simple Attribute

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

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

Note:
If Library Compiler does not find a `capacitance` attribute for an input or inout pin, it generates a warning message and uses the larger of the value settings for `fall_capacitance` and `rise_capacitance`. If `fall_capacitance` and `rise_capacitance` have not been set for the input pin, Library Compiler uses the value to which the `default_input_pin_cap` attribute is set, and for the inout pin, it uses the value to which the `default_inout_pin_cap` attribute is set.

Syntax

```
fall_capacitance : float ;
```

float

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

Example

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

```
cell (AND) {
   area : 3 ;
   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.

The `fault_model` attribute, which is used only by DFT Compiler, is optional. If you enter a `fault_model` attribute, you must designate the inverted pin associated with the noninverting pin, using the `complementary_pin` attribute.

For details on the `complementary_pin` attribute, see "complementary_pin Simple Attribute" on page 7-24.

Syntax

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

two-value string

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

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

Example

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

Table 7-5 shows how DFT Compiler interprets the complementary pin values for this example:

Table 7-5 Interpretation of Pin Values

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

inverted_output Attribute

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

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

Example

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

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

is_analog Attribute

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

Syntax

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

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

Example

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

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

pin_func_type Attribute

This attribute describes the functions of a pin.

Example

```
pin_func_type : clock_enable;
```

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

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

rise_capacitance Simple Attribute

This attribute defines the load for an input and inout pin when its signal is rising.

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

Note:

If Library Compiler does not find a `capacitance` attribute for an input or inout pin, it generates a warning message and uses the larger of the value settings for `fall_capacitance` and `rise_capacitance`. If `fall_capacitance` and `rise_capacitance` have not been set for the input pin, Library Compiler uses the value to which the `default_input_pin_cap` attribute is set, and for the inout pin, it uses the value to which the `default_inout_pin_cap` attribute is set.

Syntax

```
rise_capacitance : float ;
```

float

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

Example

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

```
cell (AND) {
   area : 3 ;
   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 ;
}
```

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.

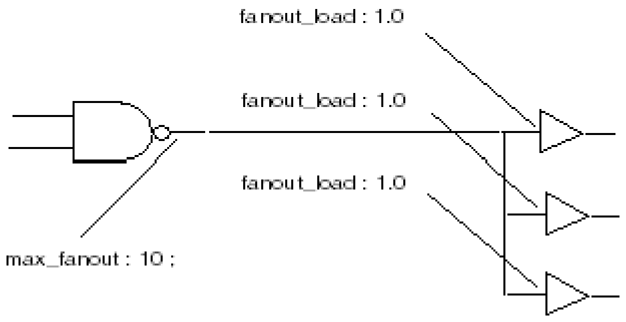
Example

```
fanout_load : 1.0;
```

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

Figure 7-3 illustrates `max_fanout` and `fanout_load` attributes for a cell.

Figure 7-3 Fanout Attributes



max_fanout Attribute

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

Example

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

Some designs have limitations on input load that an output pin can drive regardless of any loading contributed by interconnect metal layers. To limit the number of inputs on the same net driven by an output pin, define a `max_fanout` value for each output pin and a `fanout_load` on each input pin in a cell. (See Figure 7-3.)

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

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

In Figure 7-3, Design Compiler adds all the `fanout_load` values of the input to equal 3.0. Then Design Compiler compares the sum of the `fanout_load` on all inputs with the `max_fanout` constraint of the driving output, which is 10. In this case, no design rule violation occurs. If a design rule violation occurs, Design Compiler might replace the driving cell with a cell that has a higher `max_fanout` value.

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

- The value you assign for the pin always overrides the library default.
- When you assign a `max_fanout` value at the command line, Design Compiler uses the smallest value. If the pin value is smaller, Design Compiler ignores the command-line value.

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

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

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

- The fanout values are used only for design rule checking.
- The capacitance values are used by the Design Compiler timing verifier in delay calculations.

min_fanout Attribute

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

Example

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

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

max_transition Attribute

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

Example

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

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

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

With an input pin, `max_transition` indicates that the pin cannot be connected to a net that has a transition time greater than the defined limit. Design Compiler attempts to resolve `max_transition` violations, possibly at the expense of other design constraints.

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

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

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

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

max_capacitance Attribute

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

Example

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

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

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

min_capacitance Attribute

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

Example

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

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

cell_degradation Group

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

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

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

There are two ways to model cell degradation,

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

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” on page 7-18. The number of floating-point numbers in `index_1` determines the size of the table dimension.

This is an example of a cell degradation template.

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

See the “Timing Arcs” chapter in the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide* for more information about lookup table templates.

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

   The following example shows the `cell_degradation` group:

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

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

   - combinational
   - three_state_enable
   - rising_edge
   - falling_edge

- preset
- clear

Assigning Values to Lookup Tables

These are the rules for specifying cell degradation lookup tables:

- You can overwrite the `index_1` value in the lookup table template with the optional `index_1` attribute in the `cell_degradation` group, but the overwrite must occur before the actual values are defined.
- The number of floating-point numbers in the `value` attribute must be the same as the number in the corresponding `index_1` in the table template.
- Each entry in the values attribute uses the `capacitive_load_unit` library attribute as its unit and must be 0.0 or greater.
- When a timing arc is state-dependent and the cell has cell degradation defined for all valid timing types, you must create a cell degradation table for each state-dependent timing arc.

Describing Clocks

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

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

clock Attribute

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

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

Example

```
clock : true ;
```

min_period Attribute

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

Example

```
min_period : 26.0;
```

If a `min_period` attribute is placed on a pin that is not a clock pin, Library Compiler ignores the attribute.

min_pulse_width_high and min_pulse_width_low Attributes

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

The VHDL library generator uses the `min_pulse_width` attributes for simulation.

Note:
Design Compiler does not support minimum pulse width as a constraint during synthesis.

Example

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

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

Yield Modeling

An example of modeling yield information is as follows.

For syntax details, see the *Library Compiler Technology and Symbol Libraries Reference Manual.*

```
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 7-6 lists the Boolean operators that are valid in a `function` statement.

Table 7-6 Valid Boolean Operators

Operator	Description
'	Invert previous expression
!	Invert following expression
^	Logical XOR
*	Logical AND
&	Logical AND
space	Logical AND
+	Logical OR
|	Logical OR
1	Signal tied to logic 1
0	Signal tied to logic 0

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

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

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

Grouped Pins in function Statements

Grouped pins can be used as variables in a `function` statement; see "Defining Bused Pins" on page 7-56 and "Defining Signal Bundles" on page 7-63. In `function` statements that use bus or bundle names, all the variables in the statements must be either a single pin or buses or bundles of the same width.

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

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

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

the function for Z[0] is interpreted as

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

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

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

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

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

The function for Z[0] is interpreted as

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

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

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

three_state Attribute

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

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

In Example 7-8, output pin Z is a three-state output pin. When input pin E (enable) goes low, pin Z goes to a high-impedance state. The value of the `three_state` attribute is, therefore, E'.

Example 7-8 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.

Note:
Only DFT Compiler and Formality use the `x_function` attribute.

The `three_state`, `function`, and `x_function` attributes are defined for output and inout pins and can have shared input. You can assign `three_state`, `function`, and `x_function` to be the function of the same input pins. When these functions have shared input, however, the cell will not be inserted by Design Compiler. It must be inserted manually.

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

1. x_function
2. three_state
3. function

Example

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

state_function Attribute

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

Example

```
state_function : 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 might also have the optional `input_map` attribute.

Example

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

Describing Sequential Devices

To describe an input pin of a flip-flop or latch, use the `prefer_tied` attribute.

prefer_tied Attribute

When it is processing each sequentially modeled component, Library Compiler generates the combinational logic necessary to implement the D flip-flop functionality that uses that component. Because D flip-flop functionality can be implemented in more than one way by use of a complex flip-flop, the `prefer_tied` pin attribute lets you choose the logic that results in tying the pin to a specified logic value.

Example

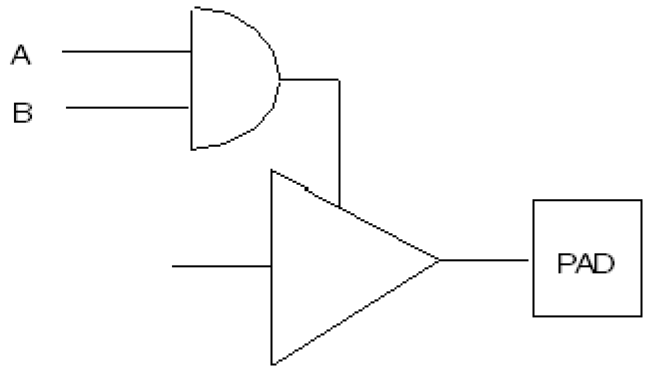
```
prefer_tied : "1";
```

Library Compiler honors as many `prefer_tied` attributes as possible while it is still able to implement D flip-flop functionality. However, it cannot honor all of them. For example, if the library developer specifies `prefer_tied : 0` on all the inputs, Library Compiler honors as many as possible and ignores the rest. If Library Compiler ignores any inputs, it issues a message stating so when reading in the `.lib` file. You typically use `prefer_tied` on multiplexed flip-flop input.

Note:
The `prefer_tied` attribute directs Design Compiler in implementing a D flip-flip. It is important to define all complex register elements with the `prefer_tied` attribute. For sequential optimization, Design Compiler ignores this attribute.

The `prefer_tied` attribute can also be used to indicate which pins are tied to fixed logic during three_state degeneration for three-state pad cells. For example, Library Compiler tries to degenerate the input pad shown in Figure 7-4 into a simple three-state output pad by tying either pin A or pin B to logic 1. Design Compiler uses the degenerated pad cell for pad-related operations.

Figure 7-4 Output Pad Tied to Fixed Logic



To control the degeneration process to prefer one version over the other, place a `prefer_tied` attribute on the pin. Library Compiler ties that pin to the logic specified in the attribute.

If Library Compiler cannot degenerate the pad cell according to the specified `prefer_tied` attribute but can degenerate the cell by tying the same pin to the opposite logic, it does so and issues a warning.

In this example, to tie pin B rather than pin A to logic 1 during degeneration, specify a `prefer_tied` attribute in the `pin` group of pin B.

```
pin (B) {
      prefer_tied : "1";
      ...
}
```

Describing Pad Pins

To describe a pad pin, use these `pin` group attributes:

- `is_pad`
- `multicell_pad_pin`
- `input_voltage`
- `output_voltage`
- `pulling_resistance`
- `pulling_current`
- `hysteresis`
- `drive_current`
- `slew_control`
- `rise_current_slope_before_threshold`
- `rise_current_slope_after_threshold`
- `fall_current_slope_before_threshold`
- `fall_current_slope_after_threshold`
- `rise_time_before_threshold`
- `rise_time_after_threshold`
- `fall_time_before_threshold`
- `fall_time_after_threshold`

is_pad Attribute

After you identify a cell as a pad cell, you must indicate which pin represents the pad. The valid values are `true` and `false`. The `is_pad` attribute must be used on at least one pin with a `pad_cell` attribute. You can also specify the `is_pad` attribute on a PG pin. If the `pad_cell` attribute is specified on a I/O cell, you must set the `is_pad` attribute to `true` in either a `pg_pin` group or on a signal pin for that cell. Otherwise, Library Compiler issues an error message. If the cell-level `pad_cell` attribute is specified on a I/O cell and there is no signal pin or PG pin in the pad cell, Library Complier issues a warning message. For information about the `pad_cell` attribute, see “pad_cell Attribute” on page 7-10.

Examples

```
cell(INBUF) {
   ...
   pin(PAD) {
      direction : input;
      is_pad : true;
   }
}
```

In the following example, the `is_pad` attribute is specified on a PG pin:

```
cell (POWER_PAD_CELL) {
  …
  pad_cell : true ;
  pg_pin (my_pg_pin) {
    is_pad : true ;
    …
  }
  pin (my_pin) {
    …
  }
}
```

multicell_pad_pin Attribute

Use this attribute to indicate which pin on a cell should be connected to another cell to create the correct configuration. For example, some technologies implement pads with multiple cells by connecting a pad cell and a pad driver.

Example

```
multicell_pad_pin : true;
```

Use this attribute for all pins on a pad or auxiliary pad cell that are connected to another cell. Design Compiler connects the correct multicell pad pins according to the `connection_class` attribute.

Voltage Attributes

These are voltage attributes:

- `input_voltage` is used for an input pin definition.
- `output_voltage` is used for an output pin definition.

You can define a special set of voltage thresholds in the `library` group with the `input_voltage` or `output_voltage` attribute. You can then apply the default voltage ranges in the group to selected cells by using the `input_voltage` or `output_voltage` attribute in the pin definition.

Example 7-9 selects a voltage range for the input pin of a pad cell.

Example 7-9 Voltage Range for Input Pin

```
input_voltage(CMOS_SCHMITT) {
   ...
}
cell(INBUF) {
   ...
   pin(PAD) {
      input_voltage : CMOS_SCHMITT ;
   ...
   }
}
```

Pull-Up and Pull-Down Strength Attributes

The pull-up and pull-down strength attributes are

- `pulling_resistance`
- `pulling_current`

These attributes let you define the resistance or current-drawing capability of a pull-up or pull-down device on a pin. These attributes can be used for pins with a driver type of `pull_up` or `pull_down`.

Example

```
driver_type : pull_up ;
pulling_resistance : 1000 ;
```

Assigning a negative value to the `pulling_current` attribute produces a current flow in the opposite or negative direction. Assigning a negative value to the `pulling_resistance` attribute produces a warning and converts that value to 0.

hysteresis Attribute

Use the `hysteresis` attribute on an input pad when you anticipate a long transition time or when you expect the pad to be driven by a particularly noisy line.

Example

```
hysteresis : true;
```

This attribute allows the pad to accommodate longer transition times, which are more subject to noise problems. A pad with this option has wider threshold tolerances, so voltage spikes on the input do not propagate to the core cells. Using the `hysteresis` attribute cuts down on power consumption, because the input signals change only after the threshold is exceeded.

The derating factors of pads with `hysteresis` can sometimes be different from those of core cells. You might want to define a `scaled_cell` group for the pads to account for this difference.

Drive-Current Strength Attribute

Most output and bidirectional pad cells are characterized with an approximate drive-current capability. You select the pad with the lowest drive current that meets your alternating current's timing constraints.

Example

```
drive_current : 5.0;
```

Each library cell can have only one `drive_current` attribute. You might need to create a different pad cell for each drive-current possibility in your library. Output and inout pads must have the `drive_current` attribute assigned.

Slew-Rate Control Attributes

Slew-rate control limits peak noise by smoothing out fast output transitions, thus decreasing the possibility of a momentary disruption in the power or ground planes.

Library Compiler lets you describe two levels of slew-rate control, using the `slew_control` attribute for coarse tuning and the threshold attributes for fine tuning.

slew_control Attribute

The more slew-rate control you apply, the slower the output transitions are. The `slew_control` attribute provides increasing levels of slew-rate control to slow down the transition rate.

Threshold Attributes

You can define slew-rate control in terms of the current versus time characteristics of the output pad (dI/dT). The syntax is

```
threshold_attribute : value ;
```

The *value* is a floating-point number in the units specified by `current_unit` for current-related attributes and `time_unit` for time-related attributes.

The eight threshold attributes are

rise_current_slope_before_threshold

This value represents a linear approximation of the change in current with respect to time from the beginning of the rising transition to the threshold point.

rise_current_slope_after_threshold

This value represents a linear approximation of the change in current with respect to time from the point at which the rising transition reaches the threshold to the end of the transition.

fall_current_slope_before_threshold

This value represents a linear approximation of the change in current with respect to time from the beginning of the falling transition to the threshold point.

fall_current_slope_after_threshold

This value represents a linear approximation of the change in current with respect to time from the point at which the falling transition reaches the threshold to the end of the transition.

rise_time_before_threshold

This value gives the time interval from the beginning of the rising transition to the point at which the threshold is reached.

rise_time_after_threshold

This value gives the time interval from the threshold point of the rising transition to the end of the transition.

fall_time_before_threshold

This value gives the time interval from the beginning of the falling transition to the point at which the threshold is reached.

fall_time_after_threshold

This value gives the time interval from the threshold point of the falling transition to the end of the transition.

Example 7-10 shows the slew-rate control attributes on an output pad pin.

Example 7-10 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;
   ...
}
```

CMOS pin Group Example

Example 7-11 shows pin attributes in a CMOS library.

Example 7-11 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";
      }
   }
}
```

Defining Bused Pins

To define bused pins, use these groups:

- `type` group
- `bus` group

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

type Group

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

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

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

name

Identifies the bus type.

A `type` group can one of the following:

base_type

Only the array base type is supported.

data_type

Only the bit data type is supported.

bit_width

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

bit_from

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

bit_to

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

downto

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

Example 7-12 illustrates a `type` group statement.

Example 7-12 type Group Statement

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

It is not necessary to use all the `type` group attributes. For example, the `type` group statements in Example 7-13 are both valid descriptions of BUS4 in Example 7-12.

Example 7-13 Alternative type Group Statements

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

Because Library Compiler checks the attributes you use for consistency, using all of them is a good way to verify your description.

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

bus Group

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

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

A `bus` group contains the following elements:

- `bus_type` attribute
- `pin` groups

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

bus_type Attribute

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

Note:
The bus type name must exist in a type group.

Syntax

```
bus_type : name ;
```

Pin Attributes and Groups

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

All pin attributes are valid inside bus and pin groups. See "General pin Group Attributes" on page 7-20 for a description of pin attributes. The direction attribute value of all bus members must be the same.

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

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

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

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

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

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

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

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

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

Note:
Include only the base name of a bused pin as the name of the symbol library pin. This naming method allows one symbol to represent cells of the same type but with variable-width input pins.

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

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

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

Example Bus Description

Example 7-14 is a complete bus description that includes `type` and `bus` groups. It also shows the use of bus variables in `function`, `related_pin`, `pin_opposite`, and `pin_equal` attributes.

Example 7-14 Bus Description
```
library (ExamBus) {
   date : "May 14, 2002";
   revision : 2002.05;
   bus_naming_style :"%s[%d]";/* Optional; this is the
                         default */
   type (bus4) {
      base_type : array;/* Required */
      data_type : bit;/* Required if base_type is array */
      bit_width : 4;/* Optional; default is 1 */
      bit_from : 0;/* Optional MSB; defaults to 0 */
      bit_to : 3;/* Optional LSB; defaults to 0 */
      downto : false;/* Optional; defaults to false */
   }
   cell (bused_cell) {
      area : 10;
      bus (A) {
```

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

```
            capacitance : 1;
            pin (A[0:3]) {
               capacitance : 2;
            }
            pin (A[3]) {
               capacitance : 2.5;
            }
         }
         bus (B) {
            bus_type : bus4;
            direction : input;
            capacitance : 2;
         }
         pin (E) {
            direction : input ;
            capacitance 2 ;
         }
         bus(X) {
            bus_type : bus4;
            direction : output;
            capacitance : 1;
            pin (X[0:3]) {
               function : "A & B'";
               timing() {
                  related_pin : "A B";
                  /* A[0] and B[0] are related to X[0],
                  A[1] and B[1] are related to X[1], etc. */
               }
            }
         }
      }
```

Defining Signal Bundles

You need certain attributes to define a bundle. A bundle groups several pins that have similar timing or functionality. The bundle is used only within Library Compiler; Design Compiler cannot refer to a bundle in a netlist. Bundles are used for multibit cells such as multibit latch, multibit flip-flop, and multibit AND gate.

bundle Group

You define a `bundle` group in a `cell` group, as shown:

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

A `bundle` group contains the following elements:

members attribute

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

pin attributes

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

members Attribute

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

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

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

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

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

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

pin Attributes

For information about pin attributes, see “General pin Group Attributes” on page 7-20.

Example 7-15 shows a `bundle` group in a multibit latch.

Example 7-15 Multibit Latch With Signal Bundles

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

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

Defining Layout-Related Multibit Attributes

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

The `single_bit_degenerate` attribute is for use on multibit bundle or bus cells that are black boxes. The value of this attribute is the name of a single-bit library cell, which Design Compiler uses as a link to associate the multibit cell with the single-bit cell in the library.

Example 7-16 shows multibit library cells with the `single_bit_degenerate` attribute.

Example 7-16 Multibit Cells With single_bit_degenerate Attribute

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

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

Library Compiler verifies that all the following conditions exist for cells in a library containing black box cells and using the `single_bit_degenerate` attribute. If any one of the conditions does not exist, the `single_bit_degenerate` attribute is removed from the cell.

- For any `single_bit_degenerate` attribute on a multibit black box, a single-bit cell by that name must exist in the library.

- For multibit black boxes with the `single_bit_degenerate` attribute, all buses or bundles in a cell must have the same width.
- For multibit black boxes with the `single_bit_degenerate` attribute, the unbused pins, buses, and bundles must match the ports of the single-bit cell, and vice versa.

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

Defining 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. If the default scaling is sufficient for all but a few parameters, you can define only those parameters in the scaled cell and let Design Compiler use and scale the nominal attributes for all the others.

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 might increase processing and load times.

Note:
Library Compiler does not support `scaled_cell` groups used in combination with multiple timing arcs between two pins.

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" ;
         }
      }
   }
}
```

Defining Multiplexers

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

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

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

Library Requirements

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

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

  `C={C_1, C_2, ..., 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' &
  ```

  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, if 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)";
  ... ...
 }
}
```

Defining Decoupling Capacitor Cells, Filler Cells, and Tap Cells

Decoupling capacitor cells, or *decap cells*, are cells that have a capacitor placed between the power rail and the ground rail to overcome dynamic voltage drop; filler cells are used to connect the gaps between the cells after placement; and tap cells are physical-only cells that have power and ground pins and do not have signal pins. Tap cells are well-tied cells that bias the silicon infrastructure of n-wells or p-wells. Library Compiler provides cell-level attributes that identify decoupling capacitor cells, filler cells, and tap cells in libraries.

Syntax

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

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

Cell-Level Attributes

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

is_decap_cell

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

is_filler_cell

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

is_tap_cell

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

Library Checking Rules

Library Compiler performs the following checks. If the conditions are not met, Library Compiler issues an error message.

- Only one of the following attributes can be set to true in a given cell: `is_decap_cell`, `is_filler_cell`, `is_tap_cell`
- Tap cells must have power and ground pins only.

Netlisting Considerations

Certain netlist formats require specific pin ordering and specific case (uppercase or lowercase) for writing out a correct netlist.

The case of the cell name might be important for EDIF and NDL (LSI netlist) because the originating or destination EDA system might expect either lowercase or uppercase cell names and might not be able to do case changes automatically.

If order-based netlists are used with the library, the order in which the `pin` groups appear within a `cell` group must match the order in which the pins are specified in the netlist.

Note:

Although netlists can be written out in either implicit or explicit formats, Library Compiler supports only the implicit format.

Implicit, or order-based, netlists attach nets to components according to the order in which they are defined in the netlist. The order in which the pins are listed in the descriptions of the cells is the order in which they will be referenced when writing (and possibly reading) a netlist. Therefore, the Library Compiler cell descriptions must also have output pins listed before input pins.

Explicit netlists include the pin name and the net connected to the pin; the order is not important. Library Compiler does not support this format.

Gate-level Verilog netlists from the Verilog simulator use the implicit format, with output pins listed before input pins. Other Design Compiler input and output interfaces that use implicit pin-ordering schemes are NDL (LSI netlist) and TDL (Tegas netlist). TDL, VHDL, and LSI netlists can have explicit and implicit formats.

In Example 7-17, the D pin is defined first and the clock is defined second in the DFF reference in the technology library.

Example 7-17 Partial TDL Netlist With Implicit Pin Ordering

```
COMPILE;
DIRECTORY MASTER;
MODULE CALCSEL;
INPUTS  OPRDY,ID_REG_2,ID_REG_1,ID_REG_0,INTF,CLK;
OUTPUTS SEL_Q;

LEVEL FUNCTION;
DEFINE
N27 = (OPRDY);
º
º
N35 = (CLK);
U52(N65) = ND2(N66,N27);
U54(N64) = INV(N65);
º
º
U29(N41) = MUX21(N42,N43,N44);
I0_FF(N38) = DFF(N37,N35);

END MODULE;

END COMPILE;
END;
```

Example 7-17 uses an implicit pin-referencing netlist scheme. Design Compiler has no information about the pin ordering used by a specific cell, except for the order of pins used in the cell descriptions of the Design Compiler technology library. Therefore, if an implicit pin-referencing netlist is used as an input to Design Compiler, special attention must be paid to the order in which the input and output pins are listed in the cell descriptions in a technology library. Synopsys tools always write explicit pin-referencing netlists, as shown in the following example.

```
U29(N41=Z) = MUX21(N42=A,N43=B,N44=C);
```

Verilog Netlist-Pin Order

In cases where Verilog simulator models of ASIC cells have a different pin ordering that can conflict with the one defined in the technology library used with Synopsys, Design Compiler reads in the Verilog netlist from the simulator environment because it is syntactically correct. However, when Design Compiler performs synthesis and optimization on the design, either the design does not optimize or the results obtained are incorrect.

The discrepancy is usually in the description of flip-flops. If the D and CLK pins are reversed, the result can be designs that are incorrectly optimized. In some cases, the flip-flops disappear from the design. Typically, this happens when a component has been instantiated in a Verilog design, as in the following example.

```
Verilog Output:DFF   U1(Q, QN, D, CLK)
Design Compiler expects:DFF U1(D, CLK, Q, QN)
```

In the preceding example, the Verilog netlists were written out by use of an implicit pin-referencing scheme where the output pins are listed first, then the inputs. The technology library description for DFF has the input pins D and CLK described first, then the output pins Q and QN. In this case, the inputs and output pins are reversed.

To avoid this problem, the order of the pins in the Design Compiler technology library and the netlist generator you use must be identical. Design Compiler can read implicit or explicit pin-referencing netlists, but always writes out explicit pin-referencing netlists.

8

Defining Sequential Cells

This chapter describes the peculiarities of defining flip-flops and latches, building upon the cell description syntax given in Chapter 7, "Defining Core Cells." It describes group statements that apply only to sequential cells and also describes a variation of the `function` attribute that makes use of state variables.

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

- Using Sequential Cell Syntax
- Describing a Flip-Flop
- Using the function Attribute
- Describing a Multibit Flip-Flop
- Describing a Latch
- Describing a Multibit Latch
- Describing Sequential Cells With the Statetable Format
- Critical Area Analysis Modeling
- Flip-Flop and Latch Examples
- Cell Description Examples

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 complex sequential cells, such as the following:

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

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

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

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

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

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

- Library Compiler uses the statetable format to generate the VITAL model of a VHDL library.

- DFT Compiler supports the statetable format design rule checking (DRC) but requires ff/ latch format for scan substitution. See Chapter 10, "Defining Test Cells," for information about modeling cells for test.

Describing a Flip-Flop

To describe an edge-triggered storage device, include a `ff` group or a `statetable` group in a cell definition. This section describes how to define a flip-flop by using the ff/latch format. See "Describing Sequential Cells With the Statetable Format" on page 8-26 for the way to define cells using the `statetable` group.

Using the ff Group

A `ff` group describes either a single-stage or a master-slave flip-flop. The `ff_bank` group represents multibit registers, such as a bank of flip-flops. See "Describing a Multibit Flip-Flop" on page 8-12 for more information 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" ;
         power_down_function : "Boolean_expression" ;}}}
```

variable1

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

variable2

The state of the inverting output

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

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

clocked_on and clocked_on_also Attributes

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

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

Examples

A rising-edge-triggered device is:

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

A falling-edge-triggered device is:

```
clocked_on : "CP’";
```

next_state Attribute

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

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

Example

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

nextstate_type Attribute

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

Any pin with the `nextstate_type` attribute must be included in the value of the `next_state` attribute. Library Compiler checks the consistency between the pin’s `nextstate_type` attribute and the `next_state` attribute.

Note:

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

Example

```
nextstate_type : data;
```

Example 8-5 on page 8-11 and Example 8-6 on page 8-14 show the use of the `nextstate_type` attribute.

clear Attribute

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

The example defines an active-low clear signal.

Example

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

For more information about the `clear` attribute, see “Describing a Single-Stage Flip-Flop” on page 8-8.

preset Attribute

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

The example defines an active-high preset signal.

Example

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

For more information about the preset attribute, see “Describing a Single-Stage Flip-Flop” on page 8-8.

clear_preset_var1 Attribute

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

Example

```
clear_preset_var1 : L;
```

For more information about the `clear_preset_var1` attribute, including its function and values, see “Describing a Single-Stage Flip-Flop” on page 8-8.

clear_preset_var2 Attribute

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

Example

```
clear_preset_var2 : L ;
```

For more information about the `clear_preset_var2` attribute, including its function and values, see “Describing a Single-Stage Flip-Flop” on page 8-8.

power_down_function Attribute

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

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

Syntax

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

Example

```
library ("low_power_cells") {
   cell ("retention_dff") {
      pg_pin(VDD) {
         voltage_name : VDD;
         pg_type : primary_power;
      }
      pg_pin(VSS) {
         voltage_name : VSS;
```

```
            pg_type : primary_ground;
         }
         pin ("D") {
            direction : "input";
         }
         pin ("CP") {
            direction : "input";
         }
         ff(IQ,IQN) {
            next_state : "D" ;
            clocked_on : "CP" ;
            power_down_function : "!VDD + VSS" ;
         }
         pin ("Q") {
            function : " IQ ";
            direction : "output";
            power_down_function : "!VDD + VSS";
         }
      …
      }
…
}
```

ff Group Examples

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

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

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

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

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

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

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

Describing a Single-Stage Flip-Flop

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

Table 8-1 shows the functions of the attributes in the `ff` group for a single-stage flip-flop.

Table 8-1 Function Table for Single-Stage Flip-Flop

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

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

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

The `clear_preset_var1` and `clear_preset_var2` attributes specify the value for *variable1* and *variable2* when `clear` and `preset` are both active at the same time. Valid values are shown in Table 8-2.

Table 8-2 Valid Values for the clear_preset_var1 and clear_preset_var2 Attributes

Variable values	Equivalence
L	0
H	1
N	No change[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. Design Compiler does not recognize these values and issues a warning that the cell is a black box.

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

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

Example 8-1 is a `ff` group for a single-stage D flip-flop with a rising edge, negative clear and preset, and the output pins set to 0 when both `clear` and `preset` are active (low).

Example 8-1 Single-Stage D Flip-Flop

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

Example 8-2 is a `ff` group for a single-stage, rising edge-triggered JK flip-flop with scan input, negative clear and preset, and the output pins set to 0 when `clear` and `preset` are both active.

Example 8-2 Single-Stage JK Flip-Flop

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

Example 8-3 is a `ff` group for a D flip-flop with synchronous negative clear.

Example 8-3 D Flip-Flop With Synchronous Negative Clear

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

Describing a Master-Slave Flip-Flop

The specification for a master-slave flip-flop is the same as for a single-stage device, except that it includes the `clocked_on_also` attribute. Table 8-3 shows the functions of the attributes in the `ff` group for a master-slave flip-flop.

Table 8-3 Function Tables for Master-Slave Flip-Flop

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

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

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

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

Example 8-4 shows the `ff` group for a master-slave D flip-flop with a rising-edge sampling, falling-edge data transfer, negative clear and preset, and output values set to high when the `clear` and `preset` attributes are both active.

Example 8-4 Master-Slave D Flip-Flop

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

Using the function Attribute

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

Example 8-5 shows a complete functional description of a rising-edge-triggered D flip-flop with active-low clear and preset.

Example 8-5 D Flip-Flop Description

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

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

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

Describing a Multibit Flip-Flop

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

The syntax is similar to that of the `ff` group; see “Describing a Flip-Flop” on page 8-3.

Syntax

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

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

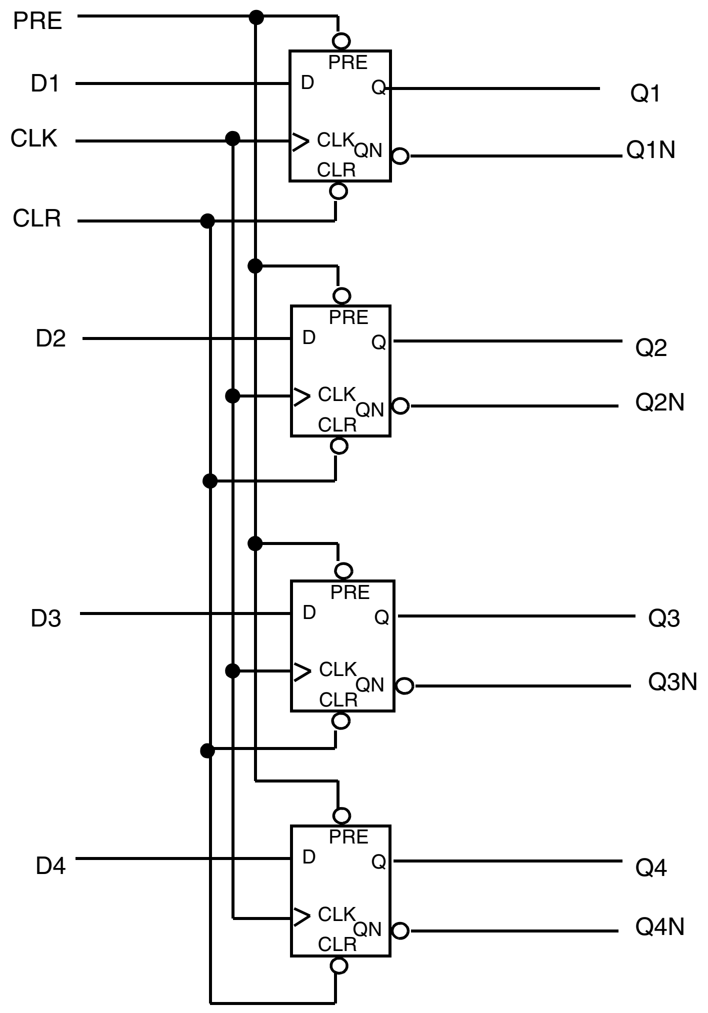
Three-state output pins are allowed; you can add a `three_state` attribute to an output bus or bundle to specify this function.

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

Figure 8-1 shows a multibit register containing four rising-edge-triggered D flip-flops with `clear` and `preset`.

Example 8-6 is the description of the multibit register shown in Figure 8-1.

Figure 8-1 Multibit Flip-Flop



Example 8-6 Multibit D Flip-Flop Register

```
    cell (dff4) {
      area : 1 ;
      pin (CLK) {
        direction : input ;
        capacitance : 0 ;
        min_pulse_width_low  : 3 ;
        min_pulse_width_high : 3 ;
      }
      bundle (D) {
          members(D1, D2, D3, D4);
          nextstate_type : data;
          direction : input ;
          capacitance : 0 ;
          timing() {
          related_pin     : "CLK" ;
          timing_type     : setup_rising ;
          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 */
```

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” on page 8-26 for information about defining cells using the `statetable` group.

latch Group

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

Syntax

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

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 8-4.

Example

```
clear_preset_var1 : L;
```

Table 8-4 Valid Values for the clear_preset_var1 and clear_preset_var2 Attributes

Variable values	Equivalence
L	0
H	1

Table 8-4 Valid Values for the clear_preset_var1 and clear_preset_var2 Attributes (Continued)

Variable values	**Equivalence**
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. Design Compiler does not recognize these values and issues a warning that the cell is a black box.*

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 8-4.

Example

```
clear_preset_var2 : L ;
```

Table 8-5 shows the functions of the attributes in the `latch` group.

Table 8-5 Function Table for latch Group Attributes

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

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

Example 8-7 shows a `latch` group for a D latch with active-high `enable` and negative `clear`.

Example 8-7 D Latch With Active-High enable and Negative clear

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

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

Example 8-8 SR Latch

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

The “Sample CMOS Technology Library” appendix in the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide* shows a fully described D latch.

Determining a Latch Cell’s Internal State

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

For more information about `power_down_function`, see “power_down_function Attribute” on page 8-6.

power_down_function Syntax For Latch Cells

```
library (name) {
   cell (name) {
      latch
(variable1,variable2)
{
         //...latch
description...
         clear : "Boolean expression" ;
         clear_preset_var1 : L | H | N | T | X ;
         clear_preset_var2 : L | H | N | T | X ;
         data_in : "Boolean expression" ;
         enable : "Boolean expression" ;
         preset : "Boolean expression" ;
         power_down_function : "Boolean expression" ;
      }
   …
   }
…
}
```

Describing a Multibit Latch

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

latch_bank Group

The syntax is similar to that of the `latch` group. See "Describing a Latch" on page 8-17.

Syntax

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

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

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

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

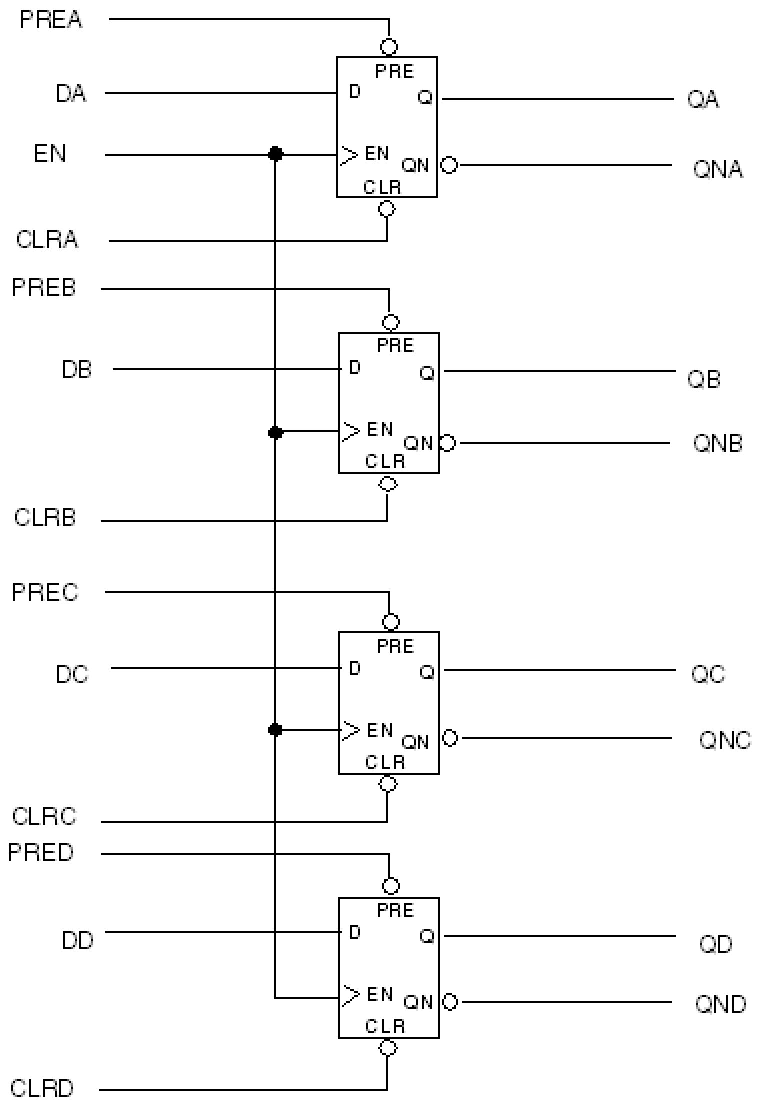
Example 8-9 shows a `latch_bank` group for a multibit register containing four rising-edge-triggered D latches.

Example 8-9 Multibit D Latch

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

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

Figure 8-2 shows a multibit register containing four high-enable D latches with `clear`.

Figure 8-2 Multibit Latch



Example 8-10 is the cell description of the multibit register shown in Figure 8-2 that contains four high-enable D latches with `clear`.

Example 8-10 Multibit Latches With clear

```
cell (DLT2) {

/* note: 0 hold time */

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

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

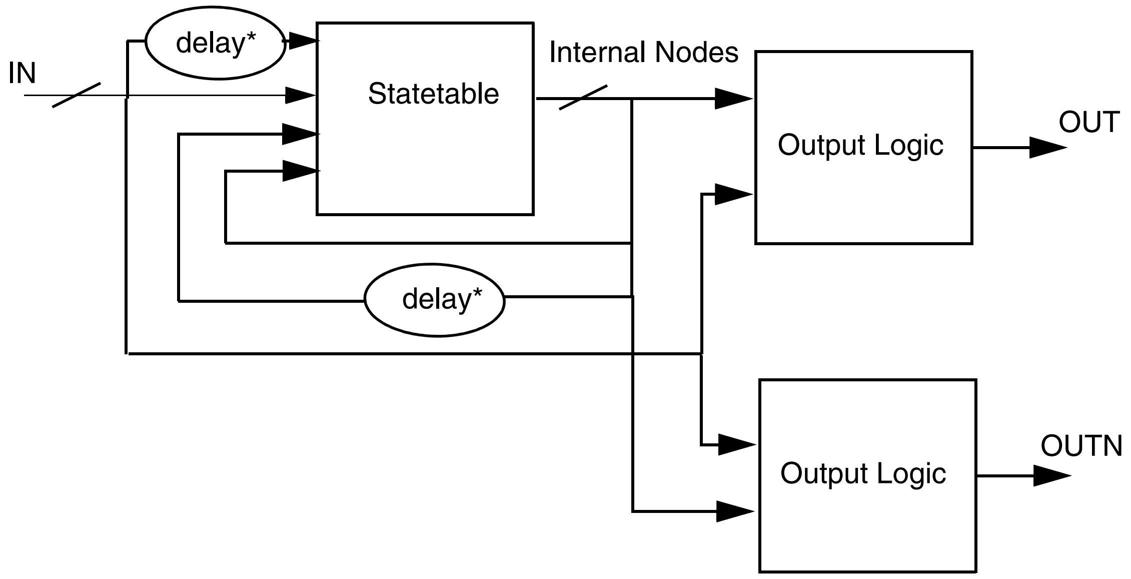
```
      timing() {
        related_pin     : "PRE" ;
        timing_type     : clear ;
        timing_sense    : positive_unate ;
        intrinsic_fall  : 1.0 ;
      }
    }
  } /* end of cell DLT2 */
```

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 Library Compiler cell description.

Figure 8-3 shows how you can model each sequential output port (OUT and OUTN) in a sequential library cell.

Figure 8-3 Generic Sequential Library Cell Model



OUT and OUTN

Sequential output ports of the sequential cell.

IN

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

delay*

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

Statetable

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

Internal Nodes

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

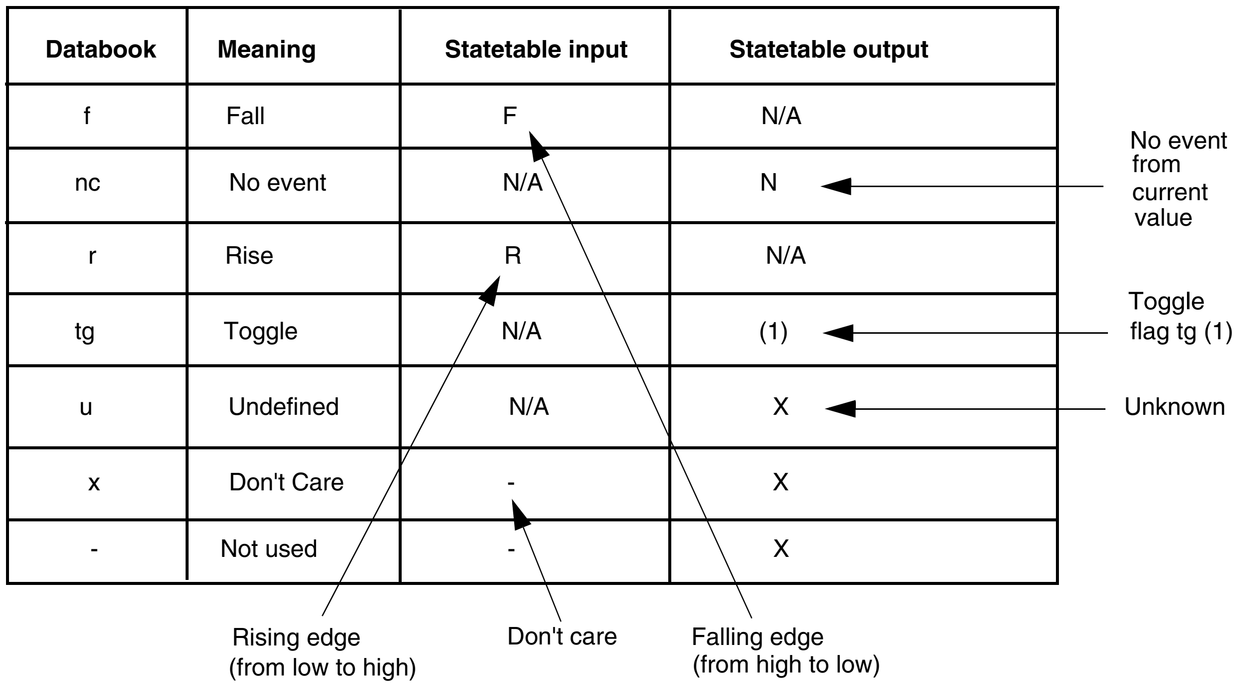
Output Logic

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

To capture the function of complex sequential cells, use the `statetable` group in the `cell` group to define the statetable in Figure 8-3. The `statetable` group syntax maps to the truth tables in databooks.

Figure 8-4 is an example of a table that maps a truth table from an ASIC vendor's databook to the statetable syntax. For table input token values, see "statetable Group" on page 8-29.

Figure 8-4 Mapping Databook Truth Table to Statetable



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

No event from current value
Toggle flag tg (1)
Unknown
Rising edge (from low to high)
Don't care
Falling edge (from high to low)

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

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

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

statetable Group

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

Syntax

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

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

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

Figure 8-5 shows an example of a `statetable` group.

Figure 8-5 statetable Group Example

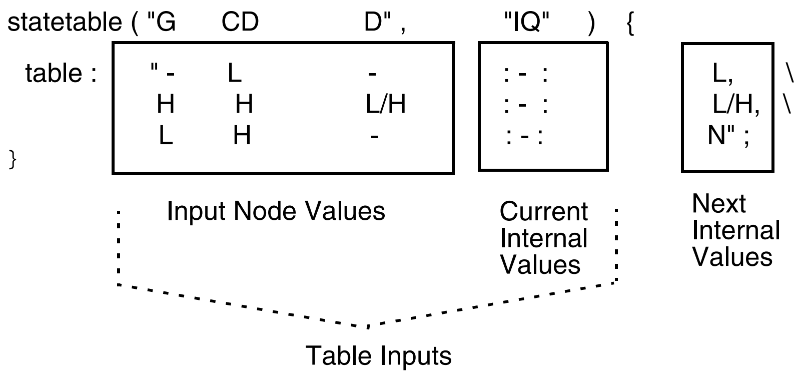


Table 8-6 shows the token values for table inputs.

Table 8-6 Legitimate Values for Table Inputs (Input and Current Internal Nodes)

Input 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 8-7 shows the token values for the next internal node.

Table 8-7 Legitimate Values for Next Internal Node

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

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

Using Shortcuts

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

Example 8-11 shows a fully expanded `statetable` group for a data latch with active-low `clear`.

Example 8-11 Fully Expanded statetable Group for Data Latch With Active-Low Clear

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

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

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

Table 8-8 Internal Representation

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

Priority ordering of outputs

The outputs follow a prioritized order. Two rules can cover the same input combinations (with a warning that Library Compiler provides), but the rule defined first has priority over subsequent rules.

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

Partitioning the Cell Into a Model

You can partition a structural netlist to match the general sequential output model described in Example 8-12 by performing these tasks:

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

Internally, the statetable is broken down into specific subtables for each internal node. The amount of memory and processing power required for a subtable is related exponentially to the number of inputs to the subtable. Library Compiler issues a warning when a subtable requires more than 16 inputs; however, a subtable might be of any size if the workstation has the resources to process it.

Note:
There are two ways to specify that an output does not change logic values. One way is to use the inactive N value as the next internal value, and the other way is to have the next internal value remain the same as the current internal value (see the italic lines in Example 8-12).

Example 8-12 JK Flip-Flop With Active-Low, Direct-Clear, and Negative-Edge Clock

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

In Example 8-12, the value of the next internal node of the second rule is N, because both the clear CD and clock CN are inactive. The value of the next internal node of the third rule is L/H, because the clock is active.

Defining an Output pin Group

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

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

An output pin can have one of the following combinations:

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

state_function Attribute

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

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

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

Example

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

internal_node Attribute

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

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

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

Example

```
internal_node :

"IQ";
```

input_map Attribute

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

Example

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

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

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

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

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

Internal node

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

Synchronous data input node

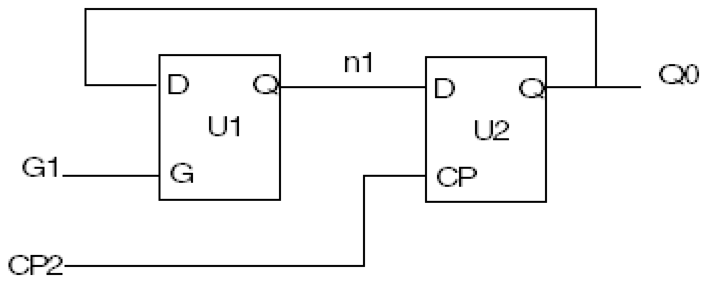
  When an output port is specified for this type of node, the input is forced to map to the delayed value of the output port.

Asynchronous or clock input node

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

For example, Figure 8-6 shows a circuit consisting of latch U1 and flip-flop U2.

Figure 8-6 Circuit With Latch and Flip-Flop



In Figure 8-6, internal pin n1 should map to ports “Q0 G1 n1*" and output Q0 should map to “n1* CP2 Q0*". The subtle significance is relevant during simultaneous input transitions.

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

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

inverted_output Attribute

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

Example

```
inverted_output : true;
```

Some downstream tools make assumptions based on whether an output is considered inverted or noninverted. In the statetable format, an output is considered inverted if it has any data input with a negative sense. This algorithm might be inconsistent with a user’s intentions. For example, whether a toggle output is considered inverted or noninverted is arbitrary.

Internal Pin Type

In 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.

Because a cell with a statetable model is considered a black box for synthesis, no checks are performed to ensure consistency between timing and function.

Example

```
pin (n1) {
   direction : internal;
   internal_node :"IQ"; /* use default input_map */
   ...
}
pin (QN) {
   direction : output;
   internal_node :"IQN";
   input_map : "Gx1 CDx1 Dx1 QN"; /* QN is internal node */

   three_state : "EN2";
   ...
}
pin (QNZ) {
   direction : output;
   state_function :"QN"; /* ignores QN’s three state */

   three_state : "EN";
   ...
}
pin (Y) {
   direction : output;
   state_function : "A"; /* combinational feedthrough */

   ...
}
```

Checking Statetables

In addition to checking for syntax errors, Library Compiler checks for the following errors in `statetable` groups:

- An input that is not functionally required by any of the internal nodes
- An internal node that is purely combinational (that is, the node has no N)
- L/H or H/L present in the output without any L/H or H/L in the inputs
- An input or an internal node name in the statetable that is the same as another input or internal node name in the statetable
- Overlapping entries and unspecified entries, which cause warnings

In `pin` groups, Library Compiler checks for the following errors:

- A functionally related input that is explicitly defined as don't care (-) in the `input_map` attribute
- Too many names defined in the `input_map` attribute
- An internal node name in the `input_map` attribute that does not correspond to the current port name, which results in a warning
- An `internal_node` attribute that does not match any statetable internal node name
- Ports that have missing or illegal attributes
- An input port defined as an internal node in the `input_map` attribute
- Specified internal pins that do not affect any outputs, which results in a warning
- Inclusion of illegal ports in the `state_function` expression; illegal ports are outputs without an `internal_node` attribute and inouts without `three_state` attributes
- Not having any output with an `internal_node` attribute

update_lib Command

The `update_lib` command supports the statetable format as fully as the `read_lib` command does. With the statetable format, there is no difference between a cell created by the `read_lib` command and a cell created by the `update_lib` command.

Note, however, that `report_lib` command does not report on statetable information after a `update_lib` command.

Timing

With the statetable format, you can specify timing arcs to internal pins.

When you use the ff/latch format in a cell, Library Compiler performs the same timing checks as it does with the statetable format.

When you define only the statetable, Library Compiler checks the timing types only for consistency with other timing types and with the three-state conditions. Library Compiler does not check the timing types against the functional statetable description.

Recommended Statetable Practices

Within the flexibility allowed in specifying the statetable, it is important to follow guidelines consistently for maintainability and legibility. Use the `report_lib` command to verify that the statetable is interpreted correctly.

Beginning users should observe the following recommendations:

- Align the node names and node values for legibility. Align the node values in a column, as shown here:

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

- Use detailed comments when appropriate.

```
statetable("DCP""IQ") {
   table : "H/L R : - : H/L,/*data capture*/\
           -    ~R : - : N"; /* hold */
}
```

- Place each rule on a separate line.
- Practice with sequential elements that are also specified by the ff/latch format. Verification is automatic.
- Set the `libgen_max_differences` environment variable to -1.
- Do not use unspecified rules or overlapping rules.
- Do not manually suppress warnings or errors.
- Use real port names in the statetable. Make sure that the node names correspond to the port names. This correspondence reduces the number of names to track.
- Be explicit in rules. Be careful when using shortcuts.

- Be explicit when using `input_map`. If possible, do not use default names.
- Represent complex cells, such as master-slave cells, in one statetable, as shown here:

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

- Avoid using internal pins.
- Use the `report_lib` command to double-check the statetable and the table input mapping. For information on using the `report_lib` command, see the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide*.
- Do not map outputs to edge-sensitive inputs; use a more-complex statetable.
- Do not represent more than one transition per rule.
- Describe test cells only in the ff/latch format.

More experienced users can follow these recommendations:

- Use the don't care (-) shortcuts to reduce the number of rules (see "Using Shortcuts" on page 8-31).
- Use rule order priority to reduce the number of rules.

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

- Use the `input_map` attribute to represent networks of tables (for example, shift registers).
- Use the default `input_map` attribute by defining only the minimum number of port names.
- Use the `state_function` attribute for all output logic.
- Define delayed inputs in the statetable and in the `input_map` attribute.
- Filter out warnings.
- Specify internal nodes in the statetable that are not referenced—for example, Q and QN—even though the cell has only a Q output. Such specification makes it easier to reuse statetables among library cells and incurs only a small cost in Library Compiler processing.

Determining a Complex Sequential Cell’s Internal State

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

For more information about the `power_down_function` attribute, see “power_down_function Attribute” on page 8-6.

power_down_function Syntax for State Tables

The `power_down_function` attribute is specified after `table`, in the `statetable` group, as shown in the following syntax:

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

Critical Area Analysis Modeling

Liberty syntax models critical area analysis data for library cells 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.

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_template>) {
          values("float1, float2,… floatn");
      }

      critical_area_table (<template_name>) {
          defect_type : enum (open, short, open_and_short);

          related_layer : string;
          index_1 ("float1, float2, … floatn");
          values ("float1, float2, … floatn");
      }
…
    }
  }
}
```

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.

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 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.

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");
     }
     …
     }
  }
}
```

Flip-Flop and Latch Examples

Example 8-13 through Example 8-25 show flip-flops and latches in ff/latch and statetable syntax.

Example 8-13   D Flip-Flop

```
    /* ff/latch format */

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

```
}
/* statetable format */

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

Example 8-14 D Flip-Flops With Master-Slave Clock Input Pins

```
/* ff/latch format */

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

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

Example 8-15 D Flip-Flop With Gated Clock

```
/* ff/latch format */

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

/* statetable format */

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

Example 8-16 D Flip-Flop With Active-Low Direct-Clear and Active-Low Direct-Set

```
/* ff/latch format */

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

```
    clear : " CD’ " ;
    preset : " SD’ " ;
    clear_preset_var1 : L ;
    clear_preset_var2 : L ;
  }

  /* statetable format */

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

Example 8-17 D Flip-Flop With Active-Low Direct-Clear, Active-Low Direct-Set, and One Output

```
  /* ff/latch format */

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

  /* statetable format */

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

Example 8-18 JK Flip-Flop With Active-Low Direct-Clear and Negative-Edge Clock

```
  /* ff/latch format */

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

  /* statetable format */
```

```
statetable(" J   K   CD   CD",  "IQ") {
   table : " -   -   -    L    :  -  :   L,\
             -   -   ~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 8-19 D Flip-Flop With Scan Input Pins

```
/* ff/latch format */

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

/* statetable format */

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

Example 8-20 D Flip-Flop With Synchronous Clear

```
/* ff/latch format */

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

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

Example 8-21 D Latch

```
/* ff/latch format */

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

/* statetable format */

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

Example 8-22 SR Latch

```
   /* ff/latch format */

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

   /* statetable format */

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

Example 8-23 D Latch With Active-Low Direct-Clear

```
   /* ff/latch format */

   latch (IQ,IQN) {
     data_in : "D" ;
     enable : "G" ;
     clear : " CD’ " ;
   }

   /* statetable format */

   statetable(" D   G   CD",   "IQ") {
      table : " H/L H   H   : - :   H/L,\
                -   L   H   : - :   N,\
                -   -   L   : - :   L";
   }
```

Example 8-24 Multibit D Latch With Active-Low Direct-Clear

```
   /* ff/latch format */

     latch_bank(IQ, IQN, 4) {
       data_in : "D" ;
       enable  : "EN" ;
       clear   : "CLR’" ;
       preset  : "PRE’" ;
       clear_preset_var1  : H ;
       clear_preset_var2 : H ;
     }
```

```
/* statetable format */

statetable(" D   EN  CL  PRE","IQ IQN") {
   table : "H/L  H   H   H  : -   - :  H/L  L/H,\
            -    L   H   H  : -   - :  N    N,\
            -    -   L   H  : -   - :  L    H,\
            -    -   H   L  : -   - :  H    L,\
            -    -   L   L  : -   - :  H    H";
}
```

Example 8-25 D Flip-Flop With Scan Clock

```
statetable(" D   S   CD  SC  CP",  "IQ") {
   table : " H/L  -    ~R  R   : - :  H/L,\
             -    H/L  R   ~R  : - :  H/L,\
             -    -    ~R  ~R  : - :  N,\
             -    -    R   R   : - :  X";
}
```

Cell Description Examples

Example 8-26 through Example 8-28 illustrate some of the concepts this chapter discusses.

Example 8-26 D Latches With Master-Slave Enable Input Pins

```
cell(ms_latch) {
  area : 16;
  pin(D) {
    direction : input;
    capacitance : 1;
  }
  pin(G1) {
    direction : input;
    capacitance : 2;
  }
  pin(G2) {
    direction : input;
    capacitance : 2;
  }
  pin(mq) {
    internal_node : "Q";
    direction : internal;
    input_map : "D G1";
    timing() {
      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 8-27   FF Shift Register With Timing Removed

```
    cell(shift_reg_ff) {
      area : 16;
      pin(D) {
        direction : input;
        capacitance : 1;
      }
      pin(CP) {
        direction : input;
        capacitance : 2;
      }
      pin (Q0) {
        direction : output;
        internal_node : "Q";
        input_map : "D CP";
      }
      pin (Q1) {
        direction : output;
        internal_node : "Q";
        input_map : "Q0 CP";
      }
      pin (Q2) {
        direction : output;
        internal_node : "Q";
        input_map : "Q1 CP";
      }
      pin (Q3) {
        direction : output;
        internal_node : "Q";
        input_map : "Q2 CP";
      }
      statetable( "D CP", "Q QN" ) {
        table : "-   ~R : - - : N    N,\
                 H/L  R : - - : H/L  L/H";
      }
    }
```

Example 8-28 FF Counter With Timing Removed

```
  cell(counter_ff) {
    area : 16;
    pin(reset) {
      direction : input;
      capacitance : 1;
    }
    pin(CP) {
      direction : input;
      capacitance : 2;
    }
    pin (Q0) {
      direction : output;
      internal_node : "Q0";
      input_map : "CP reset Q0 Q1";
    }
    pin (Q1) {
      direction : output;
      internal_node : "Q1";
      input_map : "CP reset Q0 Q1";
    }
    statetable( "CP reset", "Q0 Q1" ) {
      table : "-   L : - - : L  H,\
            ~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";
    }
  }
```

9

Defining I/O Pads

To define I/O pads, you use the `library`, `cell`, and `pin` group attributes that describe input, output, and bidirectional pad cells. The pad modeling features support the Design Compiler synthesis of I/O pads in FPGA and standard CMOS technologies.

To model I/O pads, you must understand the following concepts covered in this chapter:

Special Characteristics of I/O Pads

I/O pads are the special cells at the chip boundaries that allow communication with the world outside the chip. Their characteristics distinguish them from the other cells that make up the core of an integrated circuit. These characteristics must be described in Synopsys technology libraries so that Design Compiler can understand the pads well enough to insert them automatically during synthesis.

Pads typically have longer delays and higher drive capabilities than the cells in an integrated circuit's core. Because of their higher drive, CMOS pads sometimes exhibit noise problems. Slew-rate control is available on output pads to help alleviate this problem.

Pads are fixed resources—a limited number are available on each die size. In addition, special types, such as clock pads, might be more limited than others. These limits must be modeled.

Another distinguishing feature of pad cells is the voltage level at which input pads transfer logic 0 or logic 1 signals to the core or at which output pad drivers communicate logic values from the core.

Integrated circuits that communicate with one another must have compatible voltage levels at their pads. Because pads communicate with the world outside the integrated circuit, you must describe the pertinent units of peripheral library properties, such as external load, drive capability, delay, current, power, and resistance. This description makes it easier to design chips from multiple technologies.

Some libraries create logical pads out of multiple cells; such logical pads must be modeled so that they are just as easy to insert automatically on a design as a single-cell pad.

You must capture all these properties in the library to make it possible for the integrated circuit designer to insert the correct pads during synthesis.

Identifying Pad Cells

Use the attributes described in the following sections to specify I/O pads and pad pin behaviors.

Regular Pad Cells

Use the `pad_cell` attribute to tag a cell as an I/O pad. Design Compiler filters pad cells out of the normal core optimization and treats them differently during technology translation.

Example

```
pad_cell : true;
```

Clock Pads

Certain pad and auxiliary pad cells are special and require special treatment. The most important of these is the clock driver pad cell. It is important to identify clock drivers, because Design Compiler and DFT Compiler treat them differently.

To designate a clock pad on a cell tagged with a `pad_cell` or an `auxiliary_pad_cell` attribute, use the `pad_type` attribute statement.

Example

```
pad_type : clock;
```

Pad Pin Attributes

Each cell identified as an I/O pad must have a pin that represents the pad. Additionally, the pin must have a direction indicating whether the cell is an input, output, or bidirectional pad.

Use the `is_pad` attribute on pins or parameterized pins to identify which pins are to be considered logical I/O pad pins. Use the `is_pad` attribute on at least one pin of a cell with a `pad_cell` attribute.

The `direction` attribute identifies whether the pad is an input, output, or bidirectional pad.

Example

```
pin(PAD) {
   direction : output;
   is_pad : true;
...
}
```

If a pin is bidirectional, you can supply multiple `driver_type` attributes for the same pin to model both the output behavior and the input behavior. An example is

```
driver_type : "open_drain pull_up" ;
```

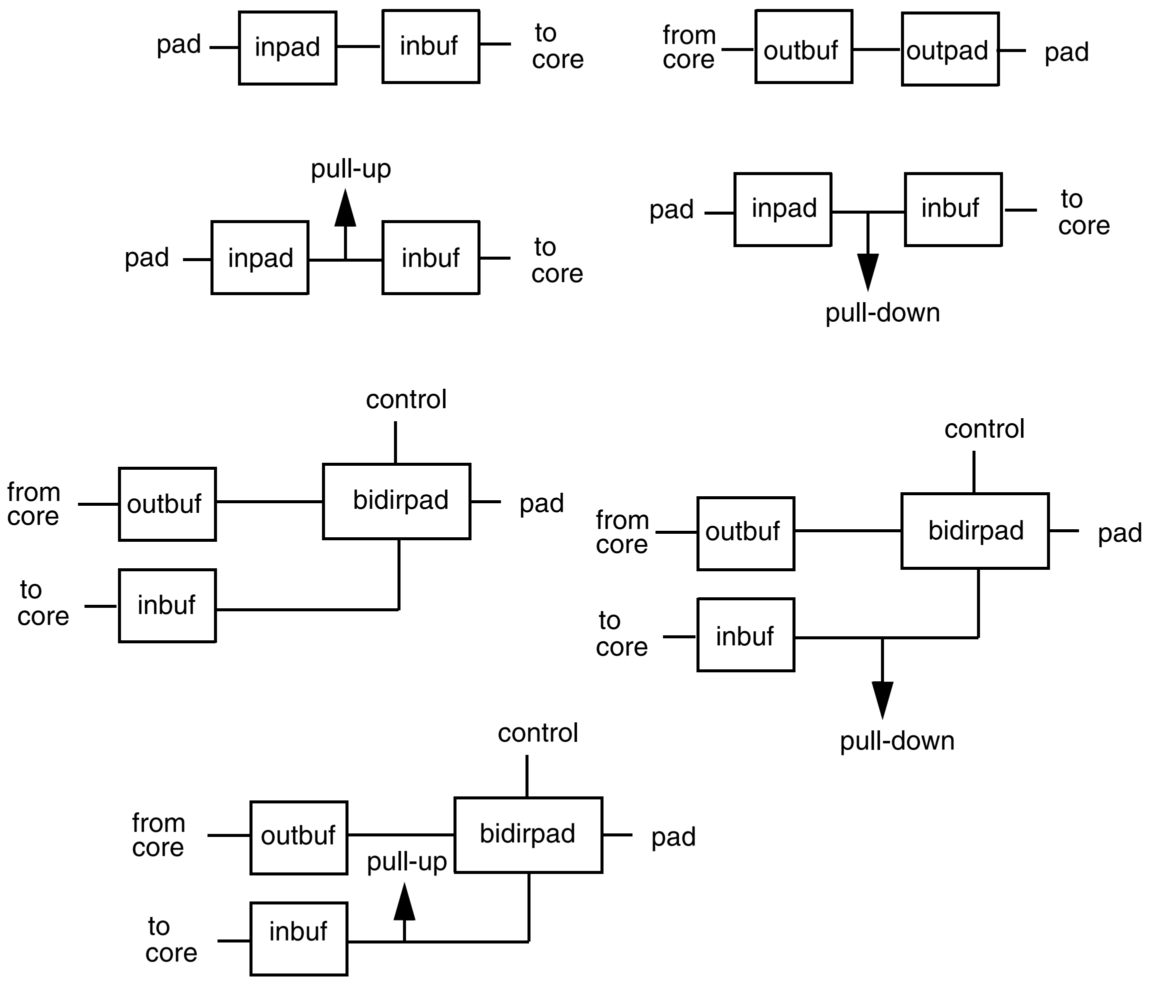
Describing Multicell Pads

I/O pads can be represented either as one cell with many attributes or as a collection of cells, each with attributes affecting the operation of the logical pad. FPGA and CMOS technologies implement pads with multiple cells—a pad cell and a driver cell, for example.

The first type of library contains a different cell for each possible combination of pad characteristics. The second type contains a smaller number of components, but each type of pad must be individually constructed from these components.

Figure 9-1 shows the different components that make up multicell pads in the second type of library. See Example 9-9 on page 9-24 for the Library Compiler descriptions of these components.

Figure 9-1 Multicell Pads



Auxiliary Pad Cells

If you implement your pads with more than one cell, you can identify those cells that can be used to build up a logical pad with an `auxiliary_pad_cell` attribute. This attribute indicates that the cell can be used as part of a logical pad.

Example

```
auxiliary_pad_cell : true;
```

Identifying such cells helps Design Compiler create I/O pads consisting of more than one cell. Cells that have the `auxiliary_pad_cell` attribute can also be used within the core.

Auxiliary pad cells are used only with pull-up and pull-down devices.

Connecting Pins

If your library uses multicell pads, Design Compiler needs to know which pins on the various cells to connect. This information lets the tool implement the pad properly. Two attributes give this information:

`multicell_pad_pin`

Identifies which pins to connect to create a working multicell pad. Use this attribute to flag all the pins to connect on a pad or an auxiliary pad cell.

`connection_class`

Indicates which pins to connect to pins on other cells. For example, pull-up cells are generally connected to the network between the input pad cell and the input driver cell. This attribute tells Design Compiler which classes of pins to connect.

Example

```
multicell_pad_pin : true;
connection_class : "INPAD_NETWORK";
```

Defining Units for Pad Cells

To process pads for full-chip synthesis, Design Compiler requires information about the units of time, capacitance, resistance, voltage, current, and power. Library Compiler uses the following attributes to provide the required unit information:

- `time_unit`
- `capacitive_load_unit`
- `pulling_resistance_unit`
- `voltage_unit`
- `current_unit`
- `leakage_power_unit`

All these attributes are defined at the library level, as described in “Defining Units” on page 6-13. These values are required. If you do not supply these attributes, Design Compiler prints a warning.

Units of Time

The VHDL library generator uses the optional `time_unit` attribute to identify the physical time unit used in the generated library.

Example

```
time_unit : 10ps ;
```

Capacitance

The `capacitive_load_unit` attribute defines the capacitance associated with a standard load. If you already represent capacitance values in terms of picofarads or femtofarads, use this attribute to define your base unit. If you represent capacitance in terms of the standard load of an inverter, define the exact capacitance for that inverter—for example, 0.101 pF.

Example

```
capacitive_load_unit( 0.1,ff );
```

Resistance

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";
```

Voltage

You can use the `input_voltage` and `output_voltage` groups to define a set of input or output voltage ranges for your pads. To define the units of voltage you use for these groups, use the `voltage_unit` attribute. All the attributes defined inside `input_voltage` and `output_voltage` groups are scaled by the value defined for `voltage_unit`. In addition, the `voltage` attribute in the `operating_conditions` groups also represents its values in these units.

Example

```
voltage_unit : "1V";
```

Current

You can define the drive current that can be generated by an output pad and also define the pulling current for a pull-up or pull-down transistor under nominal voltage conditions. Define all current values with the library-level `current_unit` attribute.

Example

```
current_unit : "1uA";
```

Power

The `leakage_power_unit` attribute defines the units of the power values in the library. If this attribute is missing, the leakage-power values are expressed without units.

Example

```
leakage_power_unit : "100uW" ;
```

Describing Input Pads

To represent input pads in your technology library, you must describe the input voltage characteristics and indicate whether hysteresis applies.

The input pad properties are described in the next section. Examples at the end of this chapter describe a standard input buffer, an input buffer with hysteresis, and an input clock buffer.

Voltage Levels

You can use the `input_voltage` group at the library level to define a set of input voltage ranges for your pads. You can then assign this set of voltage ranges to the input pin of a pad cell. For example, you can define an `input_voltage` group called TTL with a set of high and low thresholds and minimum and maximum voltage levels and use this command in the `pin` group to assign those ranges to the pad cell pin:

```
input_voltage : TTL ;
```

You can include the `vil`, `vih`, `vimin`, and `vimax` attributes in the `input_voltage` group. See "input_voltage Group" on page 6-16 for additional information.

Hysteresis

You can indicate an input pad with hysteresis, using the `hysteresis` attribute. The default for this attribute is false. When it is true, the `vil` and `vol` voltage ratings are actual transition points.

Example

```
hysteresis : true;
```

Pads with hysteresis sometimes have derating factors that are different from those of cells in the core. As a result, you need to describe the timing of cells that have hysteresis with a `scaled_cell` group, which is described in “hysteresis Attribute” on page 7-53. This construct provides derating capabilities and minimum, typical, or maximum timing for cells.

Describing Output Pads

To represent output pads in your technology library, you must describe the output voltage characteristics and the drive-current rating of output and bidirectional pads. Additionally, you must include information about the slew rate of the pad. These output pad properties are described in the sections that follow.

Examples at the end of this chapter show a standard output buffer and a bidirectional pad.

Voltage Levels

You can use the `output_voltage` group at the library level to define a set of output voltage ranges for your pads. You can then use the `output_voltage` attribute at the pin level to assign this set of voltage ranges to the output pin of a pad cell.

Example

```
output_voltage(TTL) {
   vol : 0.4;
   voh : 2.4;
   vomin : -0.3;
   vomax : VDD + 0.3;
}
```

You can include `vol`, `voh`, `vomin`, and `vomax` values in `output_voltage` groups. In this example, an `output_voltage` group called TTL contains a set of high and low thresholds and minimum and maximum voltage levels.

In the `pin` group, you can assign those ranges to the pad cell pin with the `output_voltage` attribute:

```
pin(Z) {
   direction : output;
   is_pad : true;
   output_voltage : TTL ;
   ...
}
```

See “output_voltage Group” on page 6-16 for additional information.

Drive Current

Output and bidirectional pads in a technology can have different drive-current capabilities. To define the drive current supplied by the pad buffer, use the `drive_current` attribute on an output or bidirectional pad or auxiliary pad pin. The value is in units consistent with the `current_unit` attribute you defined.

Example

```
pin(PAD) {
   direction : output;
   is_pad : true;
   drive_current : 1.0;
}
```

Slew-Rate Control

Slew-rate control limits peak noise and smooths out fast output transitions. Library Compiler implements slew-rate control by using one qualitative attribute—`slew_control`—and the following eight quantitative attributes:

- `rise_current_slope_before_threshold`
- `rise_current_slope_after_threshold`
- `fall_current_slope_before_threshold`
- `fall_current_slope_after_threshold`
- `rise_time_before_threshold`
- `rise_time_after_threshold`
- `fall_time_before_threshold`
- `fall_time_after_threshold`

The `slew_control` attribute accepts one of four possible enumerations: none, low, medium, and high; the default is none. Increasing the slew control level slows down the transition rate. This method is the coarsest way to measure the level of slew-rate control associated with an output pad.

You can define slew-rate control for more detail. Library Compiler accepts the eight quantitative attributes to define a two-piece approximation of the current versus time characteristics (dI/dT) of the pad.

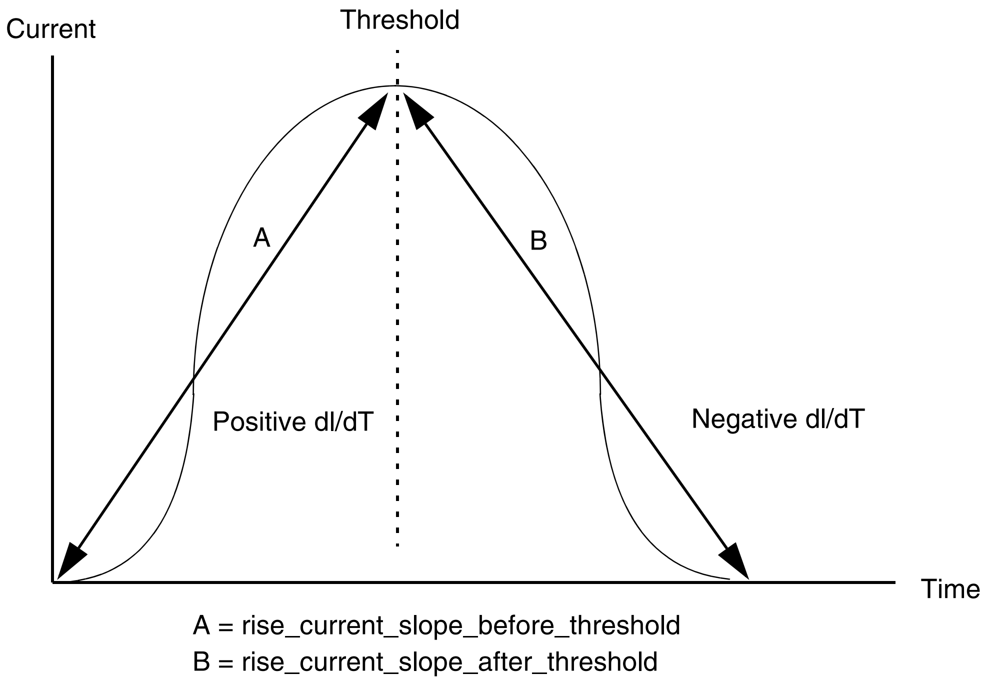
These attributes characterize the dI/dT behavior of an output pad on rising and falling transitions. In addition, two attributes approximate the difference in dI/dT before and after the threshold is reached.

Finally, the attributes define the time intervals for rising and falling transitions from start to threshold (before) and from threshold to finish (after).

For more information about slew-rate control attributes, see “Slew-Rate Control Attributes” on page 7-54.

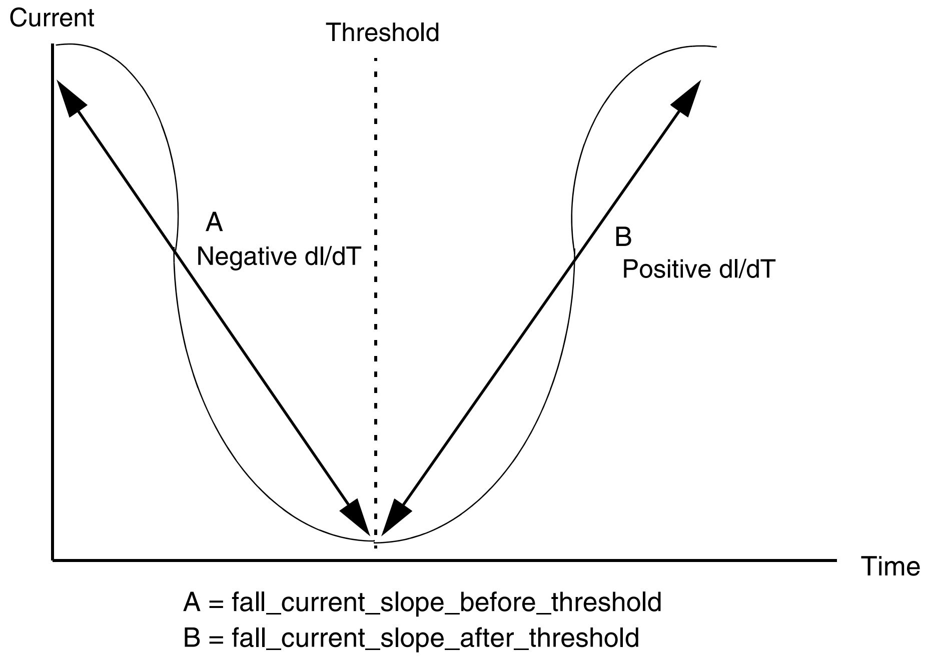
Figure 9-2 depicts the `rise_current_slope` attributes. The A value is a positive number representing a linear approximation of the change of current as a function of time from the beginning of the rising transition to the threshold point. The B value is a negative number representing a linear approximation of the current change over time from the point at which the rising transition reaches the threshold to the end of the transition.

Figure 9-2 Slew-Rate Attributes—Rising Transitions


Current
Threshold
A
B
Positive dI/dT
Negative dI/dT
Time
A = rise_current_slope_before_threshold
B = rise_current_slope_after_threshold

For falling transitions, the graph is reversed: A is negative and B is positive, as shown in Figure 9-3.

Figure 9-3 Slew-Rate Attributes—Falling Transitions


Current
Threshold
A
Negative dI/dT
B
Positive dI/dT
Time
A = fall_current_slope_before_threshold
B = fall_current_slope_after_threshold

Example 9-1 shows the slew-rate control attributes:

Example 9-1 Slew-Rate Control Attributes

```
pin(PAD) {
   is_pad : true;
   direction : output;
   output_voltage : GENERAL;
   slew_control : high;
   rise_current_slope_before_threshold : 0.18;
   rise_time_before_threshold : 0.8;
   rise_current_slope_after_threshold : -0.09;
   rise_time_after_threshold : 2.4;
   fall_current_slope_before_threshold : -0.14;
   fall_time_before_threshold : 0.55;
   fall_current_slope_after_threshold : 0.07;
   fall_time_after_threshold : 1.8;
   ...
}
```

Modeling Wire Load for Pads

You can define several `wire_load` groups to contain all the information Design Compiler needs to estimate interconnect wiring delays. Estimated wire loads for pads can be significantly different from those of core cells.

The wire load model for the net connecting an I/O pad to the core needs to be handled separately, because such a net is usually longer than most other nets in the circuit. Some 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. For example, name the group `Pad_WireLoad`. Add a level of hierarchy by placing the pads in the top level and by placing all the core circuitry at a lower level. Then, during the Design Compiler session, you can define the `Pad_WireLoad` model when you compile the hierarchical level containing the pads:

```
dc_shell> set_wire_load Pad_WireLoad
```

A different model is defined for the core hierarchical level.

Programmable Driver Type Support in I/O Pad Cell Models

To support pull-up and pull-down circuit structures, the Liberty models for I/O pad cells support pull-up and pull-down driver information using the `driver_type` attribute with the `pull_up` or `pull_down` values.

Liberty syntax also supports conditional (programmable) pull-up and pull-down driver information for I/O pad cells. The programmable pin syntax has also been extended to other `driver_type` attribute values, such as `bus_hold`, `open_drain`, `open_source`, `resistive`, `resistive_0`, and `resistive_1`.

Syntax

The following syntax supports programmable driver types in I/O pad cell models. Unlike the nonprogrammable driver type support, the programmable driver type support allows you to specify more than one driver type within a pin.

```
pin (<pin_name>) { /* programmable driver type pin */
   …
   pull_up_function : "<function string>";
   pull_down_function : "<function string>";
   bus_hold_function : "<function string>";
   open_drain_function : "<function string>";
   open_source_function : "<function string>";
```

```
        resistive_function : "<function string>";
        resistive_0_function : "<function string>";
        resistive_1_function : "<function string>";
        …
    }
```

Programmable Driver Type Functions

The functions in Table 9-1 have been introduced on top of (as an extension of) the existing `driver_type` attribute to support programmable pins. These driver type functions help model the programmable driver types. The same rules that apply to nonprogrammable driver types also apply to these functions.

Table 9-1 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 9-1 is specified on an inout pin, it is used only for output pins.

The following rules apply to programmable driver type functions (as well as nonprogrammable driver types in I/O pad cell models):

- The attribute can be applied to pad cell only.
- Only the input and inout pin can be specified in the function string.
- The function string is a Boolean function of input pins.

The following rules apply to an inout pin:

- If `pull_up_function` or `pull_down_function` and `open_drain_function` are specified within the same inout pin, `pull_up_function` or `pull_down_function` is used for the input pins.
- If `bus_hold_function` is specified on an inout pin, it is used for input and output pins.

Example

Example 9-2 models a programmable driver type in an I/O pad cell.

Example 9-2 Example of Programmable Driver Type

```
library(cond_pull_updown_example) {
delay_model : table_lookup;

time_unit : 1ns;
voltage_unit : 1V;
capacitive_load_unit (1.0, pf);
current_unit : 1mA;

cell(conditional_PU_PD) {
    dont_touch : true ;
    dont_use : true ;
    pad_cell : true ;
    pin(IO) {
        drive_current   : 1 ;
        min_capacitance : 0.001 ;
        min_transition  : 0.0008 ;
        is_pad          : true ;
        direction       : inout ;
        max_capacitance : 30 ;
        max_fanout      : 2644 ;
        function        : "(A*ETM')+(TA*ETM)" ;
        three_state     : "(TEN*ETM')+(EN*ETM)" ;
                   pull_up_function    : "(!P1 * !P2)" ;
        pull_down_function : "( P1 *  P2)" ;
        capacitance     : 2.06649 ;
        timing() {
            related_pin : "IO A ETM TEN TA" ;
            cell_rise(scalar) {
                values("0" ) ;
            }
            rise_transition(scalar) {
                values("0" ) ;
            }
            cell_fall(scalar) {
                values("0" ) ;
            }
            fall_transition(scalar) {
                values("0" ) ;
            }
         }
        timing() {

            timing_type : three_state_disable;
            related_pin : "EN ETM TEN" ;
            cell_rise(scalar) {
                values("0" ) ;
            }
            rise_transition(scalar) {
                values("0" ) ;
            }
            cell_fall(scalar) {
                values("0" ) ;
            }
            fall_transition(scalar) {
                values("0" ) ;
            }
         }

  pin(ZI) {
    direction : output;
        function        : "IO" ;
        timing() {
            related_pin : "IO" ;
            cell_rise(scalar) {
                values("0" ) ;
            }
            rise_transition(scalar) {
                values("0" ) ;
            }
            cell_fall(scalar) {
                values("0" ) ;
            }
            fall_transition(scalar) {
                values("0" ) ;
            }
         }
  }
  pin(A) {
    direction : input;
    capacitance : 1.0;
  }
  pin(EN) {
    direction : input;
    capacitance : 1.0;
  }
  pin(TA) {
    direction : input;
    capacitance : 1.0;
  }
  pin(TEN) {
    direction : input;
    capacitance : 1.0;
}
  pin(ETM) {
    direction : input;
    capacitance : 1.0;
  }
  pin(P1) {
    direction : input;
    capacitance : 1.0;
  }
  pin(P2) {
    direction : input;
    capacitance : 1.0;
  }
} /* End cell conditional_PU_PD */
} /* End Library */
```

Library Checking Rules

Library Compiler checks the programmable driver type syntax and quits with an error if any of the following conditions are not met:

- The function string of any programmable function must be mutually exclusive within a pin.
- The function string of any programmable function must only contain input or inout pins.
- Either a nonprogrammable driver type using the old `driver_type` attribute or a programmable driver type function can be specified, but not both.
- The cell where the programmable driver type functions are defined must be a pad cell.
- The pin on which `pull_up_function` or `pull_down_function` is defined must be an input, output, or inout pin.
- The pin on which a `bus_hold_function` is defined must be an inout pin.
- The pin on which `open_drain_function`, `open_source_function`, `resistive_function`, `resistive_0_function`, or `resistive_1_function` is defined must be an output or inout pin.

Reporting Pad Information

The `report_lib` command prints information on the `voltage` groups and units defined in your library, as shown in Example 9-3. You can print a report for the library as a whole or for only the cells you select. See "Reporting Library Information" on page 3-25 for more information on the `report_lib` command.

Example 9-3 Example Library Report

```
****************************************
Report : library
Library: testpad
Version: 2000.05
Date   : Wed August 11 16:10:03 2005
****************************************

Library Type            : Technology
Tool Created            : 2005.05
Date Created            : Wed August 11 20:05:36 2005
Library Version         : 2.110100
Time Unit               : 1ns
Capacitive Load Unit    : 0.100000ff
Pulling Resistance Unit : 1kilo-ohm
Voltage Unit            : 1V
Current Unit            : 1uA
Bus Naming Style        : %s[%d] (default)
...
Input Voltages:

    Name                Vil         Vih         Vimin       Vimax
    -----------------------------------------------------------------
    TTL                 0.80        2.00        -0.30       VDD + 0.300
    TTL_SCHMITT         0.80        2.00        -0.30       VDD + 0.300
    CMOS                1.50        3.50        -0.30       VDD + 0.300
    CMOS_SCHMITT        1.00        4.00        -0.30       VDD + 0.300


Output Voltages:

    Name                Vol         Voh         Vomin       Vomax
    -----------------------------------------------------------------
    GENERAL             0.40        2.40        -0.30       VDD + 0.300

Wire Loading Model:

    No wire loading specified.

Wire Loading Model Mode: top.
```

Pad Cell Examples

These are examples of input, clock, output, and bidirectional pad cells.

Input Pads

The input pad definition in Example 9-4 represents a standard input buffer, and Example 9-5 lists an input buffer with hysteresis. Example 9-6 shows an input clock buffer.

Example 9-4 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 9-5 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 9-6 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";
         }
      }
   }
```

Output Pads

The output pad definition in Example 9-7 represents a standard output buffer.

Example 9-7 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";
         }
      }
   }
```

Bidirectional Pad

Example 9-8 shows a bidirectional pad cell with three-state enable.

Example 9-8 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";
            }
         }
      }
```

Components for Multicell Pads

Example 9-9 shows a library that allows multicell pad configurations.

Example 9-9 Multicell Pad Components

```
library (multicell_pad_example) {
   time_unit : "1ns";
   voltage_unit : "1V";
   current_unit : "1mA";
   pulling_resistance_unit : "1kohm";
   default_connection_class : "some_default";
```

```
 cell(INPAD) {
    area : 1;
    pad_cell : true;
    pin( PAD ) {
       direction : input;
       capacitance : 1.0;
       is_pad : true ;
    }
    pin( ToInBuf ) {
       direction : output;
       multicell_pad_pin : true;
       connection_class : "INPAD_NETWORK";
       timing() {
 }
 cell(OUTPAD) {
    area : 1;
    pad_cell : true;
    pin( FromOutBuf ) {
       direction : input;
       capacitance : 1.0;
       multicell_pad_pin : true;
       connection_class : "OUTPAD_NETWORK";
    }
    pin(PAD ) {
       direction : output;
       is_pad : true;
       drive_current : 1.0;
    }
 }
cell( BIDIPAD ) {
  area : 1;
  pad_cell : true;
  pin( FromOutBuf ) {
    direction : input;
    capacitance : 1.0;
    multicell_pad_pin : true;
    connection_class : "OUTPAD_NETWORK";
  }
  pin( ToInBuf ) {
    direction : output;
    multicell_pad_pin : true;
    connection_class : "INPAD_NETWORK";
  }
  pin( Control ) {
    direction : input;
    capacitance : 1.0;
  }
  pin( PAD ) {
    direction : inout;
    is_pad : true;
    drive_current : 1.0;
  }
}
```

```
    cell( INBUF ) {
      area : 1;
      auxiliary_pad_cell : true;
      pin( From_In_Pad ) {
        direction : input;
        capacitance : 1.0;
        multicell_pad_pin : true;
        connection_class : "INPAD_NETWORK";
      }
      pin( ToCore ) {
        direction : output;
      }
    }
    cell( OUTBUF ) {
      area : 1;
      auxiliary_pad_cell : true;
      pin( To_Out_Pad ) {
        direction : output;
        multicell_pad_pin : true;
        connection_class : "OUTPAD_NETWORK";
      }
      pin( FromCore ) {
        capacitance : 1.0;
        direction : input;
      }
    }
    cell( PULLUP ) {
      area : 1;
      auxiliary_pad_cell : true;
      pin( PULLING_PIN ) {
        direction : output;
        driver_type : pull_up;
        multicell_pad_pin : true;
        connection_class : "INPAD_NETWORK";
        pulling_resistance : 1000; /* 1000 ohms */
      }
    }
    cell( PULLDOWN ) {
      area : 1;
      auxiliary_pad_cell : true;
      pin( PULLING_PIN ) {
        direction : output;
        driver_type : pull_down;
        multicell_pad_pin : true;
        connection_class : "INPAD_NETWORK";
        pulling_resistance : 1000; /* 1000 ohms */
      }
    }
  }
```

Cell with contention_condition and x_function

Example 9-10 shows a cell with the `contention_condition` attribute, which specifies contention-causing conditions and the `x_function` attribute, which describes the X behavior of the pin. See “contention_condition Attribute” on page 7-7 and the “x_function Attribute” description in “Describing Clock Pin Functions” in Chapter 7 for more information about these attributes.

Note:
DFT Compiler uses both the contention_condition and `x_function` attributes. Formality uses the `x_function` attribute, but not the `contention_condition` attribute.

Example 9-10 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";
      }
    }
  }
```

10

Defining Test Cells

DFT Compiler from Synopsys is a synthesis tool featuring a design-for-test strategy. This tool facilitates the design of highly testable circuits with minimal speed and area overheads, and generates an associated set of test vectors automatically. DFT Compiler can use either a full-scan or a partial-scan methodology to add scan cells to designs and help make designs controllable and observable.

To support DFT Compiler, 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
- Verifying Scan Cell Models
- Using Sequential Mapping-Based Scan Insertion
- Scan Cell Modeling Examples

Describing a Scan Cell

Only MUXed flip-flop, clocked scan, and level-sensitive scan device (LSSD) cells that fit one of the supported methodologies can be specified as scan cells in a library. To specify a cell as a scan cell, add the `test_cell` group to the cell description.

Only the nontest mode function of a scan cell is modeled in the `test_cell` group. The nontest operation is described by its `ff`, `ff_bank`, `latch`, or `latch_bank` declaration and `pin` function attributes. This nontest behavior determines which cells can be replaced when DFT Compiler adds scan circuitry to a design in response to the `insert_dft` command (see Figure 10-1).

Note:
For a description of `pin` groups, see "Describing a Flip-Flop" on page 8-3. The `pin` group restrictions are described in "Pins in the test_cell Group" on page 10-3. See Chapter 8, "Defining Sequential Cells," for details on `ff`, `ff_bank`, `latch`, and `latch_bank` group statements and their attributes.

DFT Compiler has two modes in which it replaces a cell with its equivalent scan cell:

- Identical function-based scan insertion
- Sequential mapping-based scan insertion

The identical function-based scan insertion method is faster but more restrictive. This method exactly matches the functional description as well as the pins when substituting scan cells.

Note:
Make sure your libraries always support the identical function-based scan procedure.

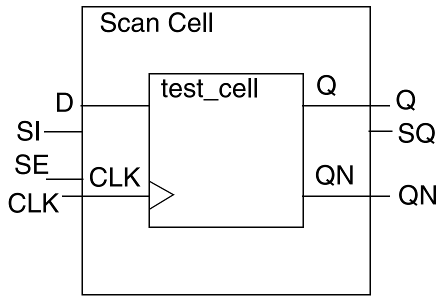
The sequential mapping-based scan insertion method is more general. It is described in "Using Sequential Mapping-Based Scan Insertion" on page 10-16. Sequential mapping-based scan insertion currently supports only cells that are edge-triggered in nontest mode.

Note:
Set the value of the `read_translate_msff` environment variable to true so that DFT Compiler correctly recognizes master-slave latches. If the variable is not set to `true`, DFT Compiler treats these cells as master-slave flip-flop pairs rather than as latch pairs.

test_cell Group

The `test_cell` group defines only the nontest mode function of a scan cell. Figure 10-1 illustrates the relationship between a `test_cell` group and the scan cell in which it is defined.

Figure 10-1 Scan Cell With test_cell Group



There are two important points to remember when defining a scan cell such as the one Figure 10-1 shows:

- Pin names in the scan cell and the test cell must match.
- The scan cell and the test cell must contain the same functional outputs.

Following is the syntax of a `test_cell` group that contains pins:

Syntax

```
library (lib_name) {cell (cell_name) {test_cell () {... test cell
description ...pin ( name ) {... pin description ...}pin ( name ) {...
pin description ...}}}
```

You do not need to give the `test_cell` group a name, because the test cell takes the cell name of the cell being defined. The `test_cell` group can contain `ff`, `ff_bank`, `latch`, or `latch_bank` group statements and `pin` groups.

Pins in the test_cell Group

Both test pins and nontest pins can appear in `pin` groups within a `test_cell` group. These groups are like `pin` groups in a `cell` group but must adhere to the following rules:

- Each pin defined in the `cell` group must have a corresponding pin defined at the `test_cell` group level with the same name.

- The pin group can contain only direction, function, test_output_only, and signal_type attribute statements. The pin group cannot contain timing, capacitance, fanout, or load information.
- The function attributes can reflect only the nontest behavior of the cell.
- Input pins must be referenced in an ff, ff_bank, latch, or latch_bank statement or have a signal_type attribute assigned to them.
- An output pin must have either a function attribute, a signal_type attribute, or both.

Example 10-1 shows a library model for a multiplexed D flip-flop scan cell with multiple test_cell groups.

Example 10-1 Multiplexed D Flip-Flop Scan Cell

```
cell(SDFF1) {
  area : 9;
  pin(D) {
    direction : input;
    capacitance : 1;
    timing() {
      timing_type : setup_rising;
      intrinsic_rise : 1.0;
      intrinsic_fall : 1.0;
      related_pin : "CLK";
    }
    timing() {
      timing_type : hold_rising;
      intrinsic_rise : 1.0;
      intrinsic_fall : 1.0;
      related_pin : "CLK";
    }
  }
  pin(CLK) {
    direction : input;
    capacitance : 1;
  }
  pin(SI) {
    direction : input;
    capacitance : 1;
    prefer_tied : "0";
    timing() {
      timing_type : setup_rising;
      intrinsic_rise : 1.0;
      intrinsic_fall : 1.0;
      related_pin : "CLK";
    }
    timing() {
      timing_type : hold_rising;
      intrinsic_rise : 1.0;
      intrinsic_fall : 1.0;
      related_pin : "CLK";
    }
  }
  pin(SE) {
    direction : input;
    capacitance : 1;
    timing() {
      timing_type : setup_rising;
      intrinsic_rise : 1.0;
      intrinsic_fall : 1.0;
      related_pin : "CLK";
    }
    timing() {
      timing_type : hold_rising;
      intrinsic_rise : 1.0;
      intrinsic_fall : 1.0;
      related_pin : "CLK";
    }
  }
  ff("IQ","IQN") {
    next_state : "(D & !SE) | (SI & SE)";
    clocked_on : "CLK";
  }
  pin(Q) {
    direction : output;
    function : "IQ"
    timing() {
      timing_type : rising_edge;
      intrinsic_rise : 1.0;
      intrinsic_fall : 1.0;
      rise_resistance : 0.1;
      fall_resistance : 0.1;
      related_pin : "CLK";
    }
  }
  pin(QN) {
    direction : output;
    function : "IQN"
    timing() {
      timing_type : rising_edge;
      intrinsic_rise : 1.0;
      intrinsic_fall : 1.0;
      rise_resistance : 0.1;
      fall_resistance : 0.1;
      related_pin : "CLK";
    }
  }
      /* first test_cell group defines nontest
         behavior with both Q and QN */
  test_cell() {
    pin(D,CLK) {
      direction : input;
    }
    pin(SI) {
      direction : input;
      signal_type : "test_scan_in";
    }
    pin(SE) {
      direction : input;
      signal_type : "test_scan_enable";
    }
    ff ("IQ","IQN") {
      next_state : "D";
      clocked_on : "CLK";
    }
    pin(Q) {
      direction : output;
      function : "IQ";
      signal_type : "test_scan_out";
    }
    pin(QN) {
      direction : output;
      function : "IQN";
      signal_type : "test_scan_out_inverted";
    }
  }
       /* second test_cell group defines nontest
          behavior with only Q  */
  test_cell() {
    pin(D,CLK) {
      direction : input;
    }
    pin(SI) {
      direction : input;
      signal_type : "test_scan_in";
    }
   pin(SE) {
      direction : input;
      signal_type : "test_scan_enable";
    }
    ff ("IQ","IQN") {
      next_state : "D";
      clocked_on : "CLK";
    }
    pin(Q) {
      direction : output;
      function : "IQ";
      signal_type : "test_scan_out";
    }
    pin(QN) {
      direction : output;
      /* notice no function attribute for QN pin */
      signal_type : "test_scan_out_inverted";
    }
  }
}
```

test_output_only Attribute

This attribute indicates that an output port is set for test only (as opposed to function only, or function and test).

Syntax

```
test_output_only : true | false ;
```

When you use statetable format to describe the functionality of a scan cell, you must declare the output pin as set for test only by setting the `test_output_only` attribute to `true`.

Example

```
test_output_only : true ;
```

signal_type Attribute

Defined in a `test_cell` group, this attribute identifies the type of test pin.

If you use an input pin in both test and nontest modes (such as the clock input in the multiplexed flip-flop methodology), do not include a `signal_type` statement for that pin in the `test_cell` pin definition.

If you use an input pin of a scan cell only in nontest mode and that pin does not exist on the equivalent non scan cell that it replaces, then you must include a `signal_type` statement for that pin in the `test_cell` pin definition.

If you use an output pin in non test mode, it must have a `function` statement. The `signal_type` attribute is used to identify an output pin as a scan-out pin. In a `test_cell` group, the `pin` group for an output pin can contain a `function` statement, a `signal_type` attribute, or both.

Example

```
signal_type : test_scan_in ;
```

For more information on the possible values for the `signal_type` attribute, see the “cell and model Group Description and Syntax” chapter in *Library Compiler Technology and Symbol Libraries Reference Manual*.

Note:
You do not have to define a function or `signal_type` attribute in the `pin` group if the pin has been defined in a previous `test_cell` group for the same cell.

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 10-2 illustrates a 2-bit piece of a 4-bit D flip-flop with scan and enable.

Figure 10-2 Multibit Scan Cell With test_cell Group

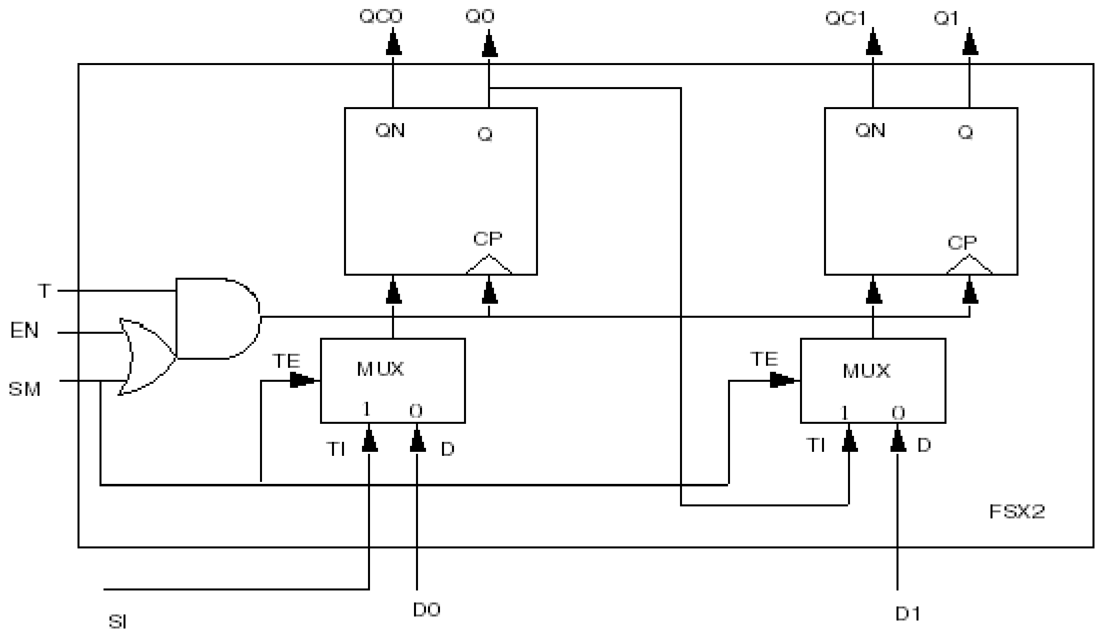
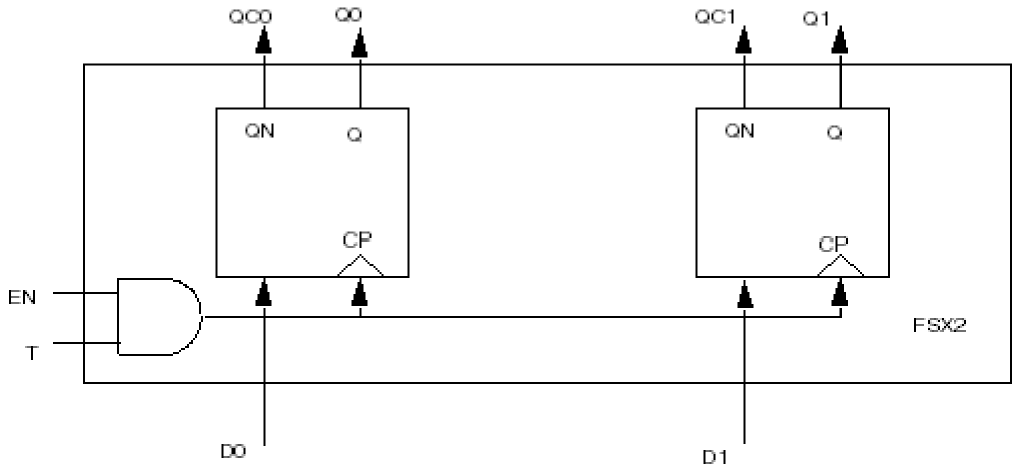


Figure 10-3 shows the `test_cell` group (the nontest mode) of the cell in Figure 10-2.

Figure 10-3 test_cell Group of Multibit Scan Cell



To create a multibit scan cell, use `statetable` in the full cell description, as shown in Example 10-2.

Example 10-2 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 10-2, `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 10-3 shows the test_cell description for multibit pins, using the `ff_bank` and `bundle` group attributes.

Example 10-3 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 10-7 on page 10-19.

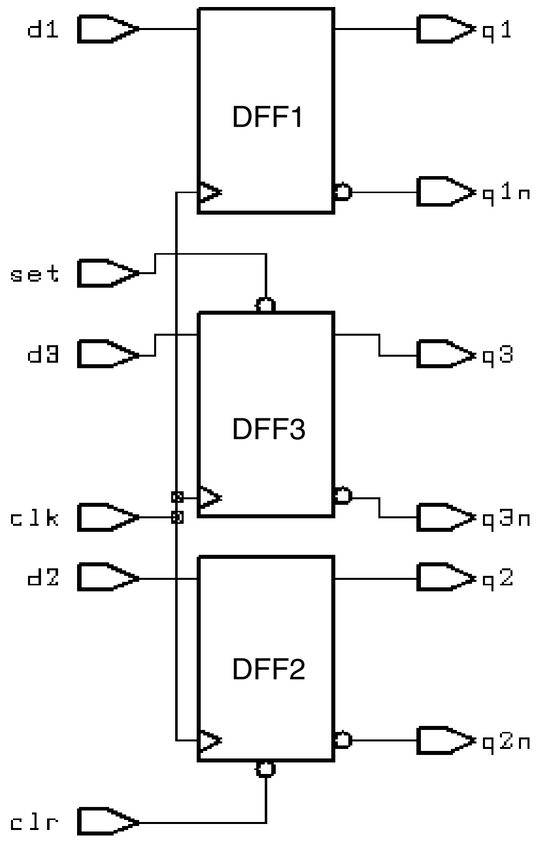
Verifying Scan Cell Models

To verify that the scan cells have been correctly modeled for scan substitution, use the `dft_drc` and `insert_dft` commands described in the DFT Compiler documentation. To verify scan cell models, check the function and timing of the cells as described in the *Library Compiler Technology and Symbol Libraries Reference Manual*. After checking the cells, you can create test cases and verify the scannable equivalents, as described in the following sections.

Creating Test Cases

For each test method represented in the technology library, create a test circuit that includes a single instance of each nonscan cell that has a scannable equivalent in that method. Figure 10-4 shows an example of a test circuit for the multiplexed flip-flop method in which the technology library contains three nonscan cells: DFF1 (simple D flip-flop), DFF2 (D flip-flop with asynchronous clear), and DFF3 (D flip-flop with asynchronous set).

Figure 10-4 Example Test Circuit for Multiplexed Flip-Flop



Verifying Scannable Equivalents

Verify that all scan/nonscan pairs in the technology library are modeled so that DFT Compiler can recognize their equivalency. If DFT Compiler cannot find a scannable equivalent for a sequential element, that cell is not included in the scan chain, which results in lowered fault coverage.

Do not use the statetable format in a `test_cell` group, because Library Compiler does not issue warnings or errors and does not verify the model. Also, DFT Compiler cannot use the statetable model for scan insertion or fault simulation, and DFT Compiler requires the ff/latch format for scan substitution and fault simulation.

To verify the scan cell models for equivalency,

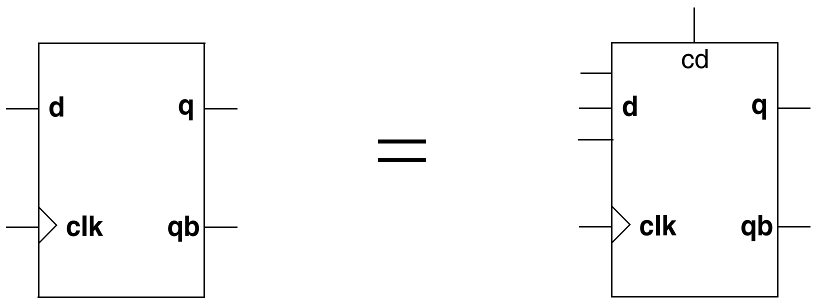
1. Select the scan implementation style

2. Verify the existence of scannable equivalents
3. Verify substitution and routing

The following sections show the recommended sequence of DFT Compiler commands that you use to verify the scan cell models for equivalency.

Figure 10-5 shows a nonscan/scan equivalent pair in which the scan cell has an asynchronous pin and the nonscan cell does not.

Figure 10-5 Scannable Equivalent Pair (Scan Cell Has Asynchronous Pin)



Nonscan Cell Function

```
ff("IO","IQN")   {
  next_state   : "d" ;
  clocked_on   : "clk" ;
}
```

test_call Function

```
ff("IO","IQN")   {
  next_state   : "d" ;
  clocked_on   : "clk" ;
  clear        : "cd" ;
}
```

Selecting the Scan Implementation

Declare the scan style you want to use with your design:

```
dc_shell> set_scan_configuration -style
```

Verifying the Existence of Scannable Equivalents

Use the `dft_drc` command to verify that a scannable equivalent exists for each cell in your test circuit:

```
dc_shell> dft_drc
```

Example 10-4 shows the results of the `dft_drc` command when the technology library does not contain a scannable equivalent for the cells in the test circuit.

Example 10-4 dft_drc Results—Scannable Equivalents Do Not Exist

```
dc_shell> dft_drc
```

```
 Loading test protocol
  ...basic checks...
  ...basic sequential cell checks...
        ...checking for scan equivalents...
  ...checking vector rules...
  ...checking pre-dft rules...

-----------------------------------------------------------------

Begin Modeling violations...

Warning: No scan equivalent exists for cell u0/reg0 (FF0). (TEST-120)
Information: There are 1132 other cells with the same violation. (TEST-171)

Modeling violations completed...
-----------------------------------------------------------------
…

-----------------------------------------------------------------
  DRC Report

  Total violations: 12293

-----------------------------------------------------------------
…
1133 MODELING VIOLATIONS
  1133 Cell has no scan equivalent violations (TEST-120)
```

Example 10-5 shows the results of the `dft_drc` command when the technology library contains a scannable equivalent for each cell in the test circuit.

Example 10-5 dft_drc Results—Scannable Equivalents Exist

```
dc_shell> dft_drc

         Loading target library ‘lsi_10k’
         Loading design ‘test’

Information: Starting test design rule checking. (TEST-222)
          ...full scan rules enabled...
          ...basic checks...
          ...basic sequential cell checks...
          ...checking for combinational feedback loops...
          ...inferring test protocol...
Information: Inferred system/test clock port clk (45.0, 55.0). (TEST-260)
Information: Inferred active low asynchronous control port clr. (TEST-261)
          ...simulating parallel vector...
          ...simulating parallel vector...
          ...simulating serial scan-in...
          ...simulating parallel vector...
          ...binding scan-in state...
          ...simulating parallel vector...
          ...simulating capture clock rising edge at port clk...
          ...checking clock used as data...
          ...simulating data capture...
          ...checking for illegal paths active on this edge...
          ...simulating capture clock falling edge at port clk...
          ...simulating data capture...
```

```
           ...creating capture clock groups...
Information: Inferred capture clock group : clk. (TEST-262)
           ...binding scan-out state...
           ...simulating serial scan-out...
           ...simulating parallel vector...
Information: There are 3 scannable sequential cells. (TEST-295)
Information: Test design rule checking completed with 0 warning(s) and 0 error(s).
(TEST-123)
```

If the `dft_drc` command reports that no scannable equivalent exists for a cell that does have a scannable equivalent in the technology library, the scan cell is probably not modeled correctly. In rare cases in which the nonscan cell is very complex, it might not be possible to model a scannable equivalent.

Verifying Substitution and Routing

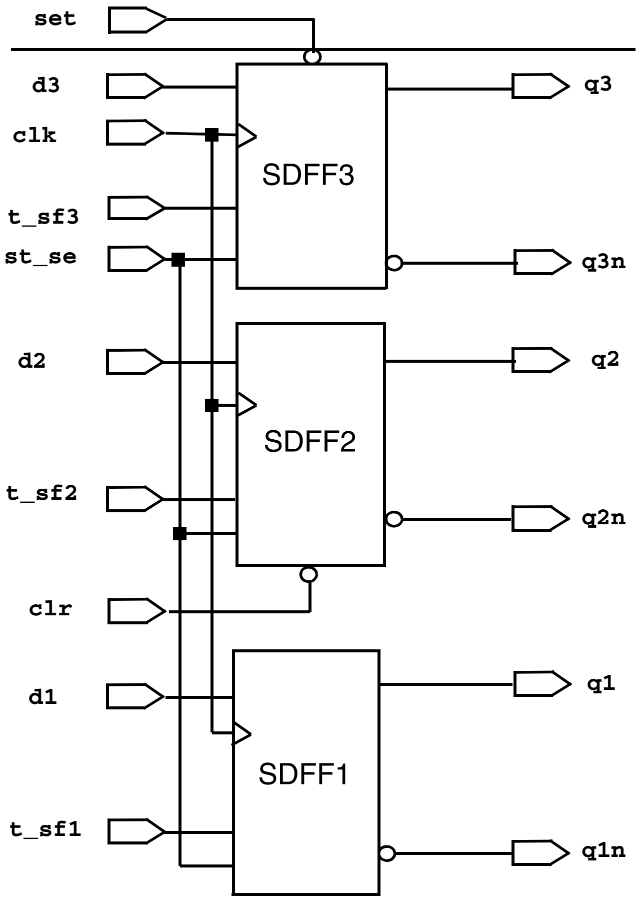
After confirming that all sequential cells have scannable equivalents, use the `insert_dft` command to insert and route the scan chain or chains. View the resulting schematic to confirm the correct substitution and routing.

Make each cell in the test circuit a separate scan chain to simplify visual verification:

```
dc_shell> insert_dft
```

Figure 10-6 shows the scannable circuit that results from the `insert_dft` command.

Figure 10-6 Scannable Test Circuit



Using Sequential Mapping-Based Scan Insertion

Library Compiler can do sequential mapping-based scan insertion; see the *DFT Compiler Scan Synthesis User Guide* for details. Scan insertion uses the sequential mapping algorithms of Design Compiler to find scan equivalent cells. This procedure can be called for any sequential cell that is edge-triggered in normal mode operation.

The sequential mapping-based scan insertion procedure cannot support all scan cells. It can support a cell if at least one test_cell in the scan cell has these characteristics:

- Shows nontest mode operation as edge-triggered
- Has a `function` statement for all outputs

Most scan cells that can use sequential mapping-based scan insertion can also use identical function-based scan insertion. Although the identical function-based scan insertion does not work in some cases, it is generally faster than sequential mapping-based scan insertion; therefore, you should use the identical function-based scan insertion where possible.

Sequential mapping-based scan insertion and identical function-based scan insertion have contradictory requirements for scan cells with dedicated scan outputs. Identical function-based scan insertion requires that dedicated scan outputs do not have function statements, even if it is possible to describe them. Such outputs can have only `signal_type` attributes. Sequential mapping-based scan insertion requires that dedicated scan outputs have a `function` statement as well as a `signal_type` attribute. These conflicting requirements can be satisfied by writing two test cells, as shown in the following example.

Example

To support the identical function procedure in the first test cell, use these statements for dedicated scan output:

```
pin (SO) {
   direction : output;
   signal_type : "test_scan_out";
}
```

To support the sequential mapping procedure in the second test cell, use these statements for dedicated scan output:

```
pin (SO) {
   direction : output;
   signal_type : "test_scan_out";
   function : "IQ";
   test_output_only : true;
}
```

The `function` statement in the second example meets the sequential mapping procedure's requirement that all outputs have a function statement, while the `test_output_only` attribute prevents the output pin from being used as a nontest output.

Scan Cell Modeling Examples

This section contains modeling examples for these test cells:

- Simple multiplexed D flip-flop
- Multibit cells with multiplexed D flip-flop and enable
- LSSD (level-sensitive scan design) scan cell

- Clocked-scan test cell
- Scan D flip-flop with auxiliary clock

Each example contains a complete cell description.

Simple Multiplexed D Flip-Flop

Example 10-6 shows how to model a simple multiplexed D flip-flop test cell.

Example 10-6 Simple Multiplexed D Flip-Flop Scan Cell

```
cell(SDFF1) {
   area : 9;
   pin(D) {
      direction : input;
      capacitance : 1;
      timing() {...}
   }
   pin(CP) {
      direction : input;
      capacitance : 1;
      timing() {...}
   }
   pin(TI) {
      direction : input;
      capacitance : 1;
      timing() {...}
   }
   pin(TE) {
    direction : input;
    capacitance : 2;
    timing() {...}
   }
   ff(IQ,IQN) {
    /* model full behavior (if possible): */
    next_state : "D TE’ + TI TE ";
    clocked_on : "CP";
   }
   pin(Q) {
    direction : output;
    function : "IQ";
    timing() {...}
   }
   pin(QN) {
    direction : output;
    function : "IQN";
    timing() {...}
   }
   test_cell() {
    pin(D) {
         direction : input
```

```
            }
            pin(CP){
               direction : input
            }
            pin(TI) {
               direction : input;
               signal_type : test_scan_in;
            }
            pin(TE) {
               direction : input;
               signal_type : test_scan_enable;
            }
            ff(IQ,IQN) {
               /* just model nontest operation behavior */
               next_state : "D";
               clocked_on : "CP";
          }
          pin(Q) {
               direction : output;
               function : "IQ";
               signal_type : test_scan_out;
          }
            pin(QN) {
               direction : output;
               function : "IQN";
               signal_type : test_scan_out_inverted;
            }
         }
      }
```

Multibit Cells With Multiplexed D Flip-Flop and Enable

Example 10-7 contains a complete description of multibit scan cells.

Example 10-7 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";
    }
  }
}
```

LSSD Scan Cell

Example 10-8 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 10-8 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";
    }
  }
}
```

Clocked-Scan Test Cell

Example 10-9 shows how to model a level-sensitive latch with separate scan clocking. This example shows the scan cell used in clocked-scan implementation.

Example 10-9 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";
      }
    }
  }
```

Scan D Flip-Flop With Auxiliary Clock

Example 10-10 shows how to model a scan D flip-flop with one input and an auxiliary clock.

Example 10-10 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";
        }
      }
    }
```

11 Modeling FPGA Libraries

This chapter describes how to model FPGA libraries using the Synopsys FPGA library format in a .lib file.

Modeling libraries using the Synopsys FPGA library format involves the following major tasks:

- Specifying the Technology
- Specifying FPGA Part Device Information
- Specifying I/O Components
- Specifying Default Part Values
- Specifying Cell Resources
- Specifying Cell Characteristics
- Specifying Built-in Pin Pads
- Example .lib File

Specifying the Technology

You use the `technology` and the `fpga_technology` attributes to specify the FPGA technology. The `technology` attribute is the first statement at the library level, as shown in Example 11-1.

Example 11-1 Specifying the Technology

```
library ("my_lib1") {
   technology ("fpga") ;
   ...
   fpga_technology : "mylib_technology" ;
   ...
   library description
   ...
}
```

Specifying FPGA Part Device Information

You can use the Synopsys standard liberty format (.lib) to convey information about the various devices that comprise an FPGA family. The particular information that you can convey about each device includes:

- The number of logic-block rows
- The number of logic-block columns
- The number of flip flops
- The number of block-select RAMS
- The number of pins
- The various speed grades and step levels and associations of speed grades with step levels
- The resource constraints
- The number of lookup tables that make up a configurable logic block

To convey this information, you represent each FPGA device as a `part` group in the .lib format. For a description of the `part` group syntax and the .lib format, see the *Library Compiler Technology and Symbol Libraries Reference Manual*.

Example

Example 11-2 shows a `part` group in the .lib format

Example 11-2 A part Group in .lib Format

```
library (example) {
   technology(fpga);
   revision : "v1.0.0";
   ...
   fpga_technology : "my_technology" ;
   ...
   part(1) {
      default_step_level : "STEP0" ;
      num_rows : 20;
      num_cols : 30;
      num_ffs : 2760;
      num_blockrams : 10;
      num_luts : 100 ;
      pin_count : 94;
      valid_speed_grades ("-6", "-5", "-4");
      valid_step_levels ( "STEP0", STEP1", STEP2" ) ;
      max_count(BUFGTS, 4);
      speed_grade (associated_speed_grade) {
         step_level (associated_step_level) :
      }
   }
   ....
}
```

Specifying I/O Components

You can use the `fpga_domain_style` attribute to describe attribute based timing calculations for various library components such that you can define all the speed grades in one library.

The `calc_mode` attribute references an `fpga_domain_style` value from an associated `domain` group in a polynomial table.

You can define `fpga_domain_style` attributes at the library, cell, and timing levels of your library. In general, the library uses the value of the last attribute read. If you do not specify an `fpga_domain_style` attribute for a cell, the default is used, if it exists.

Example 11-3 shows an `fpga_domain_style` attribute defining a default for speed grade only.

Example 11-3 Specifying I/O Components

```
library (example) {
   technology(fpga);
   revision : "v1.0.0";
   ...
   fpga_domain_style : "speed" ;
   ...
   poly_template(my_template) {
      variables () ;
      variable_1_range () ;
      domain(speed:::) {
         calc_mode : "speed" ;
      }
      ...
   }
   cell(my_cell) {
      fpga_domain_style : "speed:io_standard:slew:drive" ;
      area : 0 ;
      preferred : true
      pad_cell: true ;
      pin(O) {
         direction : output ;
         function : "I" ;
         is_pad : true ;
         ...
         timing () {
            rise_propagation(my_template) {
               domain (speed4:::) {
                  orders("0") ;
                  coefs("1.123");
               ...
            }
            fall_propagation(my_template)
               domain(speed4:::) {
                  orders("0") ;
                  coefs();
               }
               ...
            }
            rise_transition(my_template) {
                  orders("0") ;
                  coefs () ;
            }
            fall_transition(my_template) {
                  orders("0") ;
                  coefs () ;
            }
            related_pin : "I" ;
         }
      }
      pin (I) {
         direction : input ;
         capacitance : INPUT_CAP ;
```

```
            }
        }
        ...
    }
```

Specifying Default Part Values

In the .lib format you can use the `default_part` complex attribute at the library level to specify a default part and a default speed for the design.

Example 11-4 shows a `default_part` attribute in the .lib format.

Example 11-4 A default_part Group in .lib Format

```
library (example) {
   technology(fpga);
   ...
   fpga_technology : "my_technology" ;
   ...
   default_part(part100, "-5") ;
   ....
}
```

Specifying Cell Resources

You can use the `resource_usage` attribute at the cell level to specify the name and number of resources used by the cell. You can specify multiple `resource_usage` attributes with different resource names. A resource name must match a resource name declared in a `max_count` attribute. For a description of the syntax for these attributes and the `cell` group syntax, see the *Library Compiler Technology and Symbol Libraries Reference Manual*.

Example 11-5 shows `resource_usage` attributes in a `cell` group.

Example 11-5 Defining Cell Resources

```
cell(cell_example) {
   ....
   resource_usage(RES1, 1);
   resource_usage(RES2, 4);
   ...
}
```

Specifying Cell Characteristics

You can use the following five optional attributes in a `cell` group to specify the characteristics of FPGA I/O cells:

- io_type
- slew_type
- drive_type
- edif_name
- base_name

You can apply the attributes as follows.

io_type

Specifies the type of I/O standard the I/O cell uses. Applies to FPGA input cells.

slew_type, drive_type

Specify the slew type and drive power. Apply to output and inout cells. You must specify both attributes.

edif_name

Specifies the EDIF cell name to apply when you complete an EDIF file generation. If this attribute is missing, the default name, *io_cell_name* is applied.

base_name

Specifies the cell name to apply when you complete either VHDl or Verilog file generation. If this attribute is missing, the default name, *io_cell_name* is applied.

Note:
Library Compiler does not perform any checks if two cells have the same base name.

For a description of the syntax for these attributes and the `cell` group syntax, see the *Library Compiler Technology and Symbol Libraries Reference Manual*.

Example 11-6 shows a `cell` group using the optional attributes.

Example 11-6 Cell Group

```
cell (IBUF_LVTTL) {
   ...
   io_type : "LVTTL";
   edif_name : "IBUF";
   base_name " "IBUF" ;
   pin (I) {
      direction : input;
      capacitance : 1.0;
   }
   pin(O) {
      direction : output;
      function : "I";
      timing () {
         related_pin : "I";
         intrinsic_rise : 1.3;
```

```
            intrinsic_fall : 1.2;
         }
      }
   }
   cell (IBUF_PCI_33) {
      ...
      io_type : "PCI_33";
      edif_name : "IBUF";
      base_name " "IBUF" ;
      pin (I) {
         direction : input;
         capacitance : 1.0;
      }
      pin(O) {
         direction : output;
         function : "I";
         timing () {
            related_pin : "I";
            intrinsic_rise : 2.4;
            intrinsic_fall : 2.1;
         }
      }

   }
   cell (OBUF_1) {
      pad_cell : true;
      io_type : "LVTTL";
      slew_type : "fast";
      edif_name : "OBUF";
      base_name " "OBUF" ;
      drive_type : "4";
      pin (I) {
         direction : input;
         capacitance : 1.0;
      }
      pin (O) {
         direction : output;
         function : "I";
         is_pad : true;
         timing () {
            related_pin : "I";
            intrinsic_rise : 1.4;
            intrinsic_fall : 2.5;
            ...
         }
      }
   }
   cell (OBUF_2) {
      pad_cell : true;
      io_type : "LVTTL";
      slew_type : "slow";
      drive_type : "4";
      edif_name : "OBUF";
```

```
        base_name " "OBUF" ;
        pin (I) {
           direction : input;
           capacitance : 0.0;
        }
        pin (O) {
           direction : output;
           function : "I";
           is_pad : true;
           timing () {
              related_pin : "I";
              intrinsic_rise : 1.5;
              intrinsic_fall : 2.7;
              ...
           }
        }
     }
```

Specifying Built-in Pin Pads

The ASIC cores in FPGAs are customized to implement certain functions such as PCI cores and RISC cores. The FPGA vendor provides a black box cell for such cores in the .lib file. Those ASIC cores have two types of I/O pins: those that connect to the FPGA portion of the chip and those that connect beyond the FPGA portion. The latter have built-in pads.

Typically, developers instantiate such cores in their HDL and build the rest of the design around it. The `pad` attribute at the pin level supports this technology flow. Synopsys tools detect such pins when the `has_builtin_pad` attribute is set to true, which prevents the pad-mapper from inserting any pad on the net connecting these pins to the chip's port. For a description of this attribute and the `pin` group syntax, see the *Library Compiler Technology and Symbol Libraries Reference Manual*.

Example 11-7 shows a `has_builtin_pad` attribute in a `pin` group.

Example 11-7 Specifying a Built-in Pad at the pin Level.

```
pin(example) {
   ...
   has_builtin_pad : true;
   ...
}
```

Example .lib File

Example 11-8 A .lib File for an FPGA Library

```
library (example) {
   technology(fpga);
   revision : "v1.0.0" ;
   fpga_technology : "my_fpga" ;
   part(1) {
      valid_speed_grades ("-6", "-5", "-4");
      valid_step_levels ( "STEP0", "STEP1", "STEP2" ) ;
      default_step_level : "STEP0" ;
      num_rows : 20;
      num_cols : 30;
      num_ffs : 2760;
      num_blockrams : 10;
      num_luts : 100 ;
      pin_count : 94;
      max_count(BUFGTS, 4);
   }
   part (2) {
      valid_speed_grades ("-6", "-5", "-4");
      valid_step_levels ( "STEP0", "STEP1", "STEP2" ) ;
      default_step_level : "STEP0" ;
      num_rows : 20;
      num_cols: 30;
      num_ffs : 2760;
      num_blockrams : 10;
      num_luts : 100 ;
      pin_count : 162;
      max_count (BUFGTS, 4);
   }
   part (3) {
      valid_speed_grades ("-6", "-5", "-4");
      valid_step_levels ( "STEP0", "STEP1", "STEP2" ) ;
      default_step_level : "STEP0" ;
      num_rows : 20;
      num_cols: 30;
      num_ffs : 2760;
      num_blockrams : 10;
      num_luts : 100 ;
      pin_count : 98;
      max_count(BUFGTS, 4);
      max_count(BUFGP, 6);
      ......
   }
   part (4) {
      valid_speed_grades ("-6", "-5", "-4");
      valid_step_levels ( "STEP0", "STEP1", "STEP2" ) ;
      default_step_level : "STEP0" ;
      num_rows : 20;
      num_cols: 30;
```

```
            num_ffs : 2760;
            num_blockrams : 10;
            num_luts : 100 ;
            pin_count : 166;

            ......
        }
        default_part(5);
        /* library information for cells */
        ......
    }
```

A

Clock-Gating Integrated Cell Circuits

This appendix contains the following information for the `clock_gating_integrated_cell` attribute:

- A list and description of the options available
- Schematics of the circuits represented by the value options available
- An example of each of the value options available

It includes the following sections:

List and Description of Options

Table A-1 lists and describes the `clock_gating_integrated_cell` attribute values. Specify the `clock_gating_integrated_cell` attribute, as shown:

```
clock_gating_integrated_cell : value ;
```

where *`value`* is a concatenation of up to four strings that describe the cell's function. The valid values are listed in Table A-1. For more information about the cells described in this table, see "Schematics and Examples" on page A-7.

Power Compiler supports these cells as well as the generic integrated clock-gating cells in Table A-2 on page A-5. After you model a cell using the integrated clock-gating syntax and specify a value, Library Compiler derives one of the structures in Table A-1 automatically.

Table A-1 Values for the clock_gating_integrated_cell Attribute

clock_gating_integrated_cell values	**Integrated cell must contain**
latch_posedge	Latch-based gating logic Logic appropriate for rising-edge-triggered registers
latch_posedge_precontrol	Latch-based gating logic Logic appropriate for rising-edge-triggered registers Test-control logic located before the latch
latch_posedge_postcontrol	Latch-based gating logic Logic appropriate for rising-edge-triggered registers Test-control logic located after the latch
latch_posedge_precontrol_obs	Latch-based gating logic Logic appropriate for rising-edge-triggered registers Test-control logic located before the latch Observability logic
latch_posedge_postcontrol_obs	Latch-based gating logic Logic appropriate for rising-edge-triggered registers Test-control logic located after the latch Observability logic
latch_negedge	Latch-based gating logic Logic appropriate for falling-edge-triggered registers

Table A-1 Values for the clock_gating_integrated_cell Attribute (Continued)

clock_gating_integrated_cell values	**Integrated cell must contain**
latch_negedge_precontrol	Latch-based gating logic Logic appropriate for falling-edge-triggered registers Test-control logic located before the latch
latch_negedge_postcontrol	Latch-based gating logic Logic appropriate for falling-edge-triggered registers Test-control logic located after the latch
latch_negedge_precontrol_obs	Latch-based gating logic Logic appropriate for falling-edge-triggered registers Test-control logic located before the latch Observability logic
latch_negedge_postcontrol_obs	Latch-based gating logic Logic appropriate for falling-edge-triggered registers Test-control logic located after the latch Observability logic
none_posedge	Latch and flip-flop free gating logic Logic appropriate for rising-edge-triggered registers
none_posedge_control	Latch and flip-flop free gating logic Logic appropriate for rising-edge-triggered registers Test-control logic (no latch and no flip-flop)
none_posedge_control_obs	Latch and flip-flop free gating logic Logic appropriate for rising-edge-triggered registers Test-control logic (no latch and no flip-flop) Observability logic
none_negedge	Latch and flip-flop free gating logic Logic appropriate for falling-edge-triggered registers

Table A-1 Values for the clock_gating_integrated_cell Attribute (Continued)

clock_gating_integrated_cell values	Integrated cell must contain
none_negedge_control	Latch and flip-flop free gating logic Logic appropriate for falling-edge-triggered registers Test-control logic (no latch and no flip-flop)
none_negedge_control_obs	Latch and flip-flop free gating logic Logic appropriate for falling-edge-triggered registers Test-control logic (no latch and no flip-flop) Observability logic

Table A-2 on page A-5 lists additional latch-based structures that are supported by the `clock_gating_integrated_cell` attribute. The structure of the cells in this table is similar to the cells in Table A-1 on page A-2. However, you must use the following generic integrated clock-gating (ICG) syntax to model these cells:

```
clock_gating_integrated_cell : generic ;
```

When you specify the `generic` value, Library Compiler determines the clock-gating structure based on the `latch` group and the `function` attribute values specified in the cell.

Note:
The generic integrated clock-gating syntax supports only the modeling of cell functional information using the `latch` group and the `function` attribute. If you specify `statetable` or `state_function` to model generic integrated clock-gating cells, Library Compiler generates compilation errors.

Power Compiler supports generic integrated clock-gating cells as well as the cells in Table A-1. After you model a cell using the generic integrated clock-gating syntax, Library Compiler derives one of the structures in Table A-1 or Table A-2 automatically.

To see circuit schematics for the cells listed in Table A-2, see "Generic Integrated Clock-Gating Schematics" on page A-39. (This chapter does not include .lib examples for these cells.)

Table A-2 Additional Latch-Based Structures Supported by the clock_gating_integrated_cell Attribute

Derived cell	Type of gated register	Gating logic used	Control point position	obs port	Active-low enabled
latch_posedgeactivelow	posedge	enl.clk	none	no	yes
latch_posedgeactivelow_ precontrol	posedge	enl.clk	before seq	no	yes
latch_posedgeactivelow_ postcontrol	posedge	enl.clk	after seq	no	yes
latch_posedgeactivelow_ precontrol_obs	posedge	enl.clk	before seq	yes	yes
latch_posedgeactivelow_ postcontrol_obs	posedge	enl.clk	after seq	yes	yes
latch_negedgeactivelow	negedge	!enl + clk	none	no	yes
latch_negedgeactivelow_ precontrol	negedge	!enl + clk	before seq	no	yes
latch_negedgeactivelow_ postcontrol	negedge	!enl + clk	after seq	no	yes
latch_negedgeactivelow_ precontrol_obs	negedge	!enl + clk	before seq	yes	yes
latch_negedgeactivelow_ postcontrol_obs	negedge	!enl + clk	after seq	yes	yes
latch_posedge_invgclk	posedge	!(enl.clk)	none	no	no
latch_posedge_precontrol_ invgclk	posedge	!(enl.clk)	before seq	no	no
latch_posedge_postcontrol_ invgclk	posedge	!(enl.clk)	after seq	no	no
latch_posedge_precontrol_obs_ invgclk	posedge	!(enl.clk)	before seq	yes	no

Table A-2 Additional Latch-Based Structures Supported by the clock_gating_integrated_cell Attribute (Continued)

Derived cell	**Type of gated register**	**Gating logic used**	**Control point position**	**obs port**	**Active-low enabled**
latch_posedge_postcontrol_obs _invgclk	posedge	!(enl.clk)	after seq	yes	no
latch_negedge_invgclk	negedge	enl. !clk	none	no	no
latch_negedge_precontrol_ invgclk	negedge	enl. !clk	before seq	no	no
latch_negedge_postcontrol_ invgclk	negedge	enl. !clk	after seq	no	no
latch_negedge_precontrol_obs_ invgclk	negedge	enl. !clk	before seq	yes	no
latch_negedge_postcontrol_obs _invgclk	negedge	enl. !clk	after seq	yes	no
latch_posedgeactivelow_invgclk	posedge	!enl + !clk	none	no	yes
latch_posedgeactivelow_ precontrol_invgclk	posedge	!enl + !clk	before seq	no	yes
latch_posedgeactivelow_ postcontrol_invgclk	posedge	!enl + !clk	after seq	no	yes
latch_posedgeactivelow_ precontrol_obs_invgclk	posedge	!enl + !clk	before seq	yes	yes
latch_posedgeactivelow_ postcontrol_obs_invgclk	posedge	!enl + !clk	after seq	yes	yes
latch_negedgeactivelow_ invgclk	negedge	enl . !clk	none	no	yes
latch_negedgeactivelow_ precontrol_invgclk	negedge	enl . !clk	before seq	no	yes
latch_negedgeactivelow_ postcontrol_invgclk	negedge	enl . !clk	after seq	no	yes

Table A-2 Additional Latch-Based Structures Supported by the clock_gating_integrated_cell Attribute (Continued)

Derived cell	Type of gated register	Gating logic used	Control point position	obs port	Active-low enabled
latch_negedgeactivelow_ precontrol_obs_invgclk	negedge	enl . !clk	before seq	yes	yes
latch_negedgeactivelow_ postcontrol_obs_invgclk	negedge	enl . !clk	after seq	yes	yes

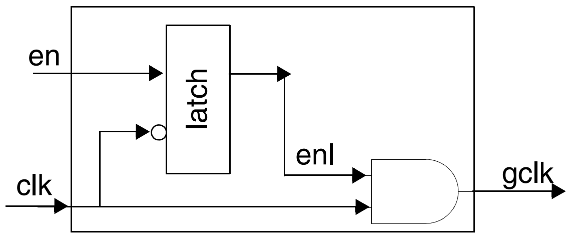
Schematics and Examples

This section shows circuit schematics and examples for the options listed in Table A-1. See “Generic Integrated Clock-Gating Schematics” on page A-39 to see circuit schematics for the derived integrated clock-gating cell structures listed in Table A-2.

latch_posedge Option

Figure A-1 shows the circuit of the `latch_posedge` option for the `clock_gating_integrated_cell` attribute, and Example A-1 demonstrates the use of this option.

Figure A-1 latch_posedge Circuit



Example A-1 latch_posedge Option

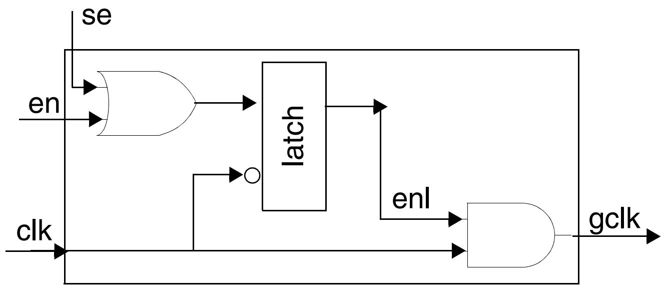
```
cell(CGLP) {
  area : 1;
  clock_gating_integrated_cell : "latch_posedge";
  dont_use : true;
  statetable(" CLK EN ", "ENL ") {
     table : " L  L  : - : L ,\
               L  H  : - : H ,\
               H  -  : - : N ";
  }
```

```
pin(ENL) {
  direction : internal;
  internal_node : "ENL";
}
pin(EN) {
  direction : input;
  capacitance : 0.017997;
  clock_gate_enable_pin : true;
  timing() {
    timing_type : setup_rising;
    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(CLK) {
  direction : input;
  capacitance : 0.031419;
  clock_gate_clock_pin : true;
  min_pulse_width_low  : 0.319;
}
pin(GCLK) {
  direction : output;
  state_function : "CLK * ENL";
  max_capacitance : 0.500;
  clock_gate_out_pin : true;
  timing() {
    timing_sense : positive_unate;
    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 : "CLK";
  }
  internal_power (){
    rise_power(li4X3){
      index_1("0.0150, 0.0400, 0.1050, 0.3550");
      index_2("0.050, 0.451, 1.501");
      values("0.141, 0.148, 0.256",\
      "0.162, 0.145, 0.234",\
      "0.192, 0.200, 0.284",\
      "0.199, 0.219, 0.297");
    }
    fall_power(li4X3){
      index_1("0.0150, 0.0400, 0.1050, 0.3550");
      index_2("0.050, 0.451, 1.500");
      values("0.117, 0.144, 0.246",\
      "0.133, 0.151, 0.238",\
      "0.151, 0.186, 0.279",\
      "0.160, 0.190, 0.217");
      }
      related_pin : "CLK EN" ;
    }
  }
}
```

latch_posedge_precontrol Option

Figure A-2 shows the circuit of the `latch_posedge_precontrol` option for the `clock_gating_integrated_cell` attribute, and Example A-2 demonstrates the use of this option.

Figure A-2 latch_posedge_precontrol Circuit



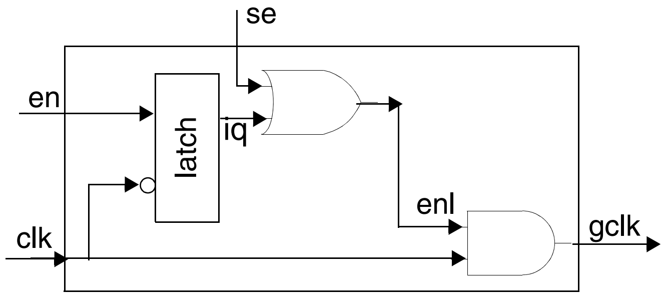
Example A-2 latch_posedge_precontrol Option

```
cell(CGLPC) {
  area : 1;
  clock_gating_integrated_cell : "latch_posedge_precontrol";
  dont_use : true;
  statetable(" CLK EN SE", "ENL ") {
     table : " L  L  L : - : L ,\
               L  L  H : - : H ,\
               L  H  L : - : H ,\
               L  H  H : - : H ,\
               H  -  - : - : N ";
  }
  pin(ENL) {
    direction : internal;
    internal_node : "ENL";
  }
  pin(EN) {
    direction : input;
    capacitance : 0.017997;
    clock_gate_enable_pin : true;
    timing() {
      timing_type : setup_rising;
      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 * ENL";
    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 : "CLK";
    }
    internal_power (){
      rise_power(li4X3){
        index_1("0.0150, 0.0400, 0.1050, 0.3550");
        index_2("0.050, 0.451, 1.501");
        values("0.141, 0.148, 0.256",\
        "0.162, 0.145, 0.234",\
        "0.192, 0.200, 0.284",\
        "0.199, 0.219, 0.297");
      }
      fall_power(li4X3){
        index_1("0.0150, 0.0400, 0.1050, 0.3550");
        index_2("0.050, 0.451, 1.500");
        values("0.141, 0.148, 0.256",\
        "0.162, 0.145, 0.234",\
        "0.192, 0.200, 0.284",\
        "0.199, 0.219, 0.297");
      }
      related_pin : "CLK EN" ;
    }
  }
}
```

latch_posedge_postcontrol Option

Figure A-3 shows the circuit of the `latch_posedge_postcontrol` option for the `clock_gating_integrated_cell` attribute, and Example A-3 demonstrates the use of this option.

Figure A-3 latch_posedge_postcontrol Circuit

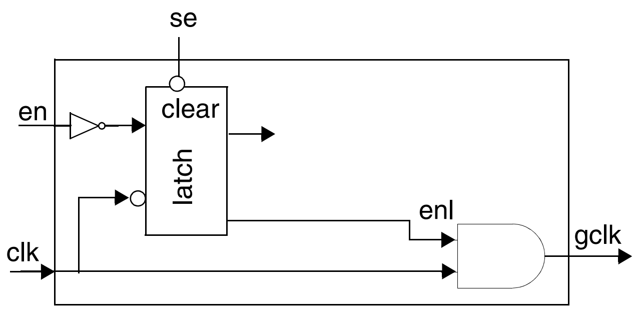


Example A-3 latch_posedge_postcontrol Option

```
cell(CGLPC2) {
  area : 1;
  clock_gating_integrated_cell : "latch_posedge_postcontrol";
  dont_use : true;
  statetable(" CLK EN ", "IQ ") {
     table : " L   L   : - : L ,\
              L   H   : - : H ,\
              H   -   : - : N ";
  }
  pin(IQ) {
    direction : internal;
    internal_node : "IQ";
  }
  pin(EN) {
    direction : input;
    capacitance : 0.017997;
    clock_gate_enable_pin : true;
    timing() {
      timing_type : setup_rising;
      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 + SE)";
    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 : "CLK";
    }
    internal_power (){
      rise_power(li4X3){
        index_1("0.0150, 0.0400, 0.1050, 0.3550");
        index_2("0.050, 0.451, 1.501");
        values("0.141, 0.148, 0.256",\
        "0.162, 0.145, 0.234",\
        "0.192, 0.200, 0.284",\
        "0.199, 0.219, 0.297");
      }
      fall_power(li4X3){
        index_1("0.0150, 0.0400, 0.1050, 0.3550");
        index_2("0.050, 0.451, 1.500");
        values("0.141, 0.148, 0.256",\
        "0.162, 0.145, 0.234",\
        "0.192, 0.200, 0.284",\
        "0.199, 0.219, 0.297");
      }
      related_pin : "CLK EN" ;
    }
  }
}
```

Figure A-4 shows the circuit of an inferred `latch_posedge_postcontrol` clock-gating integrated cell. Library Compiler infers the `clock_gating_integrated_cell` attribute as `latch_posedge_postcontrol` and saves it to the .db file.

Figure A-4 Inferred latch_posedge_postcontrol Clock-Gating Integrated Cell



Library Compiler automatically infers the `pin_is_active_high` attribute on the `clock_gate_test_pin` pin to indicate whether the pin is active high or active low. If the pin is active high, the attribute is set to `true`, or it is not specified at all; if the pin is active low, the attribute is set to `false`.

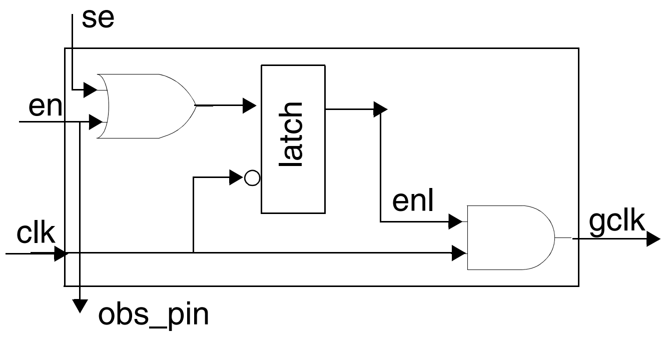
In addition, Library Compiler infers the `clock_gate_reset_pin` attribute on the latch's clear pin to indicate whether the pin is a reset pin. If the pin is a reset pin for the latch, the attribute is automatically set to `true`; if the pin is not a reset pin for the latch, the attribute is not specified at all.

Important:
If you set the `pin_is_active_high` attribute to indicate whether a pin is active high or active low, or if you set the `clock_gate_reset_pin` attribute to indicate whether a pin is a reset pin for a `latch_posedge_postcontrol` generic integrated clock gating cell, Library Compiler issues an error message. Library Compiler infers the values based on the pins. You cannot specify the values.

latch_posedge_precontrol_obs Option

Figure A-5 shows the circuit of the `latch_posedge_precontrol_obs` option for the `clock_gating_integrated_cell` attribute, and Example A-4 demonstrates the use of this option.

Figure A-5 latch_posedge_precontrol_obs Circuit



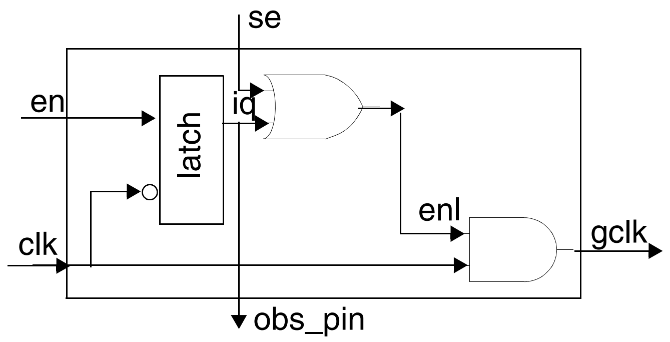
Example A-4 latch_posedge_precontrol_obs Option

```
cell(CGLPCO) {
  area : 1;
  clock_gating_integrated_cell : "latch_posedge_precontrol_obs";
  dont_use : true;
  statetable(" CLK EN SE", "ENL") {
     table : " L  L  L : - : L ,\
               L  L  H : - : H ,\
               L  H  L : - : H ,\
               L  H  H : - : H ,\
               H  -  - : - : N ";
  }
  pin(ENL) {
    direction : internal;
    internal_node : "ENL";
  }
  pin(EN) {
    direction : input;
    capacitance : 0.017997;
    clock_gate_enable_pin : true;
    timing() {
      timing_type : setup_rising;
      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 * ENL";
    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 : "EN" ;
    }
  }
  pin(OBS_PIN) {
    direction : output;
    state_function : "EN";
    max_capacitance : 0.500;
    clock_gate_obs_pin : true;
  }
}
```

latch_posedge_postcontrol_obs Option

Figure A-6 shows the circuit of the `latch_posedge_postcontrol_obs` option for the `clock_gating_integrated_cell` attribute, and Example A-5 demonstrates the use of this option.

Figure A-6 latch_posedge_postcontrol_obs Circuit



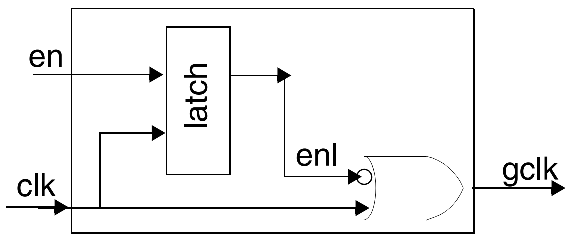
Example A-5 latch_posedge_postcontrol_obs Option

```
cell(CGLPC2O) {
  area : 1;
  clock_gating_integrated_cell : "latch_posedge_postcontrol_obs";
  dont_use : true;
  statetable(" CLK EN ", "IQ") {
     table : " L  L  : - : L ,\
               L  H  : - : H ,\
               H  -  : - : N ";
  }
  pin(EN) {
    direction : input;
    capacitance : 0.017997;
    clock_gate_enable_pin : true;
    timing() {
      timing_type : setup_rising;
      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(IQ) {
    direction : internal;
    internal_node : "IQ";
  }
  pin(GCLK) {
    direction : output;
    state_function : "CLK * (IQ + SE)";
    max_capacitance : 0.500;
    clock_gate_out_pin : true;
    timing() {
      timing_sense : positive_unate;
      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 : "CLK";
    }
    internal_power (){
      rise_power(li4X3){
        index_1("0.0150, 0.0400, 0.1050, 0.3550");
        index_2("0.050, 0.451, 1.501");
        values("0.141, 0.148, 0.256",\
        "0.162, 0.145, 0.234",\
        "0.192, 0.200, 0.284",\
        "0.199, 0.219, 0.297");
      }
      fall_power(li4X3){
        index_1("0.0150, 0.0400, 0.1050, 0.3550");
        index_2("0.050, 0.451, 1.500");
        values("0.141, 0.148, 0.256",\
        "0.162, 0.145, 0.234",\
        "0.192, 0.200, 0.284",\
        "0.199, 0.219, 0.297");
      }
      related_pin : "CLK EN" ;
    }
  pin(OBS_PIN) {
    direction : output;
    internal_node : "IQ";
    max_capacitance : 0.500;
    clock_gate_obs_pin : true;
  }
}
```

latch_negedge Option

Figure A-7 shows the circuit of the `latch_negedge` option for the `clock_gating_integrated_cell` attribute, and Example A-6 demonstrates the use of this option.

Figure A-7 latch_negedge Circuit



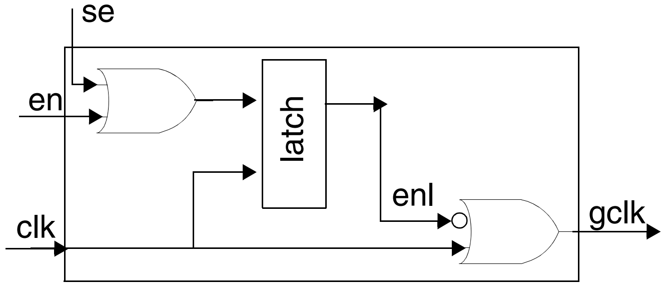
Example A-6 latch_negedge Option

```
cell(CGLN) {
  area : 1;
  clock_gating_integrated_cell : "latch_negedge";
  dont_use : true;
  dont_touch : true;
  statetable(" CLK EN ", "ENL ") {
     table : " H  L  : - : L ,\
               H  H  : - : H ,\
               L  -  : - : N ";
  }
  pin(ENL) {
    direction : internal;
    internal_node : "ENL";
  }
  pin(EN) {
    direction : input;
    capacitance : 0.017997;
    clock_gate_enable_pin : true;
    timing() {
      timing_type : setup_falling;
      intrinsic_rise : 0.4;
      intrinsic_fall : 0.4;
      related_pin : "CLK";
    }
    timing() {
      timing_type : hold_falling;
      intrinsic_rise : 0.4;
      intrinsic_fall : 0.4;
      related_pin : "CLK";
    }
  }
  pin(CLK) {
    direction : input;
    capacitance : 0.031419;
    clock_gate_clock_pin : true;
    min_pulse_width_low  : 0.319;
  }
  pin(GCLK) {
    direction : output;
    state_function : "CLK + ENL'";
    max_capacitance : 0.500;
    clock_gate_out_pin : true;
    timing() {
      timing_sense : positive_unate;
      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("0.117, 0.144, 0.246",\
        "0.133, 0.151, 0.238",\
        "0.151, 0.186, 0.279",\
        "0.160, 0.190, 0.217");
      }
      related_pin : "EN" ;
    }
  }
}
```

latch_negedge_precontrol Option

Figure A-8 shows the circuit of the `latch_negedge_precontrol` option for the `clock_gating_integrated_cell` attribute, and Example A-7 demonstrates the use of this option.

Figure A-8 latch_negedge_precontrol Circuit



Example A-7 latch_negedge_precontrol Option

```
cell(CGLNC) {
```

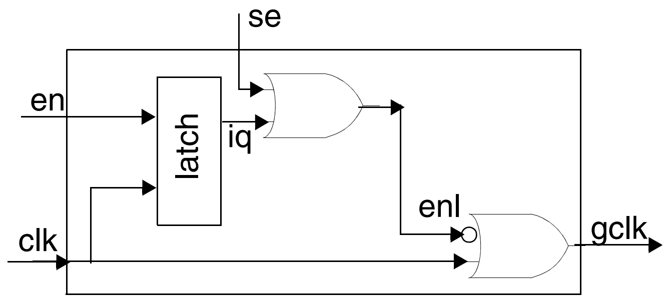
```
area : 1;
clock_gating_integrated_cell : "latch_negedge_precontrol";
dont_use : true;
dont_touch : true;
statetable(" CLK EN SE", "ENL ") {
   table : " H  L  L : - : L ,\
             H  L  H : - : H ,\
             H  H  L : - : H ,\
             H  H  H : - : H ,\
             L  -  - : - : N ";
}
pin(ENL) {
  direction : internal;
  internal_node : "ENL";
}
pin(EN) {
  direction : input;
  capacitance : 0.017997;
  clock_gate_enable_pin : true;
  timing() {
    timing_type : setup_falling;
    intrinsic_rise : 0.4;
    intrinsic_fall : 0.4;
    related_pin : "CLK";
  }
  timing() {
    timing_type : hold_falling;
    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 + ENL'";
  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 : "CLK";
  }
```

```
    internal_power (){
      rise_power(li4X3){
        index_1("0.0150, 0.0400, 0.1050, 0.3550");
        index_2("0.050, 0.451, 1.501");
        values("0.141, 0.148, 0.256",\
        "0.162, 0.145, 0.234",\
        "0.192, 0.200, 0.284",\
        "0.199, 0.219, 0.297");
      }
      fall_power(li4X3){
        index_1("0.0150, 0.0400, 0.1050, 0.3550");
        index_2("0.050, 0.451, 1.500");
        values("0.117, 0.144, 0.246",\
        "0.133, 0.151, 0.238",\
        "0.151, 0.186, 0.279",\
        "0.160, 0.190, 0.217");
      }
      related_pin : "CLK EN" ;
    }
  }
}
```

latch_negedge_postcontrol Option

Figure A-9 shows the circuit of the `latch_negedge_postcontrol` option for the `clock_gating_integrated_cell` attribute, and Example A-8 demonstrates the use of this option.

Figure A-9 latch_negedge_postcontrol Circuit



Example A-8 latch_negedge_postcontrol Option

```
cell(CGLNC2) {
  area : 1;
  clock_gating_integrated_cell : "latch_negedge_postcontrol";
  dont_use : true;
  dont_touch : true;
  statetable(" CLK EN ", "IQ ") {
     table : " H  L  : - : L ,\
               H  H  : - : H ,\
               L  -  : - : N ";
  }
```

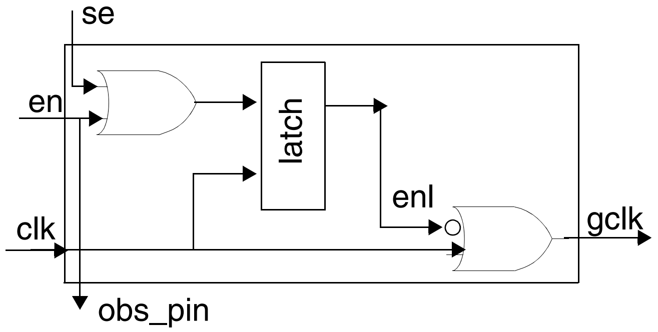
```
pin(IQ) {
  direction : internal;
  internal_node : "IQ";
}
pin(EN) {
  direction : input;
  capacitance : 0.017997;
  clock_gate_enable_pin : true;
  timing() {
    timing_type : setup_falling;
    intrinsic_rise : 0.4;
    intrinsic_fall : 0.4;
    related_pin : "CLK";
  }
  timing() {
    timing_type : hold_falling;
    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 + SE)";
  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 : "CLK";
  }
  internal_power (){
    rise_power(li4X3){
      index_1("0.0150, 0.0400, 0.1050, 0.3550");
      index_2("0.050, 0.451, 1.501");
      values("0.141, 0.148, 0.256",\
      "0.162, 0.145, 0.234",\
      "0.192, 0.200, 0.284",\
      "0.199, 0.219, 0.297");
    }
    fall_power(li4X3){
      index_1("0.0150, 0.0400, 0.1050, 0.3550");
```

```
          index_2("0.050, 0.451, 1.500");
          values("0.117, 0.144, 0.246",\
          "0.133, 0.151, 0.238",\
          "0.151, 0.186, 0.279",\
          "0.160, 0.190, 0.217");
        }
        related_pin : "CLK EN" ;
      }
    }
}
```

latch_negedge_precontrol_obs Option

Figure A-10 shows the circuit of the `latch_negedge_precontrol_obs` option for the `clock_gating_integrated_cell` attribute, and Example A-9 demonstrates the use of this option.

Figure A-10 latch_negedge_precontrol_obs Circuit



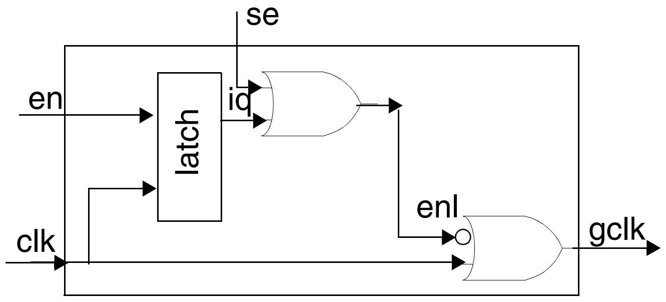
Example A-9 latch_negedge_precontrol_obs Option

```
cell(CGLNCO) {
  area : 1;
  clock_gating_integrated_cell : "latch_negedge_precontrol_obs";
  dont_use : true;
  dont_touch : true;
  statetable(" CLK EN SE", "ENL ") {
     table : " H  L  L : - : L ,\
              H  L  H : - : H ,\
              H  H  L : - : H ,\
              H  H  H : - : H ,\
              L  -  - : - : N ";
  }
  pin(ENL) {
    direction : internal;
    internal_node : "ENL";
  }
  pin(EN) {
    direction : input;
    capacitance : 0.017997;
    clock_gate_enable_pin : true;
    timing() {
      timing_type : setup_falling;
      intrinsic_rise : 0.4;
      intrinsic_fall : 0.4;
      related_pin : "CLK";
    }
    timing() {
      timing_type : hold_falling;
      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 + ENL'";
    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 : "CLK";
    }
    internal_power (){
      rise_power(li4X3){
        index_1("0.0150, 0.0400, 0.1050, 0.3550");
        index_2("0.050, 0.451, 1.501");
        values("0.141, 0.148, 0.256",\
        "0.162, 0.145, 0.234",\
        "0.192, 0.200, 0.284",\
        "0.199, 0.219, 0.297");
      }
      fall_power(li4X3){
        index_1("0.0150, 0.0400, 0.1050, 0.3550");
        index_2("0.050, 0.451, 1.500");
        values("0.117, 0.144, 0.246",\
        "0.133, 0.151, 0.238",\
        "0.151, 0.186, 0.279",\
        "0.160, 0.190, 0.217");
      }
      related_pin : "CLK EN" ;
    }
  }
  pin(OBS_PIN) {
    direction : output;
    state_function : "EN";
    max_capacitance : 0.500;
    clock_gate_obs_pin : true;
  }
}
```

latch_negedge_postcontrol_obs Option

Figure A-11 shows the circuit of the `latch_negedge_postcontrol_obs` option for the `clock_gating_integrated_cell` attribute, and Example A-10 demonstrates the use of this option.

Figure A-11 latch_negedge_postcontrol_obs Circuit



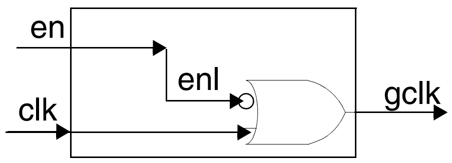
Example A-10 latch_negedge_postcontrol_obs Option

```
cell(CGLNC2O) {
  area : 1;
  clock_gating_integrated_cell : "latch_negedge_postcontrol_obs";
  dont_use : true;
  dont_touch : true;
  statetable(" CLK EN ", "IQ") {
     table : " H  L  : - : L ,\
              H  H  : - : H ,\
              L  -  : - : N ";
  }
  pin(EN) {
    direction : input;
    capacitance : 0.017997;
    clock_gate_enable_pin : true;
    timing() {
      timing_type : setup_falling;
      intrinsic_rise : 0.4;
      intrinsic_fall : 0.4;
      related_pin : "CLK";
    }
    timing() {
      timing_type : hold_falling;
      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(IQ) {
    direction : internal;
    internal_node : "IQ";
  }
  pin(GCLK) {
    direction : output;
    state_function : "CLK + !(IQ + SE)";
    max_capacitance : 0.500;
    clock_gate_out_pin : true;
    timing() {
      timing_sense : positive_unate;
      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 : "CLK";
    }
    internal_power (){
      rise_power(li4X3){
        index_1("0.0150, 0.0400, 0.1050, 0.3550");
        index_2("0.050, 0.451, 1.501");
        values("0.141, 0.148, 0.256",\
        "0.162, 0.145, 0.234",\
        "0.192, 0.200, 0.284",\
        "0.199, 0.219, 0.297");
      }
      fall_power(li4X3){
        index_1("0.0150, 0.0400, 0.1050, 0.3550");
        index_2("0.050, 0.451, 1.500");
        values("0.117, 0.144, 0.246",\
        "0.133, 0.151, 0.238",\
        "0.151, 0.186, 0.279",\
        "0.160, 0.190, 0.217");
      }
      related_pin : "CLK EN" ;
    }
  }
}
```

none_posedge Option

Figure A-12 shows the circuit of the `none_posedge` option for the `clock_gating_integrated_cell` attribute, and Example A-11 demonstrates the use of this option.

Figure A-12 none_posedge Circuit



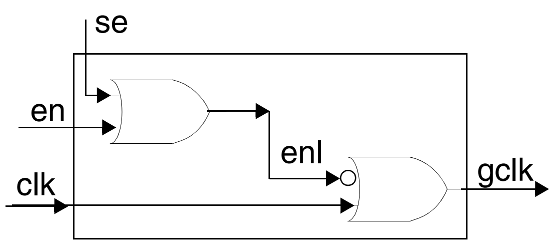
Example A-11 none_posedge Option

```
cell(CGNP) {
  area : 1;
  clock_gating_integrated_cell : "none_posedge";
  dont_use : true;
  pin(EN) {
    direction : input;
    capacitance : 0.017997;
    clock_gate_enable_pin : true;
    timing() {
      timing_type : nochange_high_low;
      intrinsic_rise : 0.4;
      intrinsic_fall : 0.4;
      related_pin : "CLK";
    }
    timing() {
      timing_type : nochange_low_low;
      intrinsic_rise : 0.4;
      intrinsic_fall : 0.4;
      related_pin : "CLK";
    }
  }
  pin(CLK) {
    direction : input;
    capacitance : 0.031419;
    clock_gate_clock_pin : true;
    min_pulse_width_low  : 0.319;
  }
  pin(GCLK) {
    direction : output;
    function : "CLK + EN'";
    max_capacitance : 0.500;
    clock_gate_out_pin : true;
    timing() {
      timing_sense : positive_unate;
      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";
    }
    timing() {
      timing_sense : positive_unate;
      intrinsic_rise : 0.22;
      intrinsic_fall : 0.22;
      rise_resistance : 0.1443;
      fall_resistance : 0.0523;
      slope_rise : 0.0;
      slope_fall : 0.0;
      related_pin : "CLK";
    }
    internal_power (){
      rise_power(li4X3){
        index_1("0.0150, 0.0400, 0.1050, 0.3550");
        index_2("0.050, 0.451, 1.501");
        values("0.141, 0.148, 0.256",\
        "0.162, 0.145, 0.234",\
        "0.192, 0.200, 0.284",\
        "0.199, 0.219, 0.297");
      }
      fall_power(li4X3){
        index_1("0.0150, 0.0400, 0.1050, 0.3550");
        index_2("0.050, 0.451, 1.500");
        values("0.117, 0.144, 0.246",\
        "0.133, 0.151, 0.238",\
        "0.151, 0.186, 0.279",\
        "0.160, 0.190, 0.217");
      }
      related_pin : "CLK EN" ;
    }
  }
}
```

none_posedge_control Option

Figure A-13 shows the circuit of the `none_posedge_control` option for the `clock_gating_integrated_cell` attribute, and Example A-12 demonstrates the use of this option.

Figure A-13 none_posedge_control Circuit



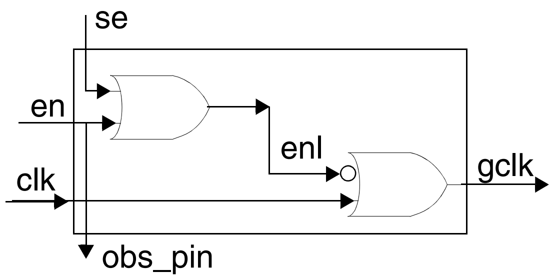
Example A-12 none_posedge_control Option

```
cell(CGNPC) {
  area : 1;
  clock_gating_integrated_cell : "none_posedge_control";
  dont_use : true;
  pin(EN) {
    direction : input;
    capacitance : 0.017997;
    clock_gate_enable_pin : true;
    timing() {
      timing_type : nochange_high_low;
      intrinsic_rise : 0.4;
      intrinsic_fall : 0.4;
      related_pin : "CLK";
    }
    timing() {
      timing_type : nochange_low_low;
      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;
    function : "CLK + !(SE + EN)";
    max_capacitance : 0.500;
    clock_gate_out_pin : true;
    internal_power (){
      rise_power(li4X3){
        index_1("0.0150, 0.0400, 0.1050, 0.3550");
        index_2("0.050, 0.451, 1.501");
        values("0.141, 0.148, 0.256",\
        "0.162, 0.145, 0.234",\
        "0.192, 0.200, 0.284",\
        "0.199, 0.219, 0.297");
      }
      fall_power(li4X3){
        index_1("0.0150, 0.0400, 0.1050, 0.3550");
        index_2("0.050, 0.451, 1.500");
        values("0.141, 0.148, 0.256",\
        "0.162, 0.145, 0.234",\
        "0.192, 0.200, 0.284",\
        "0.199, 0.219, 0.297");
      }
      related_pin : "CLK EN" ;
    }
    timing() {
      timing_sense : positive_unate;
      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";
    }
    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 : "SE";
    }
    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 : "CLK";
    }
  }
}
```

none_posedge_control_obs Option

Figure A-14 shows the circuit of the `none_posedge_control_obs` option for the `clock_gating_integrated_cell` attribute, and Example A-13 demonstrates the use of this option.

Figure A-14 none_posedge_control_obs Circuit



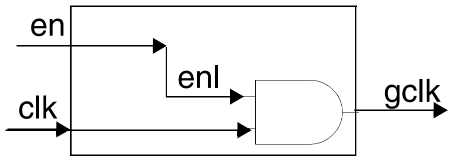
Example A-13 none_posedge_control_obs Option

```
cell(CGNPCO) {
  area : 1;
  clock_gating_integrated_cell : "none_posedge_control_obs";
  dont_use : true;
  pin(EN) {
    direction : input;
    capacitance : 0.017997;
    clock_gate_enable_pin : true;
    timing() {
      timing_type : nochange_high_low;
      intrinsic_rise : 0.4;
      intrinsic_fall : 0.4;
      related_pin : "CLK";
    }
    timing() {
      timing_type : nochange_low_low;
      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(OBS_PIN) {
    direction : output;
    function : "EN";
    clock_gate_obs_pin : true;
    max_capacitance : 0.500;
    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";
    }
  }
  pin(GCLK) {
    direction : output;
    function : "CLK + !(SE + EN)";
    max_capacitance : 0.500;
    clock_gate_out_pin : true;
    internal_power (){
      rise_power(li4X3){
        index_1("0.0150, 0.0400, 0.1050, 0.3550");
        index_2("0.050, 0.451, 1.501");
        values("0.141, 0.148, 0.256",\
        "0.162, 0.145, 0.234",\
        "0.192, 0.200, 0.284",\
        "0.199, 0.219, 0.297");
      }
      fall_power(li4X3){
        index_1("0.0150, 0.0400, 0.1050, 0.3550");
        index_2("0.050, 0.451, 1.500");
        values("0.141, 0.148, 0.256",\
        "0.162, 0.145, 0.234",\
        "0.192, 0.200, 0.284",\
        "0.199, 0.219, 0.297");
      }
      related_pin : "CLK EN" ;
    }
    timing() {
      timing_sense : positive_unate;
      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";
    }
    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 : "SE";
    }
    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 : "CLK";
    }
  }
}
```

none_negedge Option

Figure A-15 shows the circuit of the `none_negedge` option for the `clock_gating_integrated_cell` attribute, and Example A-14 demonstrates the use of this option.

Figure A-15 none_negedge Circuit



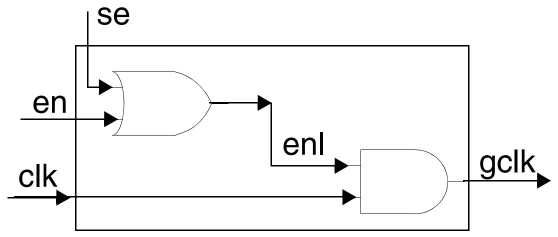
Example A-14 none_negedge Option

```
cell(CGNN) {
  area : 1;
  clock_gating_integrated_cell : "none_negedge";
  dont_use : true;
  dont_touch : true;
  pin(EN) {
    direction : input;
    capacitance : 0.017997;
    clock_gate_enable_pin : true;
    timing() {
      timing_type : nochange_high_high;
      intrinsic_rise : 0.4;
      intrinsic_fall : 0.4;
      related_pin : "CLK";
    }
    timing() {
      timing_type : nochange_low_high;
      intrinsic_rise : 0.4;
      intrinsic_fall : 0.4;
      related_pin : "CLK";
    }
  }
  pin(CLK) {
    direction : input;
    capacitance : 0.031419;
    clock_gate_clock_pin : true;
    min_pulse_width_low  : 0.319;
  }
  pin(GCLK) {
    direction : output;
    function : "CLK * EN";
    max_capacitance : 0.500;
    clock_gate_out_pin : true;
    internal_power (){
      rise_power(li4X3){
        index_1("0.0150, 0.0400, 0.1050, 0.3550");
        index_2("0.050, 0.451, 1.501");
        values("0.141, 0.148, 0.256",\
        "0.162, 0.145, 0.234",\
        "0.192, 0.200, 0.284",\
        "0.199, 0.219, 0.297");
      }
      fall_power(li4X3){
        index_1("0.0150, 0.0400, 0.1050, 0.3550");
        index_2("0.050, 0.451, 1.500");
        values("0.117, 0.144, 0.246",\
        "0.133, 0.151, 0.238",\
        "0.151, 0.186, 0.279",\
        "0.160, 0.190, 0.217");
      }
      related_pin : "CLK EN" ;
    }
    timing() {
      timing_sense : positive_unate;
      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";
    }
    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 : "CLK";
    }
  }
}
```

none_negedge_control Option

Figure A-16 shows the circuit of the `none_negedge_control` option for the `clock_gating_integrated_cell` attribute, and Example A-15 demonstrates the use of this option.

Figure A-16 none_negedge_control Circuit



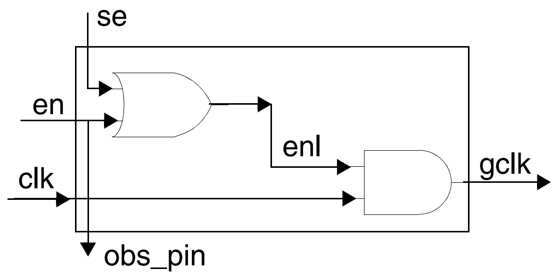
Example A-15 none_negedge_control Option

```
cell(CGNNC) {
  area : 1;
  clock_gating_integrated_cell : "none_negedge_control";
  dont_use : true;
  dont_touch : true;
  pin(EN) {
    direction : input;
    capacitance : 0.017997;
    clock_gate_enable_pin : true;
    timing() {
      timing_type : nochange_high_high;
      intrinsic_rise : 0.4;
      intrinsic_fall : 0.4;
      related_pin : "CLK";
    }
    timing() {
      timing_type : nochange_low_high;
      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;
    function : "CLK * (SE + EN)";
    max_capacitance : 0.500;
    clock_gate_out_pin : true;
    internal_power (){
      rise_power(li4X3){
        index_1("0.0150, 0.0400, 0.1050, 0.3550");
        index_2("0.050, 0.451, 1.501");
        values("0.141, 0.148, 0.256",\
        "0.162, 0.145, 0.234",\
        "0.192, 0.200, 0.284",\
        "0.199, 0.219, 0.297");
      }
      fall_power(li4X3){
        index_1("0.0150, 0.0400, 0.1050, 0.3550");
        index_2("0.050, 0.451, 1.500");
        values("0.117, 0.144, 0.246",\
        "0.133, 0.151, 0.238",\
        "0.151, 0.186, 0.279",\
        "0.160, 0.190, 0.217");
      }
      related_pin : "CLK EN" ;
    }
    timing() {
      timing_sense : positive_unate;
      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";
    }
    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 : "SE";
    }
    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 : "CLK";
    }
  }
}
```

none_negedge_control_obs Option

Figure A-17 shows the circuit of the `none_negedge_control_obs` option for the `clock_gating_integrated_cell` attribute, and Example A-16 demonstrates the use of this option.

Figure A-17 none_negedge_control_obs Circuit



Example A-16 none_negedge_control_obs Option

```
cell(CGNNCO) {
  area : 1;
  clock_gating_integrated_cell : "none_negedge_control_obs";
  dont_use : true;
  dont_touch : true;
  pin(EN) {
    direction : input;
    capacitance : 0.017997;
    clock_gate_enable_pin : true;
    timing() {
      timing_type : nochange_high_high;
      intrinsic_rise : 0.4;
      intrinsic_fall : 0.4;
      related_pin : "CLK";
    }
    timing() {
      timing_type : nochange_low_high;
      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(OBS_PIN) {
    direction : output;
    function : "EN";
    clock_gate_obs_pin : true;
    max_capacitance : 0.500;
    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";
    }
  }
  pin(GCLK) {
    direction : output;
    function : "CLK * (SE + EN)";
    max_capacitance : 0.500;
    clock_gate_out_pin : true;
    internal_power (){
      rise_power(li4X3){
        index_1("0.0150, 0.0400, 0.1050, 0.3550");
        index_2("0.050, 0.451, 1.501");
        values("0.141, 0.148, 0.256",\
        "0.162, 0.145, 0.234",\
        "0.192, 0.200, 0.284",\
        "0.199, 0.219, 0.297");
      }
      fall_power(li4X3){
        index_1("0.0150, 0.0400, 0.1050, 0.3550");
        index_2("0.050, 0.451, 1.500");
        values("0.117, 0.144, 0.246",\
        "0.133, 0.151, 0.238",\
        "0.151, 0.186, 0.279",\
        "0.160, 0.190, 0.217");
      }
      related_pin : "CLK EN" ;
    }
    timing() {
      timing_sense : positive_unate;
      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";
    }
    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 : "SE";
    }
    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 : "CLK";
      }
    }
}
```

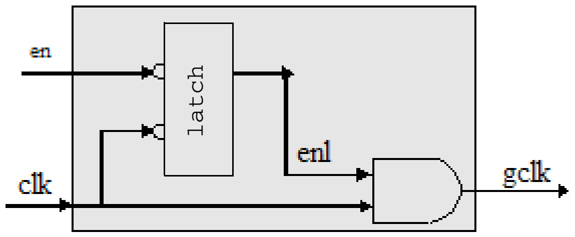
Generic Integrated Clock-Gating Schematics

This section provides circuit schematics for the cell structures listed in Table A-2 on page A-5. After you model a cell using the generic integrated clock-gating syntax, Library Compiler derives one of the following structures automatically.

latch_posedgeactivelow

Figure A-18 shows the derived `latch_posedgeactivelow` integrated clock-gating cell circuit.

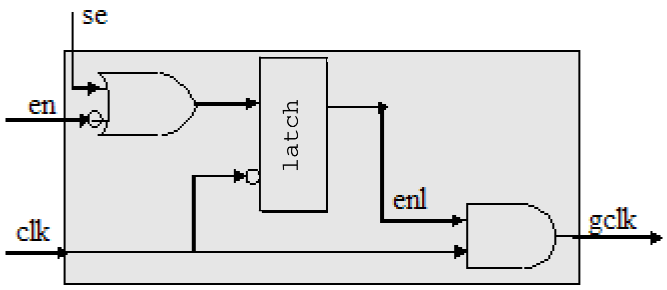
Figure A-18 latch_posedgeactivelow Circuit



latch_posedgeactivelow_precontrol

Figure A-19 shows the derived `latch_posedgeactivelow_precontrol` integrated clock-gating cell circuit.

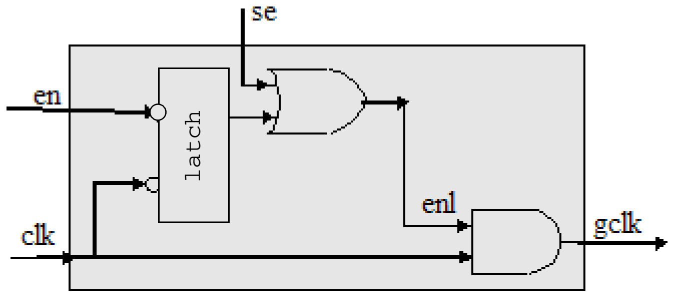
Figure A-19 latch_posedgeactivelow_precontrol Circuit



latch_posedgeactivelow_postcontrol

Figure A-20 shows the derived `latch_posedgeactivelow_postcontrol` integrated clock-gating cell circuit.

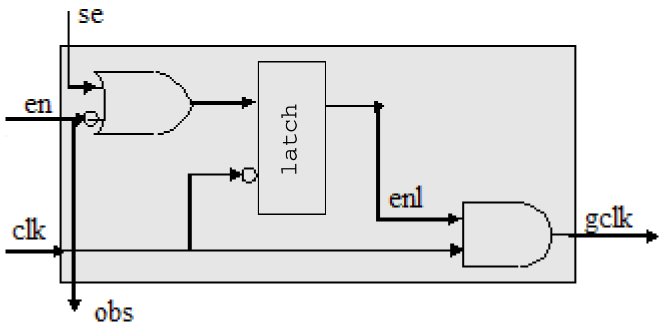
Figure A-20 latch_posedgeactivelow_postcontrol Circuit



latch_posedgeactivelow_precontrol_obs

Figure A-21 shows the derived `latch_posedgeactivelow_precontrol_obs` integrated clock-gating cell circuit.

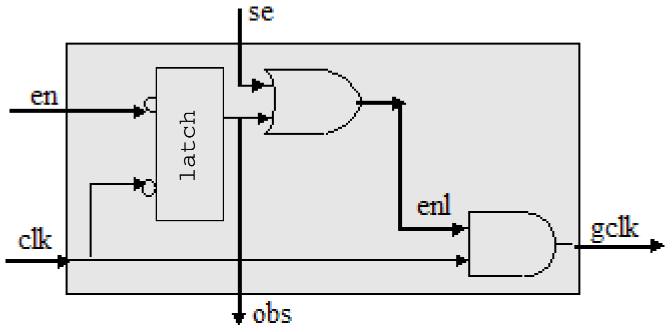
Figure A-21 latch_posedgeactivelow_precontrol_obs Circuit



latch_posedgeactivelow_postcontrol_obs

Figure A-22 shows the derived `latch_posedgeactivelow_postcontrol_obs` integrated clock-gating cell circuit.

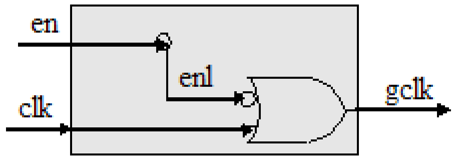
Figure A-22 latch_posedgeactivelow_postcontrol_obs Circuit



none_posedgeactivelow

Figure A-23 shows the derived `none_posedgeactivelow` integrated clock-gating cell circuit.

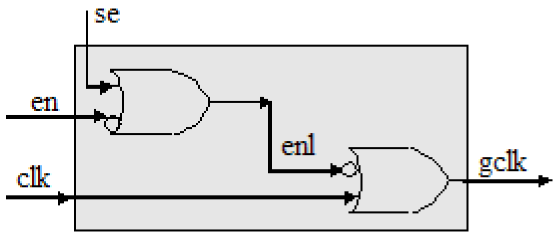
Figure A-23 none_posedgeactivelow Circuit



none_posedgeactivelow_control

Figure A-24 shows the derived `none_posedgeactivelow_control` integrated clock-gating cell circuit.

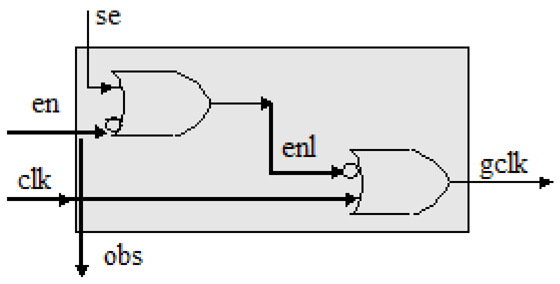
Figure A-24 none_posedgeactivelow_control Circuit



none_posedgeactivelow_control_obs

Figure A-25 shows the derived `none_posedgeactivelow_control_obs` integrated clock-gating cell circuit.

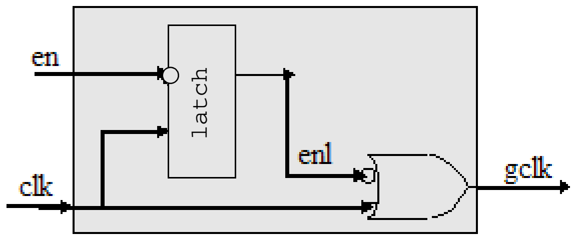
Figure A-25 none_posedgeactivelow_control_obs Circuit



latch_negedgeactivelow

Figure A-26 shows the derived `latch_negedgeactivelow` integrated clock-gating cell circuit.

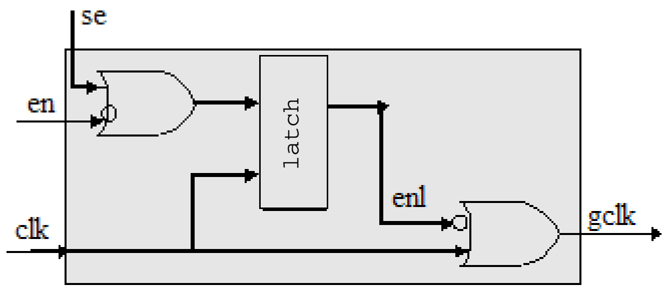
Figure A-26 latch_negedgeactivelow Circuit



latch_negedgeactivelow_precontrol

Figure A-27 shows the derived `latch_negedgeactivelow_precontrol` integrated clock-gating cell circuit.

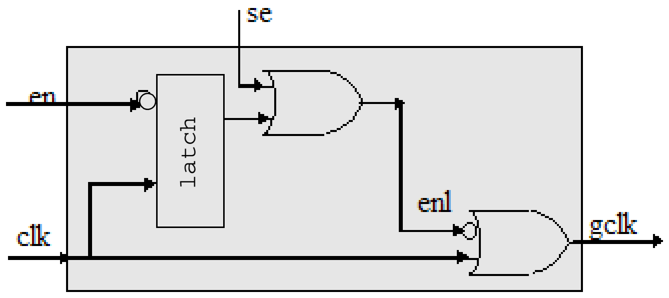
Figure A-27 latch_negedgeactivelow_precontrol Circuit



latch_negedgeactivelow_postcontrol

Figure A-28 shows the derived `latch_negedgeactivelow_postcontrol` integrated clock-gating cell circuit.

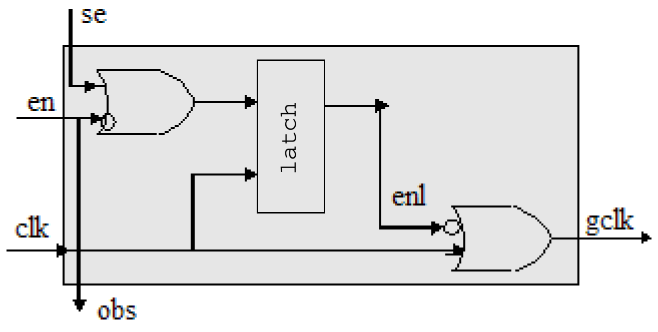
Figure A-28 latch_negedgeactivelow_postcontrol Circuit



latch_negedgeactivelow_precontrol_obs

Figure A-29 shows the derived `latch_negedgeactivelow_precontrol_obs` integrated clock-gating cell circuit.

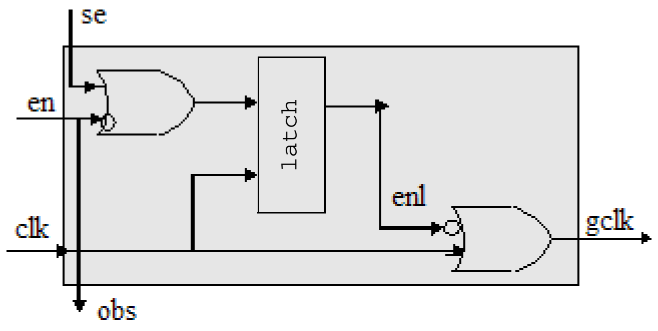
Figure A-29 latch_negedgeactivelow_precontrol_obs Circuit



latch_negedgeactivelow_postcontrol_obs

Figure A-30 shows the derived `latch_negedgeactivelow_postcontrol_obs` integrated clock-gating cell circuit.

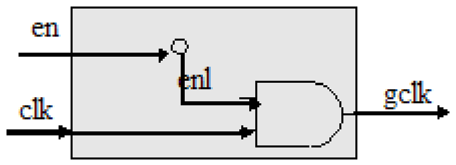
Figure A-30 latch_negedgeactivelow_postcontrol_obs Circuit



none_negedgeactivelow

Figure A-31 shows the derived `none_negedgeactivelow` integrated clock-gating cell circuit.

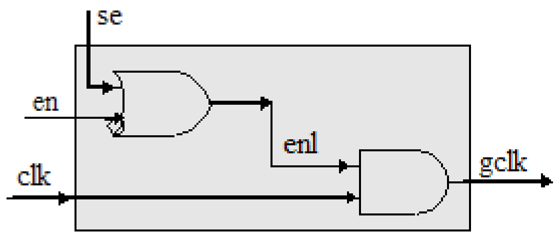
Figure A-31 none_negedgeactivelow Circuit



none_negedgeactivelow_control

Figure A-32 shows the derived `none_negedgeactivelow_control` integrated clock-gating cell circuit.

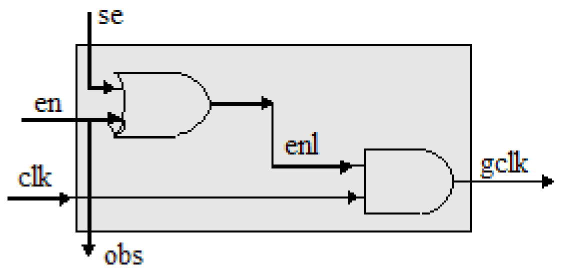
Figure A-32 none_negedgeactivelow_control Circuit



none_negedgeactivelow_control_obs

Figure A-33 shows the derived `none_negedgeactivelow_control_obs` integrated clock-gating cell circuit.

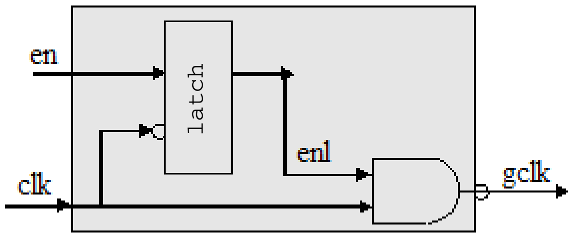
Figure A-33 none_negedgeactivelow_control_obs Circuit



latch_posedge_invgclk

Figure A-34 shows the derived `latch_posedge_invgclk` integrated clock-gating cell circuit.

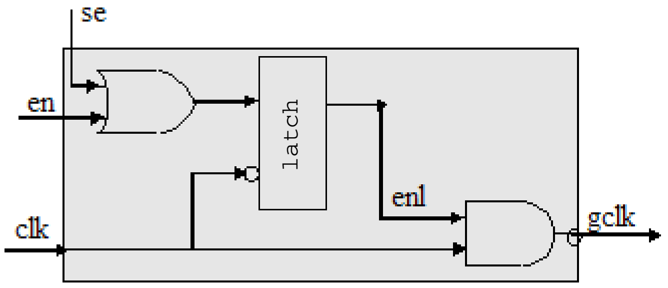
Figure A-34 latch_posedge_invgclk Circuit



latch_posedge_precontrol_invgclk

Figure A-35 shows the derived `latch_posedge_precontrol_invgclk` integrated clock-gating cell circuit.

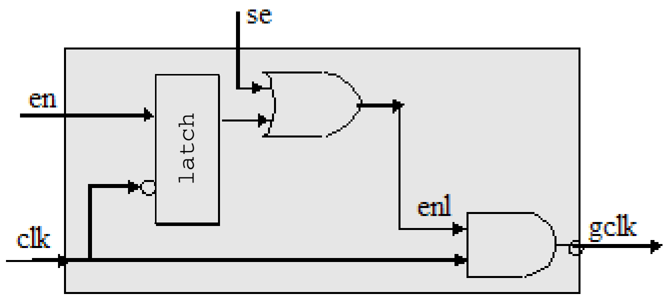
Figure A-35 latch_posedge_precontrol_invgclk Circuit



latch_posedge_postcontrol_invgclk

Figure A-36 shows the derived `latch_posedge_postcontrol_invgclk` integrated clock-gating cell circuit.

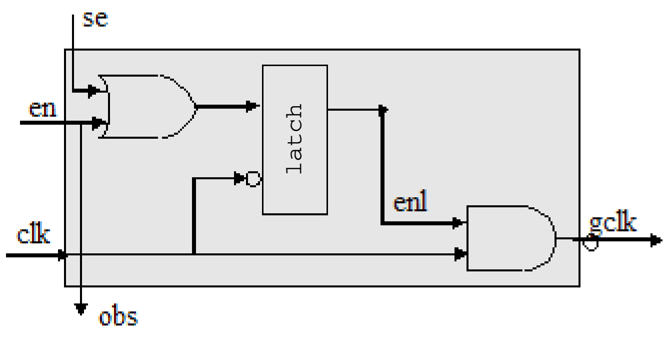
Figure A-36 latch_posedge_postcontrol_invgclk Circuit



latch_posedge_precontrol_obs_invgclk

Figure A-37 shows the derived `latch_posedge_precontrol_obs_invgclk` integrated clock-gating cell circuit.

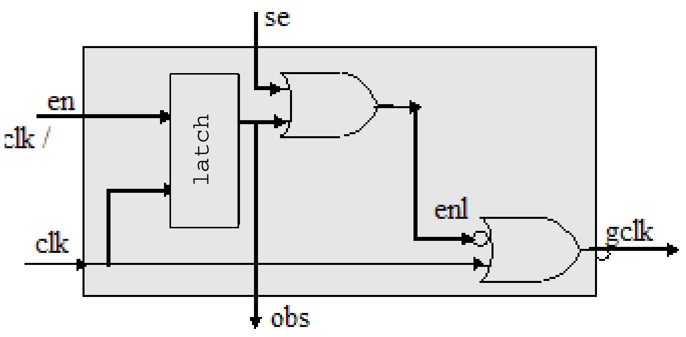
Figure A-37 latch_posedge_precontrol_obs_invgclk Circuit



latch_posedge_postcontrol_obs_invgclk

Figure A-38 shows the derived `latch_posedge_postcontrol_obs_invgclk` integrated clock-gating cell circuit.

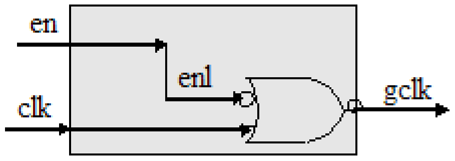
Figure A-38 latch_posedge_postcontrol_obs_invgclk Circuit



none_posedge_invgclk

Figure A-39 shows the derived `none_posedge_invgclk` integrated clock-gating cell circuit.

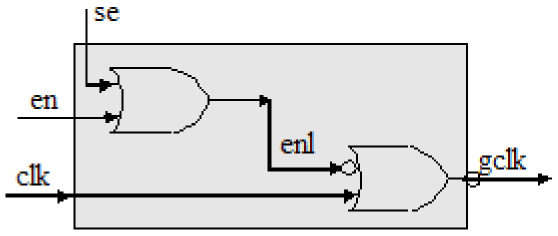
Figure A-39 none_posedge_invgclk Circuit



none_posedge_control_invgclk

Figure A-40 shows the derived `none_posedge_control_invgclk` integrated clock-gating cell circuit.

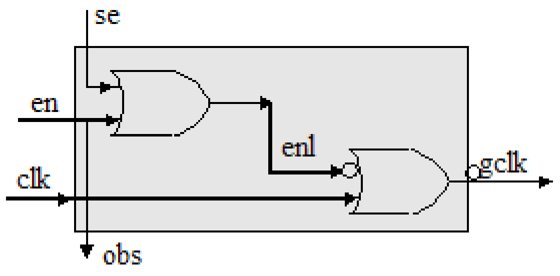
Figure A-40 none_posedge_control_invgclk Circuit



none_posedge_control_obs_invgclk

Figure A-41 shows the derived `none_posedge_control_obs_invgclk` integrated clock-gating cell circuit.

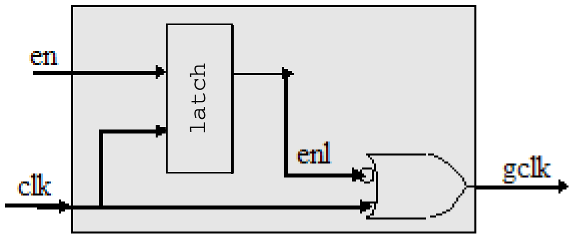
Figure A-41 none_posedge_control_obs_invgclk Circuit



latch_negedge_invgclk

Figure A-42 shows the derived `latch_negedge_invgclk` integrated clock-gating cell circuit.

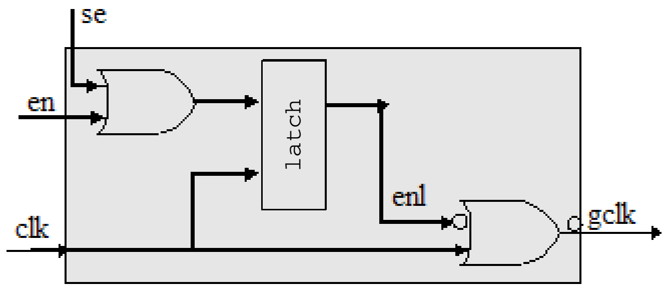
Figure A-42 latch_negedge_invgclk Circuit



latch_negedge_precontrol_invgclk

Figure A-43 shows the derived `latch_negedge_precontrol_invgclk` integrated clock-gating cell circuit.

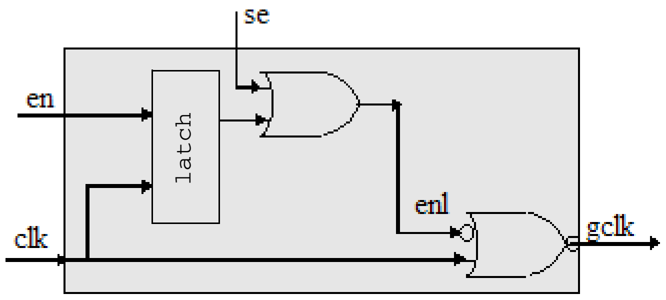
Figure A-43 latch_negedge_precontrol_invgclk Circuit



latch_negedge_postcontrol_invgclk

Figure A-44 shows the derived `latch_negedge_postcontrol_invgclk` integrated clock-gating cell circuit.

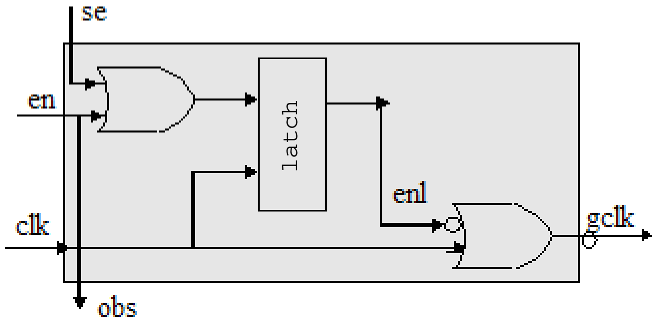
Figure A-44 latch_negedge_postcontrol_invgclk Circuit



latch_negedge_precontrol_obs_invgclk

Figure A-45 shows the derived `latch_negedge_precontrol_obs_invgclk` integrated clock-gating cell circuit.

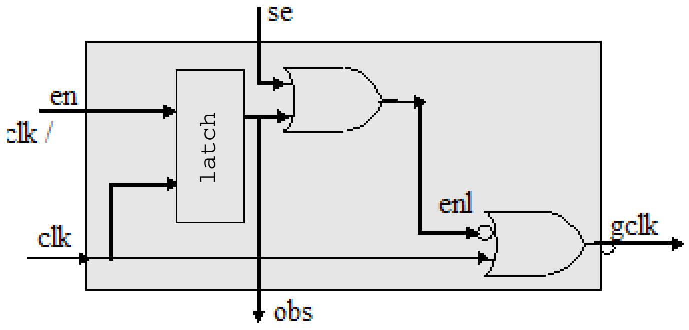
Figure A-45 latch_negedge_precontrol_obs_invgclk Circuit



latch_negedge_postcontrol_obs_invgclk

Figure A-46 shows the derived `latch_negedge_postcontrol_obs_invgclk` integrated clock-gating cell circuit.

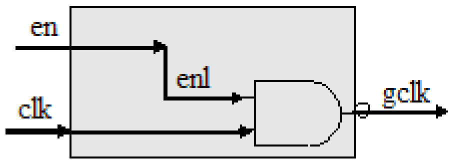
Figure A-46 latch_negedge_postcontrol_obs_invgclk Circuit



none_negedge_invgclk

Figure A-47 shows the derived `none_negedge_invgclk` integrated clock-gating cell circuit.

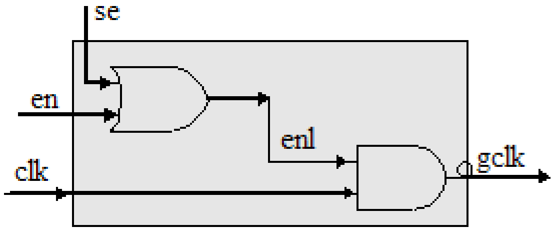
Figure A-47 none_negedge_invgclk Circuit



none_negedge_control_invgclk

Figure A-48 shows the derived `none_negedge_control_invgclk` integrated clock-gating cell circuit.

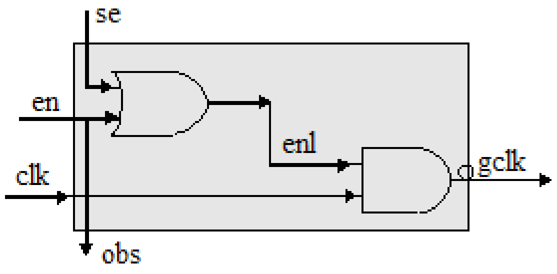
Figure A-48 none_negedge_control_invgclk Circuit



none_negedge_control_obs_invgclk

Figure A-49 shows the derived `none_negedge_control_obs_invgclk` integrated clock-gating cell circuit.

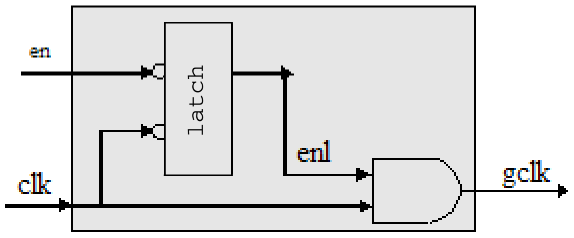
Figure A-49 none_negedge_control_obs_invgclk Circuit



latch_posedgeactivelow_invgclk

Figure A-50 shows the derived `latch_posedgeactivelow_invgclk` integrated clock-gating cell circuit.

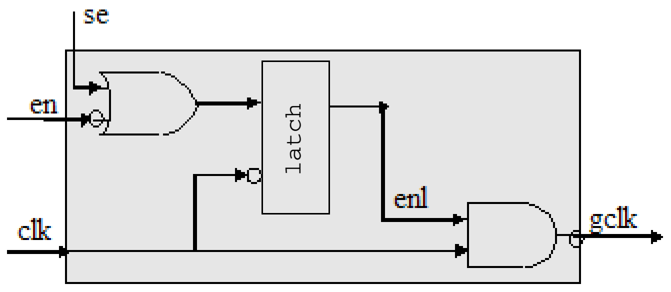
Figure A-50 latch_posedgeactivelow_invgclk Circuit



latch_posedgeactivelow_precontrol_invgclk

Figure A-51 shows the derived `latch_posedgeactivelow_precontrol_invgclk` integrated clock-gating cell circuit.

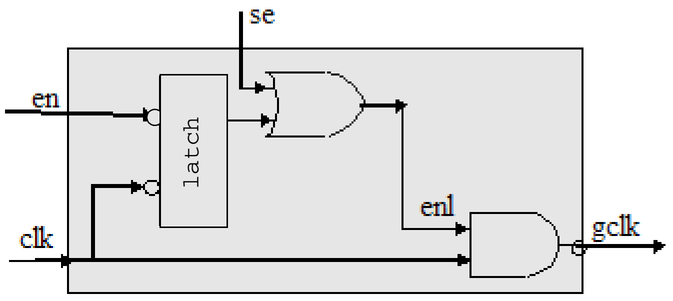
Figure A-51 latch_posedgeactivelow_precontrol_invgclk Circuit



latch_posedgeactivelow_postcontrol_invgclk

Figure A-52 shows the derived `latch_posedgeactivelow_postcontrol_invgclk` integrated clock-gating cell circuit.

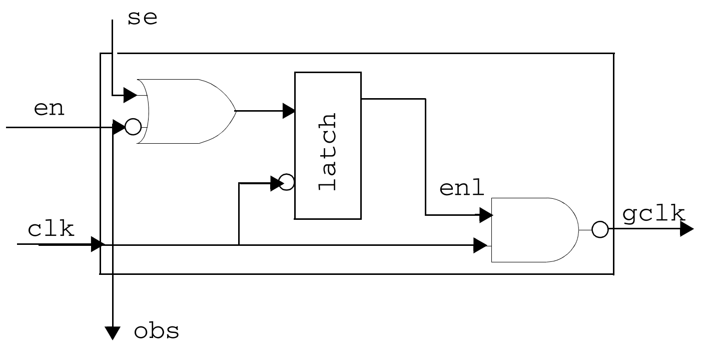
Figure A-52 latch_posedgeactivelow_postcontrol_invgclk Circuit



latch_posedgeactivelow_precontrol_obs_invgclk

Figure A-53 shows the derived `latch_posedgeactivelow_precontrol_obs_invgclk` integrated clock-gating cell circuit.

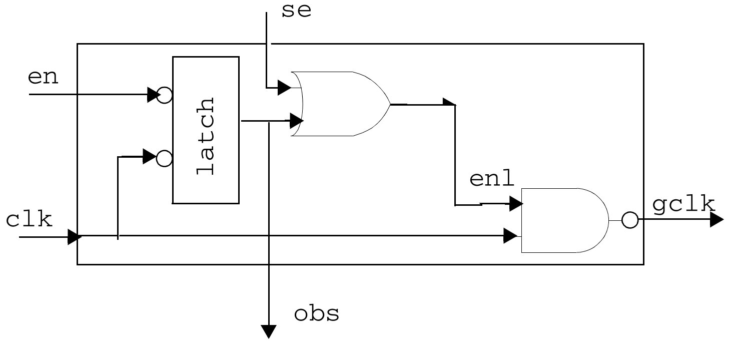
Figure A-53 latch_posedgeactivelow_precontrol_obs_invgclk Circuit



latch_posedgeactivelow_postcontrol_obs_invgclk

Figure A-54 shows the derived `latch_posedgeactivelow_postcontrol_obs_invgclk` integrated clock-gating cell circuit.

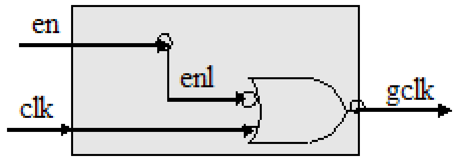
Figure A-54 latch_posedgeactivelow_postcontrol_obs_invgclk Circuit



none_posedgeactivelow_invgclk

Figure A-55 shows the derived `none_posedgeactivelow_invgclk` integrated clock-gating cell circuit.

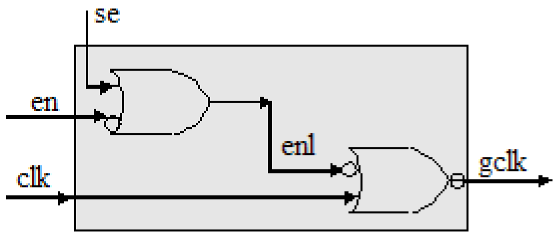
Figure A-55 none_posedgeactivelow_invgclk Circuit



none_posedgeactivelow_control_invgclk

Figure A-56 shows the derived `none_posedgeactivelow_control_invgclk` integrated clock-gating cell circuit.

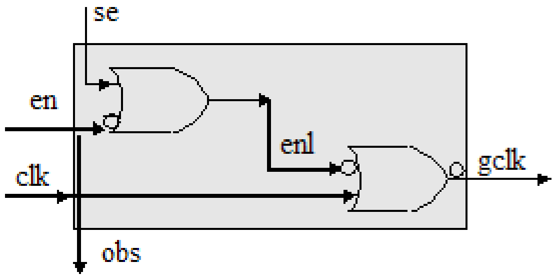
Figure A-56 none_posedgeactivelow_control_invgclk Circuit



none_posedgeactivelow_control_obs_invgclk

Figure A-57 shows the derived `none_posedgeactivelow_control_obs_invgclk` integrated clock-gating cell circuit.

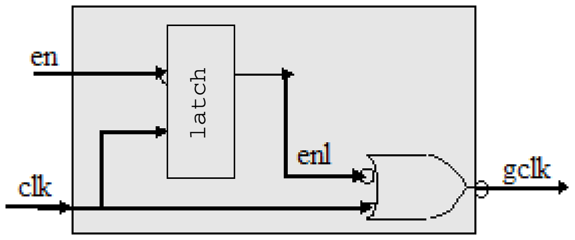
Figure A-57 none_posedgeactivelow_control_obs_invgclk Circuit



latch_negedgeactivelow_invgclk

Figure A-58 shows the derived `latch_negedgeactivelow_invgclk` integrated clock-gating cell circuit.

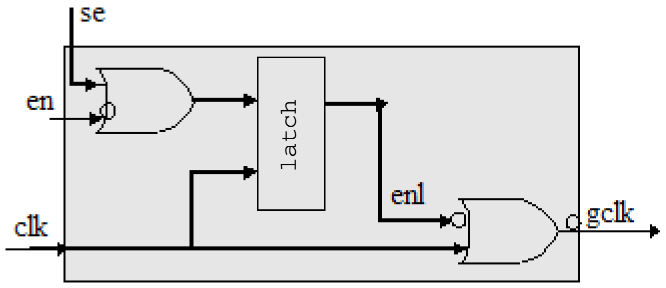
Figure A-58 latch_negedgeactivelow_invgclk Circuit



latch_negedgeactivelow_precontrol_invgclk

Figure A-59 shows the derived `latch_negedgeactivelow_precontrol_invgclk` integrated clock-gating cell circuit.

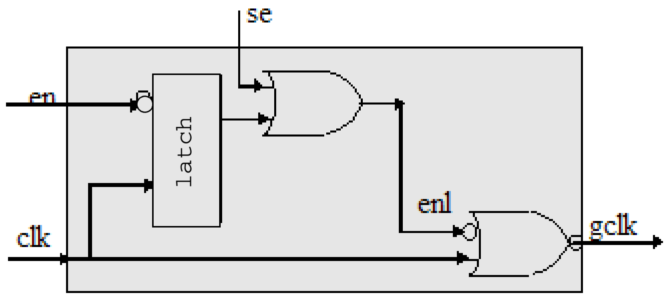
Figure A-59 latch_negedgeactivelow_precontrol_invgclk Circuit



latch_negedgeactivelow_postcontrol_invgclk

Figure A-60 shows the derived `latch_negedgeactivelow_postcontrol_invgclk` integrated clock-gating cell circuit.

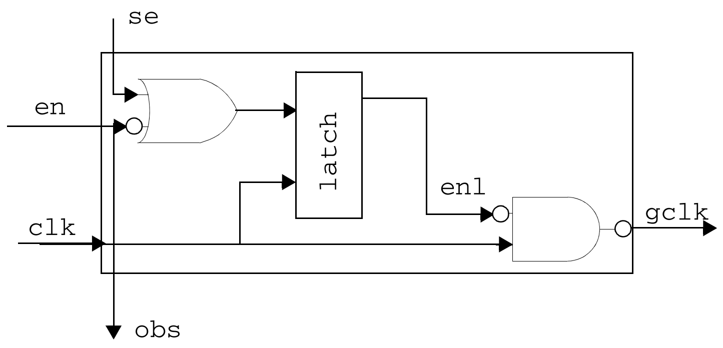
Figure A-60 latch_negedgeactivelow_postcontrol_invgclk Circuit



latch_negedgeactivelow_precontrol_obs_invgclk

Figure A-61 shows the derived `latch_negedgeactivelow_precontrol_obs_invgclk` integrated clock-gating cell circuit.

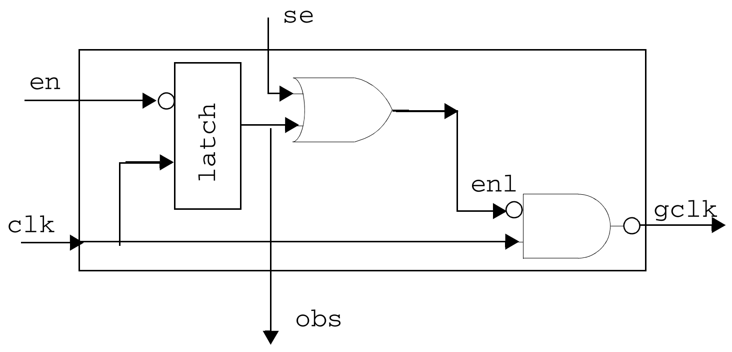
Figure A-61 latch_negedgeactivelow_precontrol_obs_invgclk Circuit



latch_negedgeactivelow_postcontrol_obs_invgclk

Figure A-62 shows the derived `latch_negedgeactivelow_postcontrol_obs_invgclk` integrated clock-gating cell circuit.

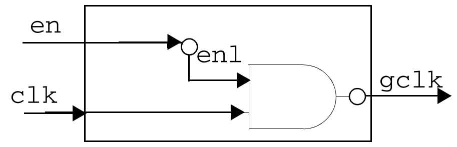
Figure A-62 latch_negedgeactivelow_postcontrol_obs_invgclk Circuit



none_negedgeactivelow_invgclk

Figure A-63 shows the derived `none_negedgeactivelow_invgclk` integrated clock-gating cell circuit.

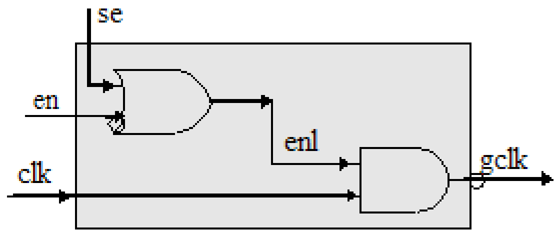
Figure A-63 none_negedgeactivelow_invgclk Circuit



none_negedgeactivelow_control_invgclk

Figure A-64 shows the derived `none_negedgeactivelow_control_invgclk` integrated clock-gating cell circuit.

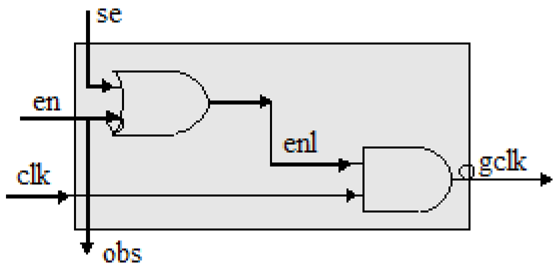
Figure A-64 none_negedgeactivelow_control_invgclk Circuit



none_negedgeactivelow_control_obs_invgclk

Figure A-65 shows the derived `none_negedgeactivelow_control_obs_invgclk` integrated clock-gating cell circuit.

Figure A-65 none_negedgeactivelow_control_obs_invgclk Circuit



Library Compiler Screener Checks

Library Compiler issues an error or a warning message, as applicable, when the following conditions occur:

- If a generic integrated clock-gating cell cannot be resolved as a specific type, Library Compiler resolves it as a black box and issues a warning message. If the integrated clock-gating cell is flip-flop based, the same warning message is issued.
- If you set the `clock_gating_integrated_cell` attribute to a specific value in a .lib file for generic integrated clock-gating cells, Library Compiler issues an error message. Library Compiler infers the clock-gating structure based on the `latch` group and the `function` attribute values specified in the cell. You cannot specify the value.
- If you set the `pin_is_active_high` attribute to indicate whether a pin is active high or active low, or if you set the `clock_gate_reset_pin` attribute to indicate whether a pin is a reset pin for a `latch_posedge_postcontrol` generic integrated clock gating cell, Library Compiler issues an error message. Library Compiler infers the values based on the pins. You cannot specify the values.

B

Library Conversion Support

Liberty Compiler allows you to convert existing libraries from one format to another by using the Model Adaptation System, which converts or merges one or more existing library files in Liberty (.lib) format to create a new library that is written out in Liberty format. You can also convert Liberty files to datasheets and Verilog models for use in characterization.

This chapter includes the following sections:

Model Adaptation System

The Model Adaptation System converts existing libraries from one format to another. The specific formatting operations are described in the following sections:

- Library Formatting Overview
- CCS Model Compaction and Expansion
- Variation-Aware Library Merging
- CCS Noise Merging
- CCS to ECSM Conversion
- CCS Noise to ECSM Noise Conversion
- CCS to NLDM Conversion
- VA-CCS to S-ECSM Conversion

Library Formatting Overview

Library Compiler can perform various types of library formatting operations. Each operation converts or merges one or more existing library files in Liberty (.lib) format to create a new library, which is written out in Liberty format. The set of formatting capabilities is called the Model Adaptation System.

To perform a formatting operation, you specify the type of operation, the input library names, the name of the new library file, and any options related to the operation in a command file. For example,

```
myfile contents:

set compact_ccs true
set input_library a90iz4.lib
set output_library a90iz4c.lib
```

Then, from the Library Compiler prompt,

```
lc_shell-xg-t> format_lib -f myfile
```

To view a list of the formatting options from Library Compiler, run the `-help` option:

```
lc_shell-xg-t> format_lib -help
```

If you need to perform multiple library formatting options, you must do so one at a time. You cannot perform multiple operations simultaneously.

The following library formatting operations are supported: CCS model compaction and expansion, variation-aware library merging, CCS Noise merging, CCS to ECSM conversion, CCS Noise to ECSM Noise conversion, CCS to NLDM conversion, and VA-CCS to S-ECSM conversion.

CCS Model Compaction and Expansion

The `compact_ccs` operation converts library cell models from expanded CCS format to compact CCS format. The compact format offers the same high accuracy as conventional expanded CCS data, but uses much less space in the library files. The theory of CCS data compaction and the Liberty syntax for CCS data are described in the section “Advanced Compact CCS Timing Model Support” in the *Library Compiler Modeling Timing, Signal Integrity, and Power in Technology Libraries User Guide*.

To convert conventional CCS data to compact CCS data, use the `compact_ccs` option, set the input library containing expanded CCS data, and set the new library to contain compact CCS data. For example,

```
set compact_ccs true
set input_library xlib82.lib
set output_library xlib82c.lib
```

The `expand_ccs` option performs the opposite function. It expands a library containing compact CCS modeling information into a library containing the original CCS modeling information. For example,

```
set expand_ccs true
set input_library xlib82c.lib
set output_library xlib82exp.lib
```

Variation-Aware Library Merging

You can use the Model Adaptation System to merge multiple CCS libraries characterized at different variation parameter values into a single variation-aware CCS library. A single library is easier to use for variation analysis than multiple libraries.

To combine multiple libraries, use the following command file syntax:

```
set va_merge true
set nominal_library {nom_lib_name par1 val1 par2 val2 ...}
set va_library_list {lib1_name val1 val2 ... \
                     lib2_name val1 val2 ... \
                     lib3_name val1 val2 ... \
                     lib4_name val1 val2 ... \
                     ...                     \
                     lib2N_name val1 val2 ... }
```

```
set output_library out_lib_name
[set slew_indexes {index1 index2 ...}]
[set load_indexes {index1 index2 ...}]
```

For example, suppose that you have created a set of five libraries with variation in two parameters, `len` and `vt`. There is a single nominal library called nom.lib, libraries characterized at lower and higher `len` values called lenlow.lib and lenhi.lib, and libraries characterized at lower and higher `vt` values called vtlow.lib and vthi.lib. To merge these five libraries into a single variation-aware CCS library, you would use a command file similar to the following:

```
set va_merge true
set nominal_library { nom.lib    len 100.0  vt 0.24 }
set va_library_list { lenlow.lib      95.0     0.24 \
                      lenhi.lib      105.0     0.24 \
                      vtlow.lib      100.0     0.22 \
                      vthigh.lib     100.0     0.26 }
set output_library va_lvt.lib
```

You specify the nominal library using `nominal_library` and the off-nominal libraries using `va_library_list`. With the nominal library name, you specify all the parameter names and their respective nominal values. With each off-nominal library in the library list, you specify the full list of parameter values for that library in exactly the same order as for the nominal library.

The libraries in the library list can be specified in any order. However, for clarity, it is suggested that you use the order shown in the foregoing example: the low and high libraries for the first variable, followed by the low and high libraries for the second variable, and so on.

You can optionally specify a list of slew index values that are to be retained from the slew data tables in the off-nominal input libraries. For example, to retain only the first, second, third, and fifth slew index values, use `slew_indexes {1 2 3 5}`. This setting affects only the libraries specified with `library_list`. All index values in the nominal library are always retained.

Similarly, by using the `load_indexes` option, you can specify a list of load index values that are to be retained from the load data tables in the off-nominal input libraries.

When you run the formatting operation, the Model Adaptation System reads in the nominal and off-nominal variation libraries in Liberty format and combines them into a single variation-aware CCS library. It writes out the new library in Liberty format, using compact CCS models. You can then compile the new library using Library Compiler.

In PrimeTime VX, when you use a single merged variation-aware library instead of a set of libraries, the `set_variation_library` command is not required because the merged library already contains information about the variation relationships between the original libraries. You only need to link in the single variation-aware library and use the `create_variation` and `set_variation` commands in the usual manner to specify the distribution of parameter values for timing analysis. For more information, see “Using a Single CCS-Based Variation-Aware Library” in the *PrimeTime VX User Guide*.

CCS Noise Merging

The current release of Liberty NCX does not support direct generation of CCS noise models. Instead, you can add CCS noise models to an existing library by using the Make CCS Noise utility. For more information, see the *Make CCS Noise User Guide*.

If you already have two different libraries containing CCS timing and CCS noise models, you can use the `ccsn_merge` operation to merge the two libraries into a single library containing both CCS timing and CCS noise models. The two input libraries must be in Liberty format and must contain the same cells.

To merge the two libraries, set `ccsn_merge` and specify the names of the CCS timing library file, the CCS noise library file, and the new merged library to be generated, as shown in the following example:

```
set ccsn_merge true
set timing_library x182c_tim.lib
set noise_library x182c_noise.lib
set output_library x182c.lib
```

When you run the formatting operation, the Model Adaptation System reads in the CCS timing and CCS noise libraries in Liberty format and combines them into a single new library in Liberty format. You can then compile the new library with Library Compiler.

CCS to ECSM Conversion

Effective Current Source Model (ECSM) is a library modeling standard developed at Cadence Design Systems, Inc. for representing cell behavior as current-source elements. You can convert a library containing CCS timing models to a new library containing ECSM models by using the `ccs2ecsm` option.

To specify whether to convert CCS timing data, CCS noise data, or both, set the `mode` option. The valid values for `mode` are `timing` and `noise`. If you specify `timing`, the CCS timing data will be converted but not the noise data. If you specify `noise`, the CCS noise data will be converted but not the timing data. You can convert both the timing and noise data by specifying the `timing` and `noise` values separated by a space, as shown in the following example:

```
set mode {timing noise}
```

When `mode` is set to `timing`, CCS timing data is converted to ECSM, and any CCS noise or power data is stripped out. Similarly, when `mode` is set to `noise`, CCS noise is converted to ECSM noise and any CCS timing or power data is stripped out from the output library.

If you do not specify the `mode` option, `ccs2ecsm` converts the timing data only.

The general syntax for performing a CCS to ECSM conversion is:

```
set ccs2ecsm true
set library_list { lib1 lib2 lib3 ... }
set output_library out_lib_name
set process value
set voltage value
set temperature value
set capacitance_mode first|second|ave|min|max]
set sample { v1 v2 v3 v4 ... }
```

To generate a new ECSM library, you can specify a single CCS library or multiple CCS libraries as input. If you specify a single library, the Model Adaptation System performs a straightforward conversion of the library. The operating conditions of the output library match those of the input library. For example,

```
set ccs2ecsm true
set library_list t52_nom.lib
set output_library t52ecsm_nom.lib
```

If you specify multiple libraries, the Model Adaptation System performs process, voltage, and temperature (PVT) scaling between the provided CCS libraries to obtain the cell behavior at a specified set of PVT conditions. Then it generates a single output library in ECSM format for that set of operating conditions.

For example, to generate a single NLDM library for voltage = 1.15 and temperature = 70 by scaling between four provided CCS libraries:

```
set ccs2ecsm true
set library_list { tv2c_p0v0t0.lib \
                   tv5c_p0v0t1.lib \
                   tv5c_p0v1t0.lib \
                   tv5c_p0v1t1.lib }
set output_library tv5c_ecsm.lib
set voltage 1.15
set temperature 70.0
```

When you run the formatting operation, the Model Adaptation System reads in the CCS library or libraries in Liberty format. If you use any of the `process`, `temperature`, or `voltage` options, you must provide multiple CCS libraries so that the Model Adaptation System can perform scaling between them to obtain models at the target operating conditions. The Model Adaptation System generates equivalent ECSM models and writes them out into the new library in Liberty format.

The CCS cell receiver model uses two capacitance values to more accurately model the Miller effect. The two values, called the first and second capacitance values, are used at the beginning and end of each logic transition, respectively. The ECSM standard, however, supports only a single receiver capacitance value. By default, the conversion process uses only the first CCS capacitance value and ignores the second value.

You can optionally specify a different conversion convention by using the `capacitance_mode` option. It can be set to `first`, `second`, `ave`, `min`, or `max`, causing the conversion process to use the first value, second value, the average of both values, the smaller of the two values, or the larger of the two values, respectively. For example,

```
set ccs2ecsm true
set library_list t52_nom.lib
set output_library t52ecsm_nom.lib
set capacitance_mode ave
```

This example causes all ECSM receiver models in the generated library to use the average of the corresponding first and second CCS capacitance values in the source library.

The conversion process generates ECSM data tables having discrete voltage sample points between the full voltage swing between 0.0 and the rail voltage. By default, the following voltage sample points are used to generate the data tables:

```
0.05 0.10 0.20 0.30 0.40 0.50 0.60 0.70 0.80 0.90 0.95
```

To specify a different set of voltage sample points, set the `sample` variable as in the following example,

```
set ccs2ecsm true
set library_list t52_nom.lib
set output_library t52ecsm_nom.lib
set sample {0.02 0.05 0.10 0.20 0.30 0.40 0.50 \
            0.60 0.70 0.80 0.90 0.95 0.97}
```

CCS Noise to ECSM Noise Conversion

You can convert a library containing CCS noise models to a new library containing Effective Current Source Model (ECSM) noise models by using the `ccs2ecsm` option. To generate a new ECSM library, you must specify a single CCS noise input library only. If you specify multiple input libraries, `ccs2ecsm` issues a warning message and deletes the noise information.

Set the following commands in a command file to convert a library with CCS noise models to a library with ECSM noise models:

```
set ccs2ecsm true
set library_list library_name.lib
set output_library   out.lib
set mode {timing noise}
set log_file out.log
```

As `mode` is set to `timing` and `noise`, it will convert both CCS timing and CCS noise models to ECSM.

To specify whether to convert CCS timing data, CCS noise data, or both, set the `mode` option in the configuration file. The valid values for `mode` are `timing` and `noise`. If you specify `timing`, the CCS timing data will be converted but not the noise data. If you specify `noise`, the CCS noise data will be converted but not the timing data. You can convert both the timing and noise data by specifying the `timing` and `noise` values separated by a space, as shown in the following example:

```
set mode {timing noise}
```

If you do not specify the `mode` option, `ccs2ecsm` converts the timing data only.

To specify the voltage variation, set the following options in the configuration file:

- `set ecsm_vhtolerance` *`value`*

  The value is an absolute floating-point number that specifies the variation in voltage that is considered to be negligible when the signal is high. The default is 0.

- `set ecsm_vltolerance` *`value`*

  The value is an absolute floating-point number that specifies the variation in voltage that is considered to be negligible when the signal is low. The default is 0.

CCS to NLDM Conversion

Nonlinear Delay Model (NLDM) is an older, well-established modeling standard for representing cell behavior by using voltage-supply elements. You can convert a library containing CCS timing models to a new library containing NLDM models by using the `ccs2nldm` option.

To generate a new NLDM library, you can specify a single CCS library or multiple CCS libraries as input. If you specify a single library, the Model Adaptation System performs a straightforward conversion of the library. In such a case, the operating conditions of the output library match those of the input library. For example,

```
set ccs2nldm true
set library_list t52_nom.lib
set output_library t52nldm_nom.lib
```

If you specify multiple libraries, the Model Adaptation System performs process, voltage, and temperature (PVT) scaling between the provided CCS libraries to obtain the cell behavior at a given set of PVT conditions. Then it generates a single output library in NLDM format for that set of operating conditions.

For example, to generate a single NLDM library at process = 1.08, voltage = 1.15, and temperature = 70 by scaling between eight provided CCS libraries, use this syntax:

```
set ccs2nldm true
set library_list { tv2c_p0v0t0.lib \
                   tv5c_p0v0t1.lib \
                   tv5c_p0v1t0.lib \
                   tv5c_p0v1t1.lib }
set output_library tv5c_nldm.lib
set voltage 1.15
set temperature 70.0
```

Liberty NCX provides the following `nldm_cap` values to generate the input pin capacitance for a scaled NLDM library:

- `linear`

  The `linear` value linearly interpolates the NLDM input capacitance found in the two corner libraries.

- `c1`

  The `c1` value linearly scales the receiver model from the two corner libraries and uses the average of all `receiver_capacitance1` values in the scaled receiver model. The `c1` value is the default for `nldm_cap`.

- `avg`

  The `avg` value linearly scales the receiver model from the two corner libraries. The `avg` value then calculates the average of all `receiver_capacitance1` values and all `receiver_capacitance2` values in the scaled receiver model and uses the average of those values.

To use linear interpolation to calculate NLDM input pin capacitance for a scaled library, set the `nldm_cap` variable in the configuration file, as shown in the following example:

```
set nldm_cap linear
```

When you run the formatting operation, the Model Adaptation System reads in the single CCS library or the multiple CCS libraries in Liberty format. If you use any of the `process`, `temperature`, or `voltage` options, you must provide multiple CCS libraries so that the Model Adaptation System can perform scaling between them to obtain models at the target operating conditions. The Model Adaptation System generates equivalent NLDM models and writes them out into the new library in Liberty format.

VA-CCS to S-ECSM Conversion

You can use the Model Adaptation System to convert VA-CCS (variation-aware CCS) libraries to S-ECSM (Statistical Effective Current Source Model) libraries. ECSM is a library modeling standard developed by Cadence Design Systems, Inc. for representing cell behavior as current-source models. S-ECSM is an extension to the ECSM timing modeling format.

Liberty NCX can convert a VA-CCS library to S-ECSM library by using the `format_lib` command. Set the following options in the `format_lib` configuration file:

```
set vaccs2secsm true
set input_library ncx.lib
set output_library out.lib
set s_ecsm_param_class_unit_list "param1 class1 unit1…paramn classn unitn"
set capacitance_mode [first|second|max|min|ave]
set sample “value”
```

The options are defined as follows:

`vaccs2secsm`

Converts a variation-aware CCS library to an S-ECSM library. If `vaccs2secsm` is set to `true`, `format_lib` generates an S-ECSM library from a variation-aware CCS library. The option is set to `false` by default.

`s_ecsm_param_class_unit_list`

Sets the S-ECSM parameters, class, and units. The valid values are *`param1`*, *`class1`*, and *`unit1`* through *`paramn`*, *`classn`*, and *`unitn`*. The values are user-defined. The *`unitn`* value can include a multiplier of 1, 10, or 100. Valid values for *`classn`* are `global`, `local`, `random`, and `environmental`.

Valid values for *`unitn`* are `nan`, `nm`, `um`, `A`, `mV`, `V`, and `C`. The `nan` value indicates no unit for the *`paramn`* parameter.

`capacitance_mode`

Specifies the conversion mode of the receiver capacitance. The following values are valid:

- `first`

  Specifies that `ecsm_capacitance_sensitivity` is derived from the `va_receiver_capacitance1_rise` and `va_receiver_capacitance1_fall` values. The `first` value is the default for `capacitance_mode`.

- `second`

  Specifies that `ecsm_capacitance_sensitivity` is derived from the `va_receiver_capacitance2_rise` and `va_receiver_capacitance2_fall` values.

- `max`

  Specifies that `ecsm_capacitance_sensitivity` is the maximum value between `va_receiver_capacitance1_rise` and `va_receiver_capacitance2_rise` or `va_receiver_capacitance1_fall` and `va_receiver_capacitance2_fall`.

- `min`

  Specifies that `ecsm_capacitance_sensitivity` is the minimum value between `va_receiver_capacitance1_rise` and `va_receiver_capacitance2_rise` or `va_receiver_capacitance1_fall` and `va_receiver_capacitance2_fall`.

- `ave`

  Specifies that `ecsm_capacitance_sensitivity` is the average value between `va_receiver_capacitance1_rise` and `va_receiver_capacitance2_rise` or `va_receiver_capacitance1_fall` and `va_receiver_capacitance2_fall`.

`Example`

Specifies the `ecsm_waveform` and `ecsm_waveform_sensitivity` sample points with a space-separated value. The `sample` values are normalized and must be in the range of 0 to 1. The default is “0.05 0.10 0.20 0.30 0.40 0.50 0.60 0.70 0.80 0.90 0.95”.

When you run the formatting operation, the Model Adaptation System reads in the variation-aware CCS library in Liberty format and generates equivalent S-ECSM models and writes them out into the new library in Liberty format.

Generating Datasheets

Characterization tools support datasheets, which provide detailed information about a cell, in addition to Liberty models. Generating datasheets from Liberty .lib files is helpful if you are new to characterization or if you are creating new libraries rather than recharacterizing existing libraries with new process models. This section describes how to generate datasheets from Liberty .lib files for use in characterization. It includes the following sections:

- Datasheet Generation Overview
- Datasheets in HTML Format
- Datasheets in Text Format
- HTML and Text Datasheet Details

Datasheet Generation Overview

The characterization flow requires that you first run Liberty NCX and generate a .lib file. Next, you generate a datasheet from the .lib file using Library Compiler. This ensures that the datasheet matches the .lib file exactly. Although datasheet generation is performed in Library Compiler, it requires a Liberty NCX license and will fail if it is not available.

You can generate a datasheet in HTML format or text format. When you generate a datasheet, Library Compiler generates a datasheet from a .lib file and creates a *cell*.html file for each cell in the library. You must set `datasheet_enable` to generate a datasheet in the specified format:

- `datasheet_enable: text` generates a text datasheet.
- `datasheet_enable: html` generates an HTML datasheet.
- `datasheet_enable: true` generates a text datasheet for backward compatibility.

Use the following commands to generate datasheets:

1. To enable datasheet generation, set the `datasheet_enable` variable to `text` or `html`. By default, the variable is set to html, and a datasheet is generated in HTML format.

   ```
   set datasheet_enable [text | html]
   ```

2. To create a datasheet from a .lib file, run the `write_lib` command in lc_shell and specify the .lib library name as shown:

   ```
   write_lib -format datasheet library_name
   ```

3. By default, datasheets are written to *library_name*_datasheet for a text datasheet, and *library_name*_datasheet.html for an HTML datasheet. If you want the datasheet to be written to a directory other than the default, use the one of the following commands:

   For a text datasheet,

   ```
   set datasheet_output_dir directory_name
   ```

   For an HTML datasheet,

   ```
   set datasheet_html_output_dir directory_name
   ```

Table B-1 provides an overview of the commands that are available to specify datasheet sections, including the file format and default settings. Each row in the table, except the first three rows, represents a section in the datasheet. You can set each command directly in lc_shell or include them in a configuration file and source the file by running the `source config_file_name` command in lc_shell.

To generate datasheets in HTML format, you must provide an HTML datasheet template. A default HTML template is available if you do not provide a custom template.

Table B-1 Commands Used to Specify Datasheet Details

Command or section	**Text format**	**HTML format**
Enabling datasheet generation	`set datasheet_enable text` (Enabled by default)	`set datasheet_enable html`
Output directory location	Default output directory library_name_datasheet. To specify a directory, use `set datasheet_output_dir` *`directory_name`*	Default output directory: library_name_datasheet_html To specify a directory, use `set datasheet_html_output_dir` *`directory_name`*
HTML template location	n/a	Default template location: $synopsys_root/auxx/template/datasheet_te mplate.html To override the default template, use `set datasheet_html_template` *`file_name`*
Logo	n/a	To include a logo, use `set datasheet_html_logo_file` *`file_name`* (No default setting)
Cell name	Derived from .lib file	Derived from .lib file
Cell description file	n/a	By default, HTML datasheet generation leaves Cell Description empty, unless it finds the following directory: `library_name_cdesc` To override this, use `set datasheet_html_cell_desc_dir directory_name` If this attribute described is defined, by default, it looks for the .cdesc file extension. To override the default suffix, use `set datasheet_html_cell_desc_suffix` *`string`*

Table B-1 Commands Used to Specify Datasheet Details (Continued)

Command or section	**Text format**	**HTML format**
Function description file	Derived from .lib file	Default behavior: Derived from .lib file To specify a directory, use `set datasheet_html_func_desc_dir` *`directory_name`* Default file suffix: .fdesc To specify a file suffix, use `set datasheet_html_func_desc_suffix` *`string`*
Symbol	n/a	Default file location: library_name_sym directory in the current working directory To specify a directory, use `set datasheet_html_symbol_dir` *`directory_name`* Default file suffix: .sym To specify a file suffix, use `set datasheet_html_symbol_suffix` *`string`*
Schematic	n/a	Default file location: library_name_sch directory in the current working directory To specify a directory, use `set datasheet_html_schematic_dir` *`directory_name`* Default file suffix: .sch To specify a file suffix, use `set datasheet_html_schematic_suffix` *`string`*
Port names	Derived from .lib file	Derived from .lib file
Cell area	Derived from .lib file	Derived from .lib file
Pin capacitance	Derived from .lib file	Derived from .lib file
Delay data	Derived from .lib file	Derived from .lib file

Table B-1 Commands Used to Specify Datasheet Details (Continued)

Command or section	**Text format**	**HTML format**
Power data	n/a	Derived from .lib file

Datasheets in HTML Format

Library Compiler creates *cell*.html files in the user-defined *library_name*_datasheet_html directory under the current working directory. If the *library_name*_datasheet_html directory already exists under the current working directory, the newly generated *cell*.html files overwrite the existing files in the directory, and Library Compiler issues a warning message.

Each cell_name.html datasheet provides the following information for a cell:

- Logo: *logo_file*

  See "Logo (HTML)" on page B-17.

- Cell: *cell_name*

  See "Cell (HTML and Text)" on page B-17.

- Cell Description File: *directory_name*

  See "Cell Description File (HTML Only)" on page B-17.

- Function Description

  See "Function Description (HTML and Text)" on page B-18.

- Symbol: *symbol_file*

  See "Symbol (HTML Only)" on page B-22.

- Schematic: *schematic_file*

  See "Schematic (HTML Only)" on page B-23.

- Port Names

  See "Port Names (HTML and Text)" on page B-24.

- Cell Area: *cell area*

  See "Cell Area (HTML and Text)" on page B-25.

- Pin Capacitance

  See "Pin Capacitance (HTML and Text)" on page B-25.

- Delay Data

  See "Delay Data (HTML and Text)" on page B-27.
- Constraint Data

  See "Constraint Data (HTML and Text)" on page B-29.
- Power Data

  See "Power Data (HTML Only)" on page B-31.

Datasheets in Text Format

Library Compiler creates cell.txt files in the user-defined library_name_datasheet directory under the current working directory. If the library_name_datasheet directory already exists under the current working directory, the newly generated cell.txt files overwrite the existing files in the directory, and Library Compiler issues a warning message.

Each cell_name.txt datasheet provides the following information for a cell:

- Cell: *cell_name*

  See "Cell (HTML and Text)" on page B-17.
- Function Description

  See "Function Description (HTML and Text)" on page B-18.
- Port Names

  See "Port Names (HTML and Text)" on page B-24.
- Cell Area: *cell area*

  See "Cell Area (HTML and Text)" on page B-25.
- Pin Capacitance

  See "Pin Capacitance (HTML and Text)" on page B-25.
- Delay Data

  See "Delay Data (HTML and Text)" on page B-27.
- Constraint Data

  See "Constraint Data (HTML and Text)" on page B-29.

HTML and Text Datasheet Details

The valid arguments to use for the cell.html and cell_name.txt files are described in the following subsections.

Logo (HTML)

If you want a logo included in the datasheet, you must specify it in a logo file. There is no default setting. To include a logo, use the following command:

```
set datasheet_html_logo_file file_name
```

There is no default setting.

Syntax

```
Logo: logo_file
```

Example

```
set datasheet_html_logo_file logo_URL or path to file
```

This option is available only for HTML datasheets.

Cell (HTML and Text)

The datasheet generator takes the current library cell name from the input Liberty (.lib) file and writes it to the first line of the datasheet file, as shown:

Syntax

```
Cell: cell_name
```

Example

```
Cell: AND2X2
```

This option is available for both HTML and text datasheets.

Cell Description File (HTML Only)

An optional cell description section can be added to the HTML Datasheet. If nothing is specified, this section is says “None”. However, you can provide information for the cell description for each cell. The default file location is the library_name_cdesc_dir directory in the current working directory. To specify a directory, use the following command:

```
set datasheet_html_cell_desc_dir directory_name
```

The default file suffix is .cdesc. To specify a different file suffix, use the following command:

```
set datasheet_html_cell_desc_suffix string
```

The datasheet generator relies on user input in the configuration file to indicate the location of the cell description file. You can locate the cell description file for the current cell in either of the following two ways:

- If there is a map.$datasheet_html_cell_desc_suffix under the $datasheet_html_cell_desc_dir directory, then the datasheet generator checks the cell's name in the mapping file, `include` wildcard. If a mapped file to this cell is found, the datasheet generator uses this file as the cell description file for this cell.
- If no $datasheet_html_cell_desc_dir or map.$datasheet_html_cell_desc_suffix exists, or the datasheet generator does not find a mapped cell description file to this cell, then it searches the cell_name.$datasheet_html_cell_desc_suffix under the $datasheet_html_cell_desc_dir directory, using it as the cell description file for this cell.

If the datasheet generator cannot find the cell description file by using one of these methods, it replaces this section with `None`.

Syntax

```
None [This is the default if nothing is specified]
```

Example

```
Cell Description:
Inverter
```

This option is available only for HTML datasheets.

Function Description (HTML and Text)

If the function description of the cell is included in the input Liberty file, it is written out in the Function Description section of the datasheet. For black box cells that have no function description in the .lib file, the datasheet generator issues a warning and adds a comment in the Function Description section of the datasheet.

Function Description for HTML Datasheets

By default the function description from the input .lib is used. However, for HTML datasheets, you can override this by specifying the location and file names of the custom function description.

For a function description file, the default file location is the library_name_fdesc directory in the current working directory. To specify a directory, use the following command:

```
set datasheet_html_func_desc_dir directory_name
```

The default file suffix is .fdesc. To specify a different file suffix, use the following command:

```
set datasheet_html_func_desc_suffix string
```

You can locate the function description file for the current cell in either of the following two ways:

- If there is a map.$datasheet_html_func_desc_suffix under the $datasheet_html_func_desc_dir directory, then the datasheet generator checks the cell's name in the mapping file `include` wildcard. If a mapped file to this cell is found, the datasheet generator uses this file as the function description file for this cell.
- If no $datasheet_html_function_desc_dir or map.$datasheet_html_func_desc_suffix exists, or the datasheet generator does not find a mapped function description file to this cell, then it searches the cell_name.$datasheet_html_func_desc_suffix under the $datasheet_html_func_desc_dir directory, using it as the function description file for this cell.

If the datasheet generator cannot find the function description file by using one of these methods, it uses the information from the function statement in the input .lib.

Example

```
set datasheet_html_func_desc_dir ./function_description_dir
set datasheet_html_func_desc_suffix .txt
```

This option is available for both HTML and text datasheets.

Default Function Description Arguments

The following list shows the function description types.

- Combinational function

  The combinational function format provides the Boolean function for a gate and has the following syntax:

  ```
  Function: output_pin = Boolean_expression
        output_pin = Boolean_expression
         ...
  ```

  The arguments are as follows:

output_pin

The output pin is the corresponding output or bidirectional pin name found in the input Liberty file.

Boolean_expression

The Boolean expression is retrieved directly from the `function` attribute in the input Liberty file.

The following examples shows the .lib syntax followed by the datasheet equivalent:

.lib Example

```
pin (Y)
function: A ^ B' & C;
}
```

Datasheet Example

```
Function: Y = A ^ B' & C;
```

- Three-State Function

  The three-state function format is similar to the combinational function. Three-state functions are generated in the datasheet when a bidirectional or output pin has a `three_state` attribute specified in input Liberty file. The three-state function has the following syntax:

  ```
  Function: output_pin = input_pin
  Three state: output_pin = Boolean_expression
  ```

output_pin

  The output pin is the name of the output or bidirectional pin with a three-state function in the input Liberty file.

Boolean_expression

  The Boolean expression is retrieved directly from the corresponding `three_state_function` attribute value in the input Liberty file.

The following examples shows the .lib syntax followed by the datasheet equivalent:

.lib Example

```
pin (O) {
         direction : "output";
         function : "I";
         three_state : "EN'";
         ...
  }
```

Datasheet Example

```
Function: O = (I)

Three state: O = (EN')
```

- Sequential Function

  The sequential function format is a state table and has the following syntax:

  ```
  statetable:
   state_table_pin_list
   table_content

  statetable:
  ```

```
 state_table_pin_list
 table_content

statetable:
output_pin = Boolean_expression
...
```

state_table_pin_list

The state table pin list is a comma-separated list and contains input pins followed by one output pin with a current state and a next state. It has the following characteristics:

- The output pin name is the real output pin name.
- If the current state pin name is *output*, the next state pin name is *output*+. For example, if the current state is `Q`, the next state is `Q+`.
- One state table has exactly one output pin.
- The order of the input pin list determines the order of the pins in the state table.
- Generally, there is a state table for each output of the cell; however, in some cases, a state table might not exist. For example, `QN` might be a logic function of `Q` (for instance, inverted). The datasheet generator writes a Boolean expression for `QN`, using `Q`.The state table has the following syntax:

```
input_state [input_state…] current_state next_state
 …
input_state
```

The input state list is a space character (“ ”) separated list of input state values. Valid characters for the input values are `0`, `1`, `r`, `f`, `b`, `?`, `x`, and `*`.

current_state

The current state is the output current state value, and the valid characters are `0`, `1`, `x`, `?`, and `b`.

next_state

The next state is the next state value, and the valid characters are `0`, `1`, `x`, and `-`.

These state table values have the following definitions:

- `0` – A low state.
- `1` – A high state.
- `x` – unknown.
- `?` – an iteration of 0, 1 and x.

- `b` – an iteration of 0 and 1.
- `r` – The same as 01.
- `f` – The same as 10.
- `*` – The same as ??.
- `-` – No change.

  Each row defines the output, based on the current state and the particular combinations of input values.

  The following examples show a latch function with `Q` and `QN` outputs and a flip-flop with asynchronous inputs and with `Q` and `QN` outputs.

  Latch Function Example

```
statetable:
CDN   D     E     SDN   Q     Q+
0     ?     ?     1     ?     0
1     1     1     ?     ?     1
1     ?     0     1     ?     -
?     0     1     1     ?     0
?     ?     ?     0     ?     1

statetable:
SDN   D     E     CDN   QN    QN+
0     ?     ?     1     ?     0
1     0     1     ?     ?     1
1     ?     0     1     ?     -
?     1     1     1     ?     0
?     ?     ?     0     ?     1
```

  Flip-Flop Example

```
statetable:
CDN   CP    D     Q     Q+
0     ?     ?     ?     0
1   (01)    1     ?     1
1   (1?)    ?     ?     -
1   (?0)    ?     ?     -
?   (01)    0     ?     0

statetable:
QN = (not Q)
```

Symbol (HTML Only)

For a symbol file, the default file location is the library_name_sym directory in the current working directory.

To specify a directory, use the following example:

```
set datasheet_html_symbol_dir directory_name
```

Default file suffix: .sym

To specify a file suffix, use the following example:

```
set datasheet_html_symbol_suffix string
```

The datasheet generator relies on user input in the configuration file to indicate the location of the symbol file. You can locate the symbol file for the current cell in either of the following two ways:

- If there is a map.$datasheet_html_symbol_suffix file under the $datasheet_html_symbol_dir directory, then the datasheet generator checks the cell's name using the mapping-file `include` wildcard. If a mapped file to this cell is found, the datasheet generator uses this file as the symbol file for this cell.
- If no map.$datasheet_html_symbol_suffix exists, or the datasheet generator does not find a mapped symbol file to this cell, then it searches for the cell_name.$datasheet_html_symbol_suffix file under the $datasheet_html_symbol_dir directory as the symbol file for this cell.

If it cannot find the symbol file by using one of these methods, the datasheet generator replaces `{symbol_file}` with `None`.

The format of the symbol file should be JPEG, BMP, or PNG.

The datasheet generator does not display a symbol picture unless you specify that it do so.

Syntax

```
Symbol: symbol_file
```

Example

```
set datasheet_html_symbol_suffix symbol_URL
```

This option is available only for HTML datasheets.

Schematic (HTML Only)

For a schematic file, the default file location is the library_name_sch directory in the current working directory.

To specify a directory, use the following command:

```
set datasheet_html_schematic_dir directory_name
```

Default file suffix: .sch

To specify a file suffix, use the following command:

```
set datasheet_html_schematic_suffix string
```

The datasheet generator relies on user input in the configuration file to indicate the location of the schematic file. You can locate the schematic file for the current cell in either of the following two ways:

- If there is a map.$datasheet_html_schematic_suffix file under the $datasheet_html_schematic_dir directory, then the datasheet generator checks the cell's name using the mapping-file `include` wildcard. If a mapped file to this cell is found, the datasheet generator uses this file as the schematic file for this cell.
- If no $map.$datasheet_html_schematic_suffix exists, or the datasheet generator does not find a mapped symbol file to this cell, then it searches for the cell_name.$datasheet_html_schematic_suffix under the $datasheet_html_schematic_dir directory as the symbol file for this cell.

If it cannot find the schematic file by using one of these methods, the datasheet generator replaces `{schematic_file}` with `None`.

The format of the schematic file should be JPEG, BMP, or PNG.

The datasheet generator does not display a schematic picture unless you specify that it do so.

Syntax

```
schematic: schematic_file
```

Example

```
set datasheet_html_schematic_suffix schematic_URL
```

This option is available only for HTML datasheets.

Port Names (HTML and Text)

Syntax

```
Inputs: pin_list
Outputs : pin_list
InOuts: pin_list
```

pin_list

The pin list is a comma-separated list. It takes its name from the current library pin name in the input Liberty file.

The rules regarding pin placement are determined by the `direction` attribute in the current pin group, as follows:

- If the value is `input`, the pin is written to the Inputs list.
- If the value is `output`, the pin is written to the Outputs list.
- If the value is `inout`, the pin is written to the Inouts list.

This option is available for both HTML and text datasheets.

Cell Area (HTML and Text)

Syntax

```
Cell area : cell_area
```

`cell_area`

The cell area is a floating-point number with six digits after the decimal point. It is retrieved from the `area` attribute at the cell-level from the input Liberty file. No units are explicitly given for the value, but the datasheet generator uses the same unit for the area of all cells in a library.

.lib Example

```
cell (ACHCINX2) {
area: 14.818000;
}
```

Datasheet Example

```
Cell area: 14.818000
```

This option is available for both HTML and text datasheets.

Pin Capacitance (HTML and Text)

The datasheet generator returns pin capacitance data in two tables, one for input pins and the other for output pins.

The input pin table includes input and bidirectional pins. The input pin capacitance values are derived from the `capacitance` attribute in the .lib file. If the .lib file does not include a `capacitance` attribute, the value is derived from the larger of the `fall_capacitance` and `rise_capacitance` attribute values. If there are no `rise_capacitance` and `fall_capacitance` attributes in the .lib file, no capacitance value is written for that pin.

The output pin table includes output and bidirectional pins. The output pin capacitance values are derived from the `max_capacitance` attribute in the .lib file.

Syntax

Inputs:

Pin	**Cap. [*capacitance_unit]***
pin_name	*capacitance_value*
pin_name	*capacitance _value*

Outputs:

Pin	**Max. Cap. [*capacitance_unit]***
pin_name	*maximum capacitance_value*
pin_name	*maximum capacitance_value*

pin_name

The pin name in the input Liberty file.

capacitance_value

A floating-point number with six digits after the decimal point.

capacitance_unit

The capacitance unit for the capacitance value and the maximum capacitance value, which is derived from the `capacitive_load_unit` attribute in the input Liberty file.

.lib Example (For an Input File)

```
capacitive_load_unit(1.000000, "pf");
cell (example) {
      pin (A) {
            capacitance: 2.258850;
}
pin (B) {
      capacitance: 5.234676;
}
pin (CIN) {
      rise_capacitance: 2.225339;
      fall_capacitance: 2.314452;
}
pin (CO) {
      max_capacitance: 109.509995;
}
}
```

Datasheet Example

```
Pin Capacitance

Inputs:
```

Pin	**Cap. (pF)**
A	2.258850
B	5.234676
CIN	2.314452

```
Outputs:
```

Pin	**Max. Cap. (pF)**
CO	109.509995

This option is available for both HTML and text datasheets.

Delay Data (HTML and Text)

Syntax

Delay Path	**Delay [*time_unit*]**
input_pin input_transition => *output_pin output_transition*	*delay_value*

The arguments are as follows:

`input_pin`

The input pin, or related pin, for a timing arc.

`output_pin`

The output pin for a timing arc.

`input_transition`

Shows the state transition of the input.

`output_transition`

Shows the state transition of the output.

time_unit

This value is derived from the time_unit attribute in the input Liberty file.

delay_value

A floating-point number with six digits after the decimal points.

This option is available for both HTML and text datasheets.

Input Transition

The input transition characteristics are as follows:

- For non_unate delay arcs, no input transition is written out.
- For positive_unate and negative_unate delays arcs, the input transition is 01 and 10.

Input transition is derived from the output transition, timing_type, and timing_sense. When timing_type is combinational, combinational_fall, combinational_rise, preset, or clear, there can be up to two input transitions, as follows:

- When timing_sense is positive_unate, the input transition is consistent with the output transition.
- When timing_sense is negative_unate, the input transition is opposite to the output transition.

When timing_type is three_state_disable, three_state_enable, three_state_disable_rise, three_state_disable_fall, three_state_enable_rise, or three_state_enable_fall, the input transition is as follows:

- When timing_sense is positive_unate, the input transition is 01.
- When timing_sense is negative_unate, the input transition is 10.

When timing_type is rising_edge, the input transition is 01, and when timing_type is falling_edge, the input transition is 10.

Output Transition

The output transition characteristics are as follows:

- The valid values are: 01, 10, 0Z, Z1, 1Z or Z0

  where 0 is low state, 1 is high state, and Z is high-impedance state. For arcs that are not three-state arcs, if the group type is cell_rise, the *output_transition* is 01, and if the group type is cell_fall, the *output_transition* is 10.

  For three-state arcs, the output transition is derived from timing_type, as follows:

  - When timing_type is three_state_disable, the output transition is 0Z/1Z.

- When `timing_type` is `three_state_enable`, the output transition is `Z0/Z1`.
- When `timing_type` is `three_state_enable_rise`, the output transition is `Z1`.
- When `timing_type` is `three_state_enable_fall`, the output transition is `Z0`.
- When `timing_type` is `three_state_disable_rise`, the output transition is `0Z`.
- When `timing_type` is `three_state_disable_fall`, the output transition is `1Z`.

The datasheet generator chooses the last value in the `cell_rise` or `cell_fall` table. This corresponds to the worst delay, as the delay value is monotonically increasing in the `cell_rise` or `cell_fall` table.

If there are duplicate timing arcs and if timing groups in the input Liberty file have multiple path delays with the same transition for the same input to output pin, they are considered duplicate path delays. The datasheet generator avoids generating duplicate path delays. If the datasheet generator finds a potential duplicate path delay, it uses the timing arc with the worst delay value.

Example

The following example shows a delay table for a 2-input NAND cell:

Delay Path	**Delays [ps]**
A 10 => Y 10	123.432424
A 01 => Y 01	221.425345
B 10 => Y 10	121.735747
B 01 => Y 01	121.758758

This option is available for both HTML and text datasheets.

Constraint Data (HTML and Text)

The datasheet contains the following information about constraint data.

Syntax

Check	**Constraint [*time unit*]**
constraint_pin constraint_pin_transition check_type related_pin related_pin_transition	*constraint_value*

The arguments are as follows:

constraint_pin

The parent pin of a timing constraint in an input Liberty file.

constraint_pin_transition

The value is derived from the `rise_constraint` or `fall_constraint` group type in the input Liberty file.

check_type

The timing check type. Currently, only setup, hold, recovery, removal, and skew are supported.

related_pin

The pin with the `related_pin` attribute in the timing constraint.

related_pin_transition

The value is derived from the `timing_type` attribute in the input Liberty file. If `timing_type` is `setup_rising`, `hold_rising`, `recovery_rising`, `removal_rising`, or `skew_rising`, the related pin transition is `01`.

time_unit

The value is derived from the `time_unit` attribute in the input Liberty file.

constraint_value

The value is a floating-point number with six digits after the decimal point.

The related pin transition characteristics are as follows:

- If `timing_type` is `setup_rising`, `hold_rising`, `recovery_rising`, `removal_rising`, or `skew_rising`, the related pin transition is `01`.
- If `timing_type` is `setup_falling`, `hold_falling`, `recovery_falling`, `removal_falling`, or `skew_falling`, the related pin transition is `10`.

The constraint pin transition characteristics are as follows:

- The constraint pin transition is `rise` when the group type is `rise_constraint`.
- The constraint pin transition is `fall` when the group type is `fall_constraint`.

The datasheet generator chooses the last value in the `rise_constraint` or `fall_constraint` table for the constraint value.

If timing groups in Liberty NCX have multiple timing checks with the same transition for the same input and output and the same type, they are considered duplicate timing checks. The duplicate timing checks are as follows:

- The datasheet generator avoids generating duplicate timing checks.
- If a potential duplicate timing check is found, the datasheet generator uses the timing constraint with the worst constraint value and ignores the rest.

Example

The following example shows the constraint data:

Check	**Constraints [ps]**
D 10 setup CK 01	123.432523
D 10 hold CK 01	221.432523
D 01 setup CK 01	121.432523
D 01 hold CK 01	121.432523

Power Data (HTML Only)

Input Power Data

Input power data shows the internal power corresponding to each input pin. This information is derived from the .lib file.

Input Syntax

Power Path	**Power [*power_unit]***
input_power_path	*input_power*

Output Power Data

Output power data provides information about the internal power at the output pin corresponding to a path from each input pin in the cell. This information is derived from the .lib file.

Output Syntax

Power Path [*power_unit*]	Load Capacitance [*cap_unit]*					
	load	load	load	load	load	load
output_power_path	*{output_ power}*	*{output_ power}*	*{output_ power}*	*{output_ power}*	*{output_ power}*	*{output_ power}*

Input and Output Syntax Arguments

input_power_path

Provides information on the input pin and whether it is rise or fall `internal_power`. For example, `I1 01` Indicates a `rise internal_power` for input pin `I1`. The *input_power_path* is a floating-point number with six digits after the decimal point.

input_power

Is the last value in the `internal_power` table corresponding to this path and type. The *input_power* is a floating-point number with six digits after the decimal point.

power_unit

A floating-point number with six digits after the decimal point.

cap_unit

A floating-point number with six digits after the decimal point.

load

A floating-point number with six digits after the decimal point.

output_power_path

Shows which path goes from the input pin to the output pin and whether it is rise or fall. The `load` is the list of output load indexes and is derived from the .lib file. The *output_power_path* is a floating-point number with six digits after the decimal points. The delay value is extracted from NLDM tables only in Liberty NCX version A-2007.12.

output_power

Is the last row of the table of the internal power values corresponding to the `output_power_path`. The *output_power* is a floating-point number with six digits after the decimal points. The delay value is extracted from NLDM tables only in Liberty NCX version A-2007.12.

This option is available only for HTML datasheets.

Generating Verilog Models

In addition to Liberty models, characterization tools support Verilog models, which provide detailed information about a cell. Generating Verilog models from Liberty .lib files is helpful if you are new to characterization or if you are creating new libraries rather than recharacterizing existing libraries with new process models. This section describes how to generate Verilog models in .v file format from Liberty .lib files for use in characterization. It includes the following sections:

- Overview
- Generating Verilog Files
- Creating UDPs
- Verilog Model Details

Overview

The characterization flow requires that you first run Liberty NCX and generate a .lib file. Next, you generate a Verilog file from the .lib file using Library Compiler. This ensures that the Verilog file matches the .lib file exactly. Although you generate the Verilog file in Library Compiler, it requires a Liberty NCX license and will fail if a Liberty NCX license is not available.

Generating Verilog Files

Library Compiler generates a Verilog model from a .lib file and creates a *cell*.v Verilog file for each cell in the library. By default, Library Compiler creates the cell.v files in the user-defined *library_name*_verilog directory, under the current working directory. If the *library_name*_verilog directory already exists under the current working directory, the newly generated *cell*.v files overwrite the existing files in the directory, and Library Compiler issues a warning message.

Use the following commands to generate Verilog models:

1. To enable Verilog model generation, set the `veriloglib_enable` variable to `true`. By default, the variable is set to `false`, and a Verilog file is not generated.

   ```
   set veriloglib_enable [true | false]
   ```

2. To create a Verilog file from a .lib file, run the `write_lib` command in lc_shell and specify the .lib library name as shown:

   ```
   write_lib -format verilog library_name
   ```

3. If you want the Verilog model to be written to a directory other than library_name_verilog, set the `veriloglib_output_dir` variable, as shown:

   ```
   set veriloglib_output_dir directory_name
   ```

Creating UDPs

The Verilog model generator can create two types of User-Defined Primitives (UDPs): independent UDPs and custom UDPs. They are described in the following subsections:

- Creating Independent UDPs
- Creating Custom UDPs

Creating Independent UDPs

The Verilog model generator generates a Verilog model with independent user-defined primitives (UDPs) and allows multiple cells to share the same user-defined primitive.

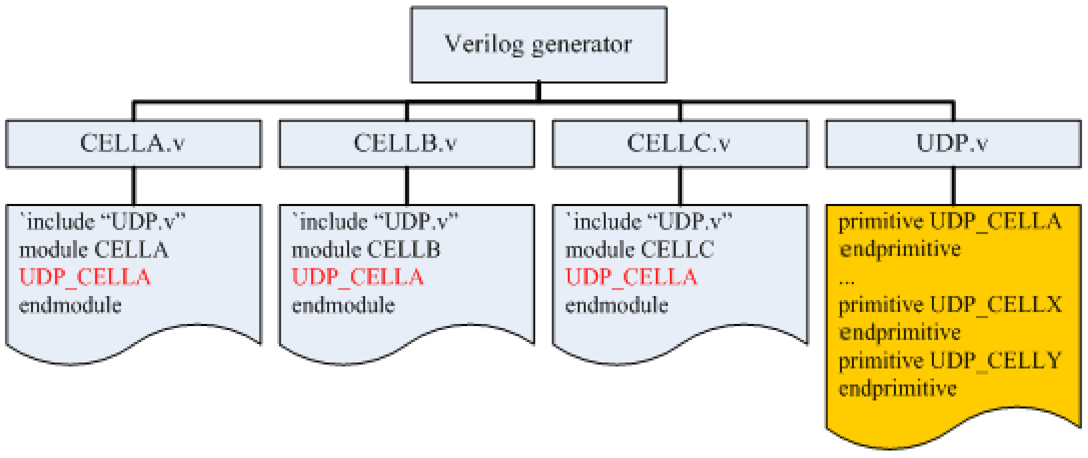
The Verilog model generator generates user-defined primitives and modules in separate Verilog files. The Verilog model generator determines whether the cell function model is the same as the one that was previously output; if the cell function model is the same, the Verilog model generator includes that user-defined primitive file and does not generate a new one. This is called the independent user-defined primitive feature of the Verilog model generator.

To use the independent user-defined primitive flow in the Verilog model generator, set the `veriloglib_independent_udp` variable to `true`. The default is `false`.

In an independent UDP flow, the Verilog file contains only the module part for each cell. The UDP parts are written in the UDP.v file, which is shared by all cells. The Verilog model generator identifies whether or not the cell's function is the same as the one that has already been generated. It uses the existing cell's UDP in the UDP.v file if they are identical; otherwise, it generates a new UDP for this cell and writes it into the UDP.v file.

This flow is shown by the example in Figure B-1 on page B-35. Assume that the output cell order is CELLA, CELLB, CELLC, and they have exactly the same function model. The Verilog model generator determines that the function of CELLB and CELLC are the same as that of CELLA, and CELLA's UDP has already been generated in the UDP.v file. CELLB and CELLC simply call CELLA's UDP, so no duplicate UDP has to be generated.

Figure B-1 Independent Verilog UDP Flow



Flow for Independent UDP

The independent UDP flow includes the following usage characteristics:

Set the `veriloglib_independent_udp` variable to `true` to use the independent UDP flow in the Verilog model generator.

Each independent UDP's naming rule is the same as the previously generated Verilog model. The generated independent UDP file is named UDP.v.

The generated independent UDP file (UDP.v) is located in the same directory as other generated Verilog files. If the `veriloglib_output_dir` directory is defined, UDP.v is output to it; otherwise, it is output to the default directory *library_name*_verilog.

Creating Custom UDPs

Although the Verilog model generator can generate user-defined primitives (UDPs) automatically, you might prefer using your own custom UDPs. You can generate Verilog models using your own UDPs by providing a complete Verilog function model and a complete Verilog UDP that matches the cell in the Liberty file. Figure B-2 on page B-36 shows the Verilog model generation flow without custom UDPs. The Verilog model is directly generated from the Liberty NCX file.

Figure B-2 Verilog Model Generation Flow Without Custom UDPs

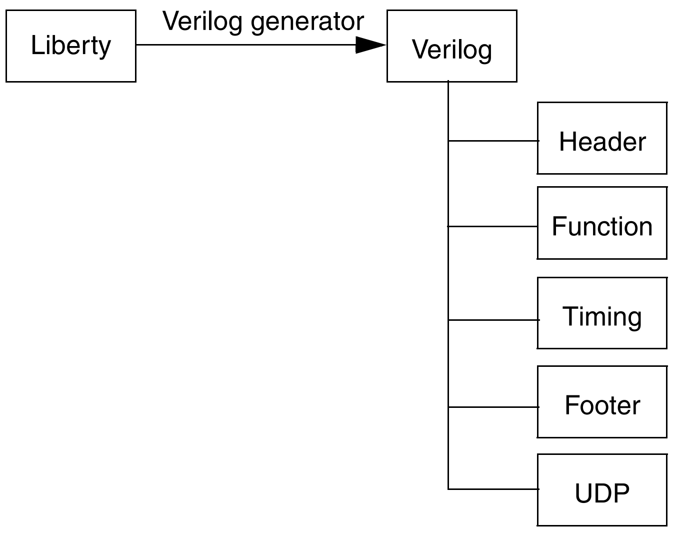
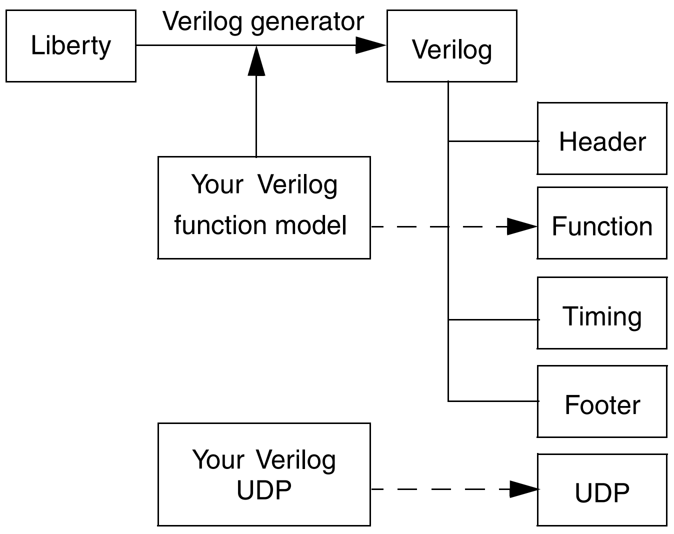


Figure B-3 shows the Verilog model generation flow with custom UDPs.

Figure B-3 Verilog Model Generation Flow With Custom UDPs



In the custom UDP flow, you must define the following:

- Verilog UDP

  Normally, the UDP definition is generated automatically in the Verilog model. In the custom UDP flow, you add the UDP definition.

- Verilog function model

  The function model is the Verilog code that instantiates the custom UDP and ties it to the logic surrounding it. You must specify this in a function file that is located either in the *library_name*_func directory or in the directory that you specify with the `veriloglib_custom_func_dir` attribute. The Verilog syntax is added to the “Function” section of the generated Verilog.

Enabling the Custom UDP Flow

You can enable the custom UDP flow in one of the following ways:

1. Create a *library_name*_func directory in the same directory as the Liberty file.
2. Define the `veriloglib_custom_func_dir` variable to point to the directory that contains the mapping files, as shown:

   ```
   set veriloglib_custom_func_dir custom_func_dir
   ```

In either case, the Verilog model generator looks for the mapping files in the directory, maps them to the cells in the Liberty file, and uses the custom function file when generating the Verilog model for the mapped cells. The Verilog model generator continues to generate Verilog models as usual for cells that are not mapped.

Defining the Function Section

You can define the function file with a specification of the cell.func or library.fun file, as described in the following two ways:

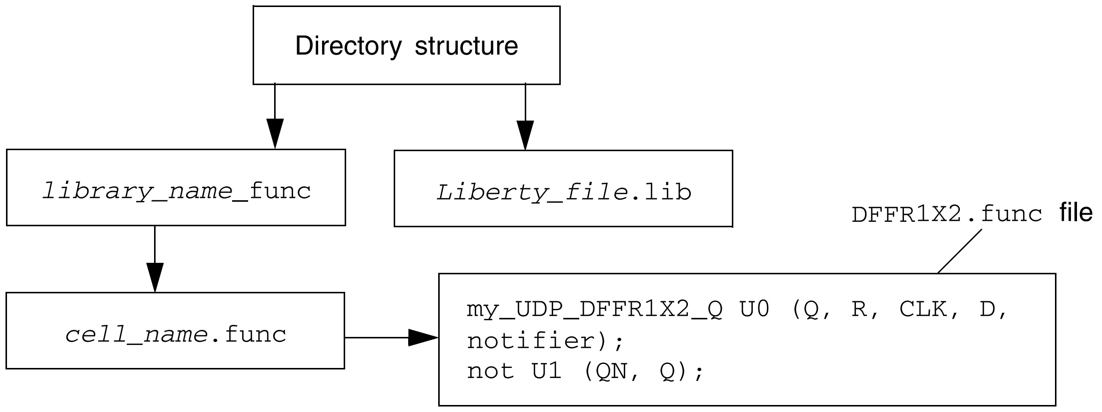
1. Figure B-4 on page B-38 shows the specification of the cell.func file. Specify the following for the *cell_name*`.func` file:

   a. Location: The *cell_name*`.func` file should be in the *library_name*_func directory or in the directory specified by the `veriloglib_custom_func_dir` attribute.

   b. Naming convention: There should be a *cell_name*`.func` file for each cell that requires a custom UDP. The Verilog model generator looks for the .func file extension and maps each of the *cell_name* prefixes to a cell in the library. If a cell matching this prefix is found, the contents of the *cell_name*`.func` file are used in the Verilog model for the cell.

   c. Content: The function file contains the Verilog instantiation of the custom UDP. The Verilog model generator does not check the syntax or the logical connection of the Verilog specified in the function file. You must make sure that the Verilog description matches the Liberty NCX description.

Figure B-4 Specifying the cell_name.func File



```
my_UDP_DFFR1X2_Q U0 (Q, R, CLK, D,
notifier);
not U1 (QN, Q);
```

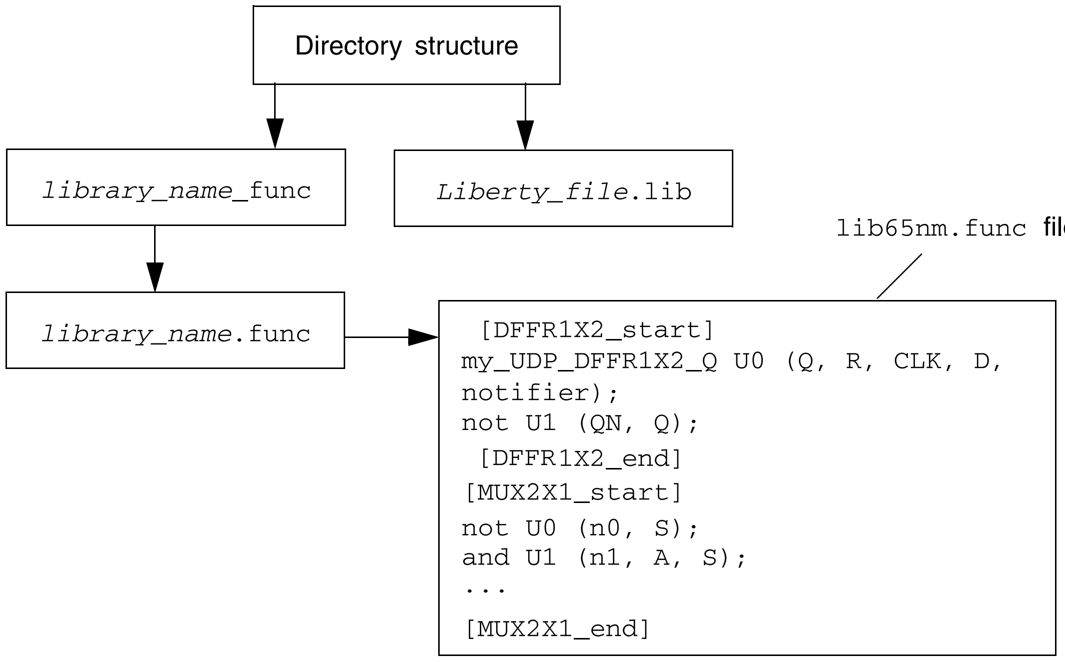
2. Figure B-5 on page B-39 shows the specification of the library.func file. Specify the following for the `library_name.func` file:

   a. Location: The `library_name.func` file should be in the *library_name*_func directory or in the directory specified by the `veriloglib_custom_func_dir` attribute.

   b. Naming convention: The name of this file should be *library_name*.func. The Verilog model generator looks for the .func file extension and maps the *library_name* prefix to the name of the library in the Liberty file. If there is a match, the custom UDP is implemented for all cells defined in the *library_name*.func file.

   c. Content: Reading each *cell_name*.func file can be a burden on the tool; therefore, the Verilog model generator supports the combining of all the custom function files into one file, reducing the number of file reading operations. The name of the file must be *library_name*.func. It includes the function section for each of the cells. The function sections are included in cell separators to uniquely identify them for each cell. For example, `[cell_name_start]:` identifies the start of the cell's function section and `[cell_name_end]:` identifies the end of the cell's function section. The brackets ([ ]) surrounding the `cell_name_start` and `cell_name_end` are required. The information between these separators is similar to what is defined in each *cell_name*.func file.

Figure B-5 Specifying the library.func File



```
 [DFFR1X2_start]
my_UDP_DFFR1X2_Q U0 (Q, R, CLK, D,
notifier);
not U1 (QN, Q);
 [DFFR1X2_end]
[MUX2X1_start]
not U0 (n0, S);
and U1 (n1, A, S);
...
[MUX2X1_end]
```

3. Figure B-6 on page B-40 shows the specification of the cell.func file. You can use the mapping file to specify the custom function section in the Verilog file. In the general flow, each cell defines its own function section. Specifying the mapping file is helpful if many cells have the same function section or when you are defining the function section for cells that have the same function but different drive strengths. Specify the mapping file:

   a. Location: The func.map mapping file should be in the *library_name*_func directory or in the directory specified by the `veriloglib_custom_func_dir` attribute.

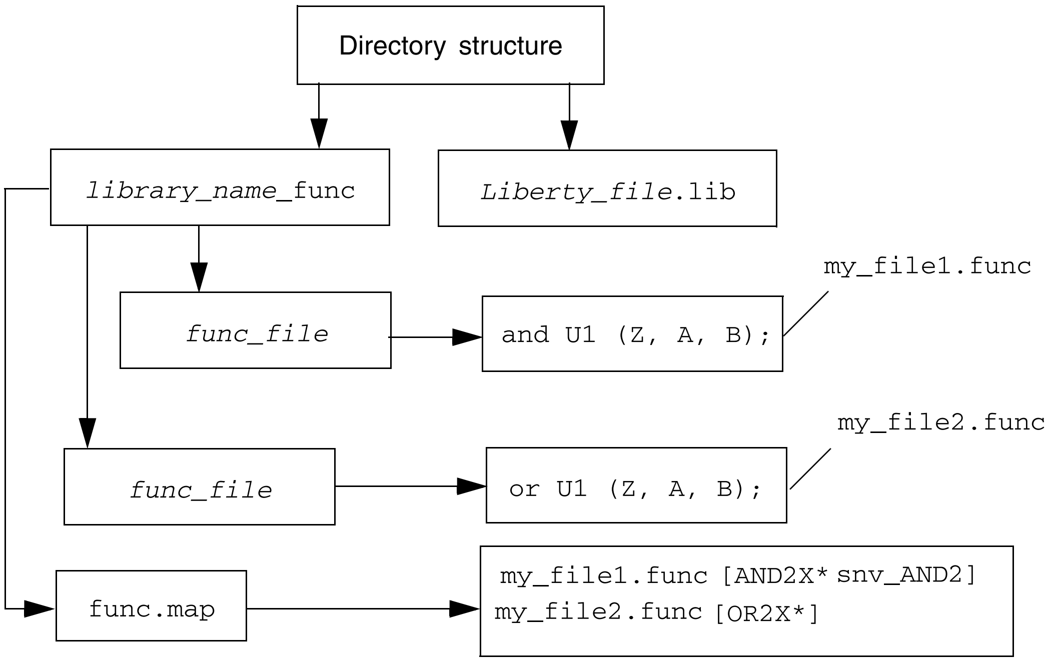
   b. Naming convention: The name of the mapping file must be func.map; however, the files containing the function section can have any name. The mapping file shows how each of the function files maps to the cells.

   c. Content: The mapping file contains multiple entries in the following format:

   ```
   func_file_name [cell_name|wildcard name_of_cell ...]
   ```

   Where the func_file_name is the file that contains the function section for all the cells in the cell list that are enclosed in brackets ([ ]). The brackets are required. The cell list can include complete cell names or cell names with wildcard characters. Currently, the asterisk (*) character in a suffix is the only wildcard that is supported. You must separate multiple cell names with spaces.

Figure B-6 Specifying the func.map Mapping File



Search Order for the Custom UDP

There are multiple ways to specify the function section of the custom UDP. The Verilog model generator searches in the following order when looking for custom UDPs for a cell to determine which function section to use:

1. The Verilog model generator looks for custom UDPs in the `func.map` file first. The `func.map` file can be located in the *library_name*_func directory or in the directory that you specify with the `veriloglib_custom_func_dir` attribute. If it finds the file, the Verilog model generator reads the contents of the file and maps the function section to the cells in the corresponding cell list. After it finds a function section for a cell, it ignores that cell.

2. The second place the Verilog model generator looks for UDP information is the *`cell_name`*`.func` file in the `library_name_func` directory or in the directory that you specify with the `veriloglib_custom_func_dir` attribute. The Verilog model generator maps the function section in this file to the cell. If the cell already has a function section defined, for example from a `func.map` definition, the function section in the *`cell_name`*`.func` file is ignored.

3. Lastly, the Verilog model generator looks for UDP information in the `library_name.func` file in the `library_name_func` directory or in the directory that you specify with the `veriloglib_custom_func_dir` attribute. Then, the Verilog model generator maps the

function section to the cells that were not mapped in steps 1 and 2. Because this is the last priority in the Verilog model generator's search for UDPs, only the cells that are not previously mapped are mapped at this point.

Guidelines for Specifying the UDP

In the custom UDP flow, the UDP part of the Verilog model is not generated by the Verilog model generator so that the custom UDP can be used. These are the rules for UDP customization:

- Multiple UDPs can be defined for one cell in the custom UDP file.
- The functionality and syntax are not verified by the Verilog model generator, so make sure the UDP function in the function section matches the Liberty definition.

Verilog Model Generator Assumptions

The Verilog model generator assumes the following:

- The Verilog specified in the function section is syntactically correct. No check is done to make sure there are no syntax errors.
- The Verilog model generator does not check that the custom UDP is correctly instantiated, including the names of pins and nets. It does not check that there are no conflicts between the nets or pins in the generated Verilog and the corresponding function section.

  For example, if the function section is

  ```
  and U0 (Z, A, B);
  ```

  and the Verilog model generated from the .lib file is

  ```
  module AND2X2 (I1, I2, O);
  ...
    // inserted by custom UDP flow
    and U0 (Z, A, B);
  ...
  endmodule
  ```

  there is a pin mismatch between what is instantiated [Z, A, B] and what is defined in the module [O, I1, I2]. You must make sure that the custom UDP is instantiated correctly. The Verilog model generator does not check this.
- It is important that you make sure the custom UDP is logically correct and matches the Liberty file used to generate the Verilog.

Verilog Model Details

The outline of a cell's Verilog model is as follows:

```
`timescale 1ns / 1ps
`celldefine
module cell_name ( cell pin list )      // header section
pin declarations
gate_instantiation &| UDP_instantiation  // function description
specify                               // specify block
specparam definition
path delay
timing checks
endspecify
endmodule
`endcelldefine
[primitive UDP_identifier ( UDP_pin_list );// UDP definition for
                                              sequential cells
UDP_pin_declaration
UDP_body
endprimitive]
```

Each Verilog model contains the following:

- The `` `celldefine `` and `` `endcelldefine `` compiler attributes and the `` `timescale `` compiler attribute. The Verilog model generator only supports these types of compiler attributes.
- The `module` declaration with all pins in the cell declared in the pin declaration section.
- The cell function, either using instances of Verilog built-in gates or instances of UDPs.
- The `specify` block, where path delay and optional timing checks reside. If the input Liberty file contains timing constraint information, corresponding timing checks appear in the `specify` block.
- The UDP definition if it is instantiated in the cell function.

Header Section

The following sections are included in the header section:

Module Declaration

The module syntax is as follows:

Syntax

```
module cell_name ( cell_pin_list );
```

The arguments are as follows:

- *`cell_name`* is the name of the cell as specified in the input Liberty file.
- *`cell_pin_list`* is a list of the pins in the cell. The order of the list follows the pin group orders specified in the input Liberty file.

Example

```
module AND2X2 (A, B, O);
```

Pin Declarations

Pin declarations specify an ordered list of the pins for the module. The order follows the pin order in the input Liberty file. Each pin declaration can be one of the following:

```
inout [range] pin_identifier;
input [range] pin_identifier;
output [range] pin_identifier;
[reg notifier;]
```

The arguments are as follows:

- *`range`* gives addresses to the individual bits in a multiple-bit bus, depending on the following pin types:
  - Scalar pins: The *`range`* argument is not used by scalar pins.
  - Liberty bundle: This Liberty construct is ignored and *`range`* is not used.
  - Liberty bus pins: The *`range`* value is derived from the bus associated type specification in the input Liberty file.
- *`pin_identifier`* corresponds to the pin name in the Liberty file and depends on the following pin types:
  - Scalar pins: The *`pin_identifier`* argument is the corresponding pin name in the input Liberty file.
  - Liberty bundle: The *`pin_identifier`* argument is each member pin of the bundle group. The bundle argument is ignored.
  - Liberty bus pins: The *`pin_identifier`* argument is the bus name found in the input Liberty file.

    Note:
    No bus member is referred here, so no `bus_naming_style` consideration is needed.
- *`reg_notifier`* is declared as a register in the module where timing check tasks are invoked, and the register notifier is passed as the last argument to a system timing check.

Example

```
input A;
input B;
output O;
```

Function Description

If the function description of the cell is included in the input Liberty file, it is written out in the Verilog model. For black box cells that have no function description in the .lib file, the Verilog model generator issues a warning and adds a comment in the Verilog function section.

Combinational Function

The combinational function section represents the behavior of the cell using gate instantiations. The Verilog model generator models all combinational cells, including adders, subtracters, multiplexers, and decoder cells with gate instantiation, rather than UDPs.

Syntax

The gate instantiation syntax is as follows:

```
gate_primitive  instance_name (terminal_list);
```

The arguments are as follows:

- *gate_primitive*

  The *gate_primitive* argument is the built-in primitive name in Verilog. For each output pin and bidirectional pin, the Verilog model generator converts the Boolean expression in the `function` attribute directly to Verilog built-in primitives.

  Valid primitive names include, `and`, `or`, `xor`, `not`, `nand`, `nor`, `xnor`, and `buf`.

- *instance_name*

  The *instance_name* argument is the name of the instance of the gate primitive. The format of the instance name is `U<n>`, where *n* starts at 0 and increases in steps of 1 for every instance (gate or UDP) that is added.

- *terminal_list*

  The *terminal_list* argument describes how the gate connects to the rest of the model. The characteristics are as follows:

  - The terminals are separated by commas.
  - The order of the terminal list is determined by the gate primitive definition. Output pins and bidirectional pins are generally specified first, followed by input pins.
  - The terminal list can be either pins of the cell or internal connection signals constructed as output from other instances.

- The terminal list determines the internal net-naming convention. To avoid possible conflict with existing pin names in a cell, the internal net names are constructed in the format `_net_<n>`, where *n* is an integer starting at 0 and increasing in steps of 1 for every net added. If the format conflicts with the cell's pin names, the Verilog model generator quits with an error. The name does not need to be explicitly declared for a single-bit signal scalar.

The following examples show the .lib syntax followed by the Verilog equivalent:

.lib Example

```
pin (Y) {
function: A ^ B' & C;
}
```

Verilog Example

```
not U0(_net_0, B);
xor U1(_net_1, _net_0, A);
and U2(Y, _net_1, C);
```

Sequential Function

Sequential cells are modeled using UDP instances. Each UDP instance models a sequential state. Cells with multiple states are modeled as a netlist of multiple UDP instances. Output logic is modeled using gate instantiation.

Syntax

The UDP instantiation syntax is as follows:

```
udp_name instance_name (terminal_list);
```

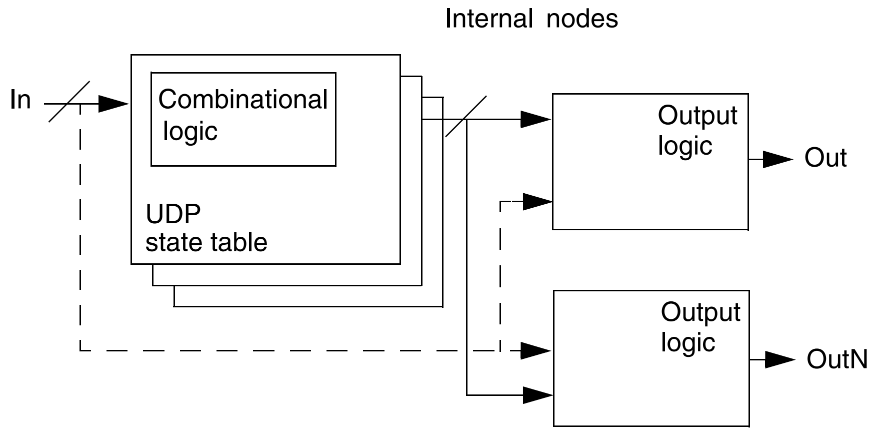
where *udp_name* is formatted as `UDP_cell_port`, and where *cell* specifies the cell name and *port* specifies either the name of the output pin or the internal node.

The *terminal_list* argument describes how the UDP is connected to the model. The characteristics are as follows:

- The order of the terminal list is determined by the UDP definition. Output pins and bidirectional pins are generally specified first, followed by input pins.
- If a cell has bus or bundle pins, each individual member is modeled independently with its own UDP instantiation of the same UDP primitive.
- Each state of a cell is modeled with a UDP instantiation.

Figure B-7 on page B-46 shows the general sequential output model with UDP state tables. Sequential cell functions might contain several sequential "states" and thus can be described with multiple UDP state tables. Each UDP instantiation (state table) describes one sequential state. Cells with multiple sequential states have multiple UDP instantiations.

Figure B-7 General Sequential Output Model



Input (combinational) logic is included in the UDP state table description. Thus, there is no need to explicitly describe the input logic in the Verilog function description. Output logic is not included in the UDP state table, which requires gate instantiations to be described in Verilog. Output logic is modeled using gate instantiation, similar to combinational functions.

Three-State Function

A three-state function is similar to the combinational function, except that it specifically uses either the `bufif0` or `bufif1` gate primitives. When the `three_state` attribute is specified on an output pin in the input Liberty file, one of the `bufif0` or `bufif1` gate primitives is used to define the output value for the three-state disable state. That whether `bufif0` or `bufif1` is used depends on the active high or low state of the *`three_state_signal`* argument.

Syntax

The gate instantiation syntax is as follows:

```
bufif0 instance_name(output_pin, normal_function_signal,
three_state_signal);
bufif1 instance_name(output_pin, normal_function_signal,
three_state_signal);
```

The arguments are as follows:

- *`output_pin`*

  Specifies the output pin name in the .lib file.

- *`normal_function_signal`*

  Specifies the internal connection signal that describes logic in the `function`, `internal_node` or `state_function` attributes in the output pin group.

- *three_state_signal*

  Specifies the internal connection signal that describes logic in the `three_state` attribute in the output pin group.

The following examples show the .lib syntax followed by the Verilog equivalent:

.lib Example

```
pin (Y) {
function: A&B;
three_state: EN';
}
```

Verilog Example

```
and U0(_net_0, A, B);   //construct function logic;
bufif1 U2(Y, _net_0, EN);  //express output value for three state;
```

specparam

The `specparam` syntax is as follows:

```
specparam
delay_param=0.01 [, delay_param=0.01,…];
[constraint_param=0.01 [, constraint_param=0.01,…]];
```

The arguments are as follows:

- *delay_param*

  Specifies the parameter name for the path delay.
- *constraint_param*

  Specifies the parameter name for the timing checks.
- The unit time (0.01) is used for the path delay and the timing check parameters.

The naming convention for the *delay_param* path is as follows:

`tdelay_`*`input_output_transition`*`[_`*`index`*`]`

where

- *input* is the input pin (`related_pin`) for a timing arc.
- *output* is the output pin for a timing arc.
- *transition* is one of the following: `01`, `10`, `0z`, `z1`, `1z`, `z0`.
- *index* is an integer value starting at 0 and increasing in steps of 1 when multiple delay parameters with the same input, output, and transition values are specified.

The naming convention for the *constraint_param* timing check parameter is as follows:

```
t<check_type>_<start_pin>_[<end_pin>_]<index>
```

where

- The `check_type` values include `setup`, `hold`, `recovery`, `removal`, `pulsewidth`, `period`, and `skew`.
- `start_pin` is the pin associated with the start event in the related timing check. The start event is determined by the specific timing check type. For more information, see "Timing Checks" on page B-52.
- `end_pin` is the pin associated with the end event in the related timing check. The end event is determined by a specific timing check type. For more information, see "Timing Checks" on page B-52.

  Because `start_pin` and `end_pin` are the same for `pulsewidth` and `period`, only `start_pin` is used in the parameter name.
- `index` is an integer value starting at 0 and increasing in steps of 1 when multiple timing check parameters with the same timing check type, start pin, and end pin values are specified.

Delay Example

```
specify
specparam
tdelay_I1_O_01_0=0.01,
tdelay_I1_O_10_0=0.01;
(I1 +=> O)=(tdelay_I1_O_01_0, tdelay_I1_O_10_0);
endspecify
```

Timing Check Example

```
specify
specparam
tsetup_D_CK_0=0.01,
thold_CK_D_0=0.01;
$setuphold(posedge CK , posedge D , tsetup_D_CK_0 ,thold_CK_D_0,
notifier);
endspecify
```

Path Delay

These path delays are described in the following subsections:

- "Simple Path Delay" on page B-48
- "Edge-Sensitive Path Delay" on page B-49

Simple Path Delay

The simple path delay syntax is as follows:

```
(input [polarity] => output) = (delay_param, delay_param);
```

```
(input [polarity] => output) = (delay_param, delay_param, delay_param
 delay_param, delay_param, delay_param);
```

The arguments are as follows:

- *input*

  Specifies the input pin name.

- *polarity* (optional)

  Specifies polarity. Specify "+" for positive unate arcs, "-" for negative unate arcs, or no polarity for `non_unate` arcs or arcs without the `timing_sense` attribute, such as three state.

- *output*

  Specifies the output pin name.

- *delay_param*

  Specifies the delay parameter name. The number of *delay_param* is either 2 for non-three-state rise and fall delays or 6 (rise, fall, three state) for three-state arc delays. See "Delay Values" on page B-52.

  The naming convention for the *delay_param* path is as follows:

  `tdelay_`*`input_output_transition`*`[_`*`index`*`]`

  where

  - *input* is the input pin (`related_pin`) for a timing arc.
  - *output* is the output pin for a timing arc.
  - *transition* is one of the following: `01`, `10`, `0z`, `z1`, `1z`, `z0`.
  - *index* is an integer value starting at 0 and increasing in steps of 1 when multiple delay parameters with the same input, output, and transition values are specified.

  The *delay_param* includes timing groups with one of the following values for `timing_type` and no `when` condition, except `default_timing_group`: `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`, `preset`, and `clear`.

  The default timing group might have no `when` condition but it is modeled using the `ifnone` simple path delay. See "State-Dependent Path Delay" on page B-51.

Edge-Sensitive Path Delay

The edge-sensitive path delay syntax is as follows:

```
(edge_identifier input => (output [polarity] : data_source)) =
```

`(`*`delay_param`*`,` *`delay_param`*`);`

The arguments are as follows:

- *`edge_identifier`*

  Specifies `posedge` or `negedge` based on the `timing_type` and `timing_sense` attributes of the timing group in the input Liberty file.

- *`data_source`*

  Describes the flow of data to the path destination. For example, for the DFF cell clock to Q output timing path, the data pin state is actually populated to Q; thus the data pin is *`data_source`*.

  Use the `veriloglib_sdf_edge` variable to control globally whether a timing group is modeled with edge-sensitive path delay, as shown:

  ```
  set veriloglib_sdf_edge [true | false]
  ```

  If set to false, no edge-sensitive path delay is modeled.

  If the `veriloglib_sdf_edge` variable is set to `true` (the default),

  - If the timing group has a `timing_type` attribute with a value of `rising_edge` or `falling_edge`, it is modeled with an edge-sensitive path delay. The `rising_edge` value denotes `posedge` on the input. The `falling_edge` value denotes `negedge` on the input.

  - If the `timing_type` value in the timing group is `preset`, the input edge is `posedge` if `timing_sense` is `positive_unate`. The input edge is `negedge` if `timing_sense` is `negative_unate`. Otherwise, the path delay is modeled with a simple path delay rather than an edge-sensitive path delay.

  - If the `timing_type` value in the timing group is `clear`, the input edge is `negedge` if `timing_sense` is `positive_unate`. The input edge is `posedge` if `timing_sense` is `negative_unate`. Otherwise, the path delay is modeled with a simple path delay rather than an edge-sensitive path delay.

  For other cases, the path delay is modeled with a simple path delay.

.lib Example

```
pin (Q) {
      direction : "output";
      function : "IQ";
      timing () {
            related_pin : "CK";
            timing_type : "falling_edge";
      }
}
```

The `falling_edge` value in the .lib example corresponds to `negedge` in the Verilog example.

Verilog Example

```
(negedge CK => Q)=(tdelay_CK_Q_01_0, tdelay_CK_Q_10_0);
```

State-Dependent Path Delay

The state-dependent path delay syntax is as follows:

```
if (sdf_cond)  simple path delay
| if (sdf_cond) edge sensitive path delay
| ifnone simple path delay
```

where

sdf_cond specifies the sdf_cond attribute in the input Liberty timing group.

If the sdf_cond value in the Liberty file is “one bit signal,” meaning that its value is not equal to any pin and does not contain any characters or operators, such as &, |, !, ^, +, *, =, and ~, the Verilog model generator adds a function description for the one-bit signal, with gate instantiation, based on the logic of the when condition in the same timing group.

When there are state-dependent timing arcs in the input Liberty file, ifnone is used to define the default timing group. It is used only if the input Liberty file has default timing groups.

Either of the following criteria identify default timing groups:

- The default_timing : true setting is found in a timing group. In this case, the timing group is used as the default timing group. Besides its when condition (sdf_cond) being used to model a state-dependent path delay, the timing group is used again to model a default timing arc using ifnone, ignoring that the timing group has a when condition.
- Among timing groups with the same related_pin attribute, if one timing group does not have a when condition and all the other timing groups have when conditions, the one without the when condition is regarded as the default timing group and is used to model a default path delay, using ifnone.

.lib Example

```
pin Z {
    timing() {
        related_pin      : "D";
        timing_sense     : negative_unate;
        when             : "(A'*B'*C')";
        sdf_cond         : "ALBLCL";
    }
}
```

Verilog Example

```
not U2 (_net_1, B);
not U3 (_net_2, A);
not U4 (_net_3, C);
```

```
and U5 (ALBLCL, _net_1, _net_2, _net_3);
…
if (ALBLCL) (D -=> Z)=(tdelay_D_Z_01_0, tdelay_D_Z_10_0);
```

Delay Values

Generally, there is output rise and output fall for a path delay. The Verilog model generator uses two values (t01, t10) for one path delay entry. With three-state timing arcs, six values are modeled in one path delay entry, as follows:

- t01, t10, t0z, tz1, t1z, tz0
- After a `three_state_enable` timing group is found, the six values (t01, t10, t0z, tz1, t1z, tz0) are used regardless of whether there are corresponding `three_state_disable` rising or falling arcs. The Verilog model generator always declares six path delay `specparams` for these six delay values.

Example

```
(EN+ => O)=(tdelay_EN_O_Z1_0, tdelay_EN_O_Z0_0, tdelay_EN_O_Z1_0,
tdelay_EN_O_Z1_0, tdelay_EN_O_Z0_0, tdelay_EN_O_Z0_0);
```

Bus and Bundle Signal

Based on the input Liberty content, if the delay applies to all members of the bundle or bus, then the bundle or bus name is used in the path delay. Otherwise, each bundle or bus member signal is modeled in separate path delay entries. When referring to individual bus members, `bus_naming_style` in Liberty is considered to convert a Liberty bus member name to a Verilog bus member name because Verilog only allows the `%s[%d]` style.

Duplicate Path Delay

Timing groups in Liberty might result in multiple path delays with the same state condition, same edge for the same input and output, and therefore duplicate path delays. In this case, the Verilog model generator avoids generating duplicate path delays. If a potential duplicate path delay is found, the Verilog model generator models the path delay one time and skips later timing groups for those duplicate path delays.

Timing Checks

This section includes timing checks for `setup`, `hold`, `setuphold`, `recovery`, `removal`, `recrem`, pulse `width`, `period`, `skew`, and `include` edge statements and conditional timing checks and duplicate timing checks.

Setup

The `setup` syntax is as follows:

Syntax

```
$setup ([edge_identifier_start] start_pin [&&& start_cond],
[edge_identifier_end] end_pin [&&& end_cond], setup_param, notifier);
```

The arguments are as follows:

- `edge_identifier_start` (optional)

  If specified, `edge_identifier_start` is the edge associated with the start pin. See "Edge Statements" on page B-61.

- `start_pin`

  Specifies the data pin.

- `start_cond` (optional) based on the input Liberty file. See "Conditional Timing Checks" on page B-66.

- `edge_identifier_end` is optional, based on the input Liberty file. If specified, it is the edge associated with the end pin. See "Edge Statements" on page B-61.

- `end_pin` is the clock pin.

- `end_cond` is optional and is dependent on the input Liberty file. See "Conditional Timing Checks" on page B-66.

- `setup_param` is the timing limit. For naming conventions, see "specparam" on page B-47.

- `notifier` is a `reg` variable used to detect timing check violations behaviorally. Whenever a timing violation occurs, the system task updates the value of the notifier. The Verilog model generator uses notifier to set the UDP output to x when a timing check violation occurs. All timing checks share the same notifier.

- The `$setup` timing check is modeled only when there is no matching hold timing check found in the Liberty timing group. See "Setuphold" on page B-55 for more information.

You can use the `veriloglib_combine_timingcheck` variable to combine setup and hold timing checks to one `$setuphold` timing check. If a setup timing check and a hold timing check have the same clock and data pin and the same edge identifier and condition for a clock and data pin, respectively, the two timing checks are combined into one `$setuphold` timing check. However, `no_edge` is regarded as a special edge identifier and "no condition" is regarded as a special condition.

Valid values for the `veriloglib_combine_timingcheck` variable include `setuphold` and `recrem`, for example:

```
set veriloglib_combine_timingcheck {setuphold}
```

The Verilog model generator does not combine setup and hold timing checks by default.

You can set the `veriloglib_write_recrem_as_setuphold` variable to `true` to output timing checks in the same format but with the keyword `$recrem` replaced by `$setuphold`, `$recovery` replaced by `$setup`, or `$removal` replaced by `$hold`. The valid values are `true` and `false`. The default is `false`.

Hold

The `hold` syntax is as follows:

Syntax

```
$hold ([edge_identifier_start] start_pin [&&& start_cond],
[edge_identifier_end] end_pin [&&& end_cond], hold_param, notifier);
```

The arguments are as follows:

- *`edge_identifier_start`* (optional)

  If specified, *`edge_identifier_start`* is the edge associated with the start pin. See "Edge Statements" on page B-61.
- *`start_pin`* is the clock pin.
- *`start_cond`* is optional and is dependent on the Liberty input file. See "Conditional Timing Checks" on page B-66.
- *`end_pin`* is the data pin.
- *`end_cond`* is optional and is dependent on the input Liberty file. See "Conditional Timing Checks" on page B-66.
- *`hold_param`*

  Specifies the timing limit. For naming conventions, see "specparam" on page B-47.
- `notifier` is a `reg` variable used to detect timing check violations behaviorally. For more information, see "Setup" on page B-52.
- The `$hold` timing check is modeled only when there is no matching setup timing check found in the Liberty timing group. For more information, see "Setuphold" on page B-55.

You can use the `veriloglib_combine_timingcheck` variable to combine setup and hold timing checks to one `$setuphold` timing check. If a setup timing check and a hold timing check have the same clock and data pin and the same edge identifier and condition for a clock and data pin, respectively, the two timing checks are combined into one `$setuphold` timing check. However, `no_edge` is regarded as a special edge identifier and "no condition" is regarded as a special condition.

Valid values for the `veriloglib_combine_timingcheck` variable include `setuphold` and `recrem`, for example:

```
set veriloglib_combine_timingcheck {setuphold}
```

The Verilog model generator does not combine setup and hold timing checks by default.

You can set the `veriloglib_write_recrem_as_setuphold` variable to `true` to output timing checks in the same format but with the keyword `$recrem` replaced by `$setuphold`, `$recovery` replaced by `$setup`, or `$removal` replaced by `$hold`. The valid values are `true` and `false`. The default is `false`.

Setuphold

The `setuphold` syntax is as follows:

Syntax

```
$setuphold ([edge_identifier_clock] clock_pin[&&&clock_cond],
[edge_identifier_data] data_pin [&&&data_cond], setup_param, hold_param,
notifier);
```

The arguments are as follows:

- *edge_identifier_clock* is optional and is based on the input Liberty file. If specified, it is the edge associated with the clock pin. See “Edge Statements” on page B-61.
- *clock_pin* is the *start_pin* for the associated hold timing check and *end_pin* for the associated setup timing check.
- *clock_cond* is optional and is dependent on the input Liberty file. See “Conditional Timing Checks” on page B-66.
- *edge_identifier_data* is optional and is based on the input Liberty file. If specified, it is the edge associated with the data pin. See “Edge Statements” on page B-61.
- *data_pin* is the *end_pin* for the associated hold timing check and *start_pin* for the associated setup timing check.
- *data_cond* is optional and is dependent on the input Liberty file. See “Conditional Timing Checks” on page B-66.
- *setup_param* is the timing limit for the associated setup timing check. For naming conventions, see “specparam” on page B-47.
- *hold_param* is the timing limit for the associated hold timing check. For naming conventions, see “specparam” on page B-47.
- `notifier` is a `reg` variable used to detect timing check violations behaviorally. For more information, see “Setup” on page B-52.
- The Verilog model generator tries to combine every setup and hold timing check into a single `$setuphold` timing check. If a setup timing check and a hold timing check have the same clock and data pin and the same edge identifier and condition for a clock and data pin, respectively, the two timing checks are combined into one `$setuphold` timing check. However, `no_edge` is regarded as a special edge identifier and “no condition” is regarded as a special condition.

You can use the `veriloglib_combine_timingcheck` variable to combine setup and hold timing checks to one `$setuphold` timing check, as shown:

```
set veriloglib_combine_timingcheck {setuphold}
```

Valid values for the `veriloglib_combine_timingcheck` variable include `setuphold` and `recrem`. The Verilog model generator does not combine setup and hold timing checks by default.

You can set the `veriloglib_write_recrem_as_setuphold` variable to `true` to output timing checks in the same format but with the keyword `$recrem` replaced by `$setuphold`, `$recovery` replaced by `$setup`, or `$removal` replaced by `$hold`. The valid values are `true` and `false`. The default is `false`.

Recovery

The `recovery` syntax is as follows:

Syntax

```
$recovery ([edge_identifier_control] control_pin [&&& control_cond],
[edge_identifier_clock] clock_pin [&&& clock_cond], recovery_param,
notifier);
```

The arguments are as follows:

- *`edge_identifier_control`* is optional, based on the input Liberty file. If specified, it is the edge associated with the control pin. See “Edge Statements” on page B-61.
- *`control_pin`* is the asynchronous control pin, such as clear and preset. It is the *`start_pin`* for the recovery timing check.
- *`control_cond`* is optional and is dependent on the input Liberty file. See “Conditional Timing Checks” on page B-66.
- *`edge_identifier_clock`* is optional, based on the input Liberty file. If specified, it is the edge associated with the clock pin. See “Edge Statements” on page B-61.
- *`clock_pin`* is the clock pin. It is the *`end_pin`* for the recovery timing check.
- *`clock_cond`* is optional and is dependent on the input Liberty file.
- *`recovery_param`* is the timing limit. For naming conventions, see “specparam” on page B-47.
- `notifier` is a `reg` variable used to detect timing check violations behaviorally. See “Setup” on page B-52 for more information.
- `$recovery` timing check is modeled only when there is no matching removal timing check in the Liberty timing groups. See “Recrem” on page B-58.

You can use the `veriloglib_combine_timingcheck` variable to combine recovery and removal timing checks to one `$recrem` timing check. If a recovery timing check and a removal timing check have the same clock and control pin and the same edge identifier and condition for a clock and control pin, respectively, the two timing checks are combined into one `$recrem` timing check.

Valid values for the `veriloglib_combine_timingcheck` variable include `setuphold` and `recrem`, for example:

```
set veriloglib_combine_timingcheck {recrem}
```

The Verilog model generator does not combine recovery and removal timing checks by default.

You can set the `veriloglib_write_recrem_as_setuphold` variable to `true` to output timing checks in the same format but with the keyword `$recrem` replaced by `$setuphold`, `$recovery` replaced by `$setup`, or `$removal` replaced by `$hold`. The valid values are `true` and `false`. The default is `false`.

Removal

The `removal` syntax is as follows:

Syntax

```
$removal ([edge_identifier_control] control_pin [&&& control_cond],
[edge_identifier_clock] clock_pin [&&& clock_cond], removal_param,
notifier);
```

The arguments are as follows:

- *`edge_identifier_control`* (optional) based on the input Liberty file. If specified, it is the edge associated with the control pin. See "Edge Statements" on page B-61.
- *`control_pin`*

  Specifies an asynchronous control pin, such as clear and preset. It is the end pin of a removal timing check.
- *`control_cond`* (optional) dependent on the input Liberty file. See "Conditional Timing Checks" on page B-66.
- *`edge_identifier_clock`* (optional) based on the input Liberty file. If specified, it is the edge associated with the clock pin. See "Edge Statements" on page B-61.
- *`clock_pin`*

  Specifies the clock pin. It is the start pin of a removal timing check.
- *`clock_cond`* is dependent on the input Liberty file.

- *removal_param* is the timing limit. For naming conventions, see "specparam" on page B-47."
- notifier is a reg variable used to detect timing check violations behaviorally. see "Setup" on page B-52 for more information.
- The $removal timing check is modeled only when there is no matching recovery timing check in the Liberty timing groups. See "Recrem" on page B-58 for more information

You can use the veriloglib_combine_timingcheck variable to combine recovery and removal timing checks to one $recrem timing check. If a recovery timing check and a removal timing check have the same clock and control pin and the same edge identifier and condition for a clock and control pin, respectively, the two timing checks are combined into one $recrem timing check.

Valid values for the veriloglib_combine_timingcheck variable include setuphold and recrem, for example:

```
set veriloglib_combine_timingcheck {recrem}
```

The Verilog model generator does not combine recovery and removal timing checks by default.

You can set the veriloglib_write_recrem_as_setuphold variable to true to output timing checks in the same format but with the keyword $recrem replaced by $setuphold, $recovery replaced by $setup, or $removal replaced by $hold. The valid values are true and false. The default is false.

Recrem

The recrem syntax is as follows:

Syntax

```
$recrem ([edge_identifier_control] control_pin[&&control_cond],
[edge_identifier_clock] clock_pin [&&clock_cond], recovery_param,
removal_param, notifier);
```

The arguments are as follows:

- *edge_identifier_control* (optional), based on the input Liberty file. If specified, it is the edge associated with the control pin. See "Edge Statements" on page B-61.
- control_pin is the asynchronous control pin, such as clear and preset.
- *control_cond* (optional) and is dependent on input Liberty file. See "Conditional Timing Checks" on page B-66.
- *edge_identifier_clock* is optional, based on the input Liberty file. If specified, it is the edge associated with the clock pin. See "Edge Statements" on page B-61.

- `clock_pin`

  Specifies the clock pin.

- `clock_cond` is dependent on the input Liberty file. See "Conditional Timing Checks" on page B-66.
- `recovery_param` is the timing limit for the associated recovery timing check. For naming conventions, see "specparam" on page B-47.
- `removal_param`: timing limit for the associated removal timing check. For the naming convention, see "specparam" on page B-47.
- `notifier` is a `reg` variable used to detect timing check violations behaviorally. See "Setup" on page B-52 for more information.

The Verilog model generator tries to combine every recovery and removal timing check to a single `$recrem` timing check. If a recovery timing check and a removal timing check have the same clock and control pin and the same edge identifier and condition for a clock and control pin, respectively, the two timing checks are combined into one `$recrem` timing check.

You can use the `veriloglib_combine_timingcheck` variable to combine recovery and removal timing checks to one `$recrem` timing check, as shown:

```
set veriloglib_combine_timingcheck {recrem}
```

Valid values for the `veriloglib_combine_timingcheck` variable include `setuphold` and `recrem`. The Verilog model generator does not combine recovery and removal timing checks by default.

You can set the `veriloglib_write_recrem_as_setuphold` variable to `true` to output timing checks in the same format but with the keyword `$recrem` replaced by `$setuphold`, `$recovery` replaced by `$setup`, or `$removal` replaced by `$hold`. The valid values are `true` and `false`. The default is `false`.

Pulse Width

The pulse `width` syntax is as follows:

Syntax

```
$width (edge_identifier pin[&&&<cond>], pulsewidth_param, 0 ,
notifier);
```

The arguments are as follows:

- `edge_identifier`

  Specifies the start event edge associated with the constraint pin based on the input Liberty file. See "Edge Statements" on page B-61.

- *pin*

  Specifies the constraint pin.

- *cond* is optional and is dependent on the input Liberty file. See "Conditional Timing Checks" on page B-66.

- *pulsewidth_param*

  Specifies the timing limit for the width timing check. The default threshold is `0`.

- `notifier` is a `reg` variable used to detect timing check violations behaviorally. For more information, see "Setup" on page B-52.

Period

The `period` syntax is as follows:

Syntax

```
$period (edge_identifier pin[&&&cond], period_param, notifier);
```

The arguments are as follows:

- *edge_identifier*

  Specifies the start event edge associated with the constraint pin, based on the input Liberty file. See "Edge Statements" on page B-61.

- *pin*

  Specifies the constraint pin.

- *cond* is optional and is dependent on the input Liberty file. See "Conditional Timing Checks" on page B-66.

- *period_param*

  Specifies the timing limit for the period timing check.

- `notifier` is a `reg` variable used to detect timing check violations behaviorally. For more information, see "Setup" on page B-52.

Skew

The `skew` syntax is as follows:

Syntax

```
$skew ([edge_identifier_start] start_pin [&&& start_cond],
[edge_identifier_end] end_pin [&&& end_cond], skew_param, notifier);
```

The arguments are as follows:

- *`edge_identifier_start`* (optional) based on the input Liberty file.

  If specified, *`edge_identifier_start`* is the edge associated with the start pin. See "Edge Statements" on page B-61.

- *`start_pin`*

  Specifies the start clock pin. It is the `related_pin` of the timing group in the input Liberty file.

- *`start_cond`* is optional and is dependent on the input Liberty file. See "Conditional Timing Checks" on page B-66.

- *`edge_identifier_end`* is optional and is based on the input Liberty file. If specified, it is the edge associated with the end pin. See "Edge Statements" on page B-61.

- *`end_pin`*

  Specifies the end clock pin. It is the `parent_pin` of the timing group in the input Liberty file.

- *`end_cond`* is optional and is dependent on the input Liberty file. See "Conditional Timing Checks" on page B-66.

- *`skew_param`*

  Specifies the timing limit for the skew timing check. For the naming convention, see "specparam" on page B-47.

- `notifier` is a `reg` variable used to detect timing check violations behaviorally. See "Setup" on page B-61 for more information.

Edge Statements

The following rules apply to setup, hold, recovery, removal, and skew timing checks:

- The `sdf_edges` attribute in the input Liberty file defines the edge specification on the start pin and the end pin. The `sdf_edges` attribute can be one of the following edge types: `noedge`, `start_edge`, `end_edge`, or `both_edges`. The default is `noedge`.
- If the `sdf_edge` attribute does not specify an edge, the exact edge is determined based on the attributes in the timing group (as described in the following sections) for each timing check that must be specified.

Setup

The following rules apply to setup timing checks:

- The `rise_constraint` group or the `intrinsic_rise` attribute denotes `posedge` for the start pin (data pin).

- The `fall_constraint` group or the `intrinsic_fall` attribute denotes `negedge` for the start pin (data pin).
- A value of `setup_rising` for the `timing_type` attribute denotes `posedge` for the end pin (clock pin).
- A value of `setup_falling` for the `timing_type` attribute denotes `negedge` for the end pin (clock pin).

Hold

The following rules apply to hold timing checks:

- A value of `hold_rising` for the `timing_type` attribute denotes `posedge` for the start pin (clock pin).
- A value of `hold_falling` for the `timing_type` attribute denotes `negedge` for the start pin (clock pin).
- The `rise_constraint` group or the `intrinsic_rise` attribute denotes `posedge` for the end pin (data pin).
- The `fall_constraint` group or the `intrinsic_fall` attribute denotes `negedge` for the end pin (data pin).

Edge Statements in $setup and $hold Timing Check .lib Example

```
cell (DFFR1X2) {
      ff (IQ,IQN) {
            next_state : "D";
            clocked_on : "CK'";
            clear : "R'";
      }
      pin (D) {
            direction : "input";
            rise_capacitance : 0.012667;
            fall_capacitance : 0.012721;
            timing () {
                  related_pin : "CK";
                  timing_type : "setup_falling";
                  rise_constraint ("vio_3_3_1") {
                        index_1("…");
                        index_2("…");
                        values("…", "…", "…");
                  }
                  fall_constraint ("vio_3_3_1") {
                        index_1("…");
                        index_2("…");
                        values("…", "…", "…");      }
            }
            timing () {
                  related_pin : "CK";
                  timing_type : "hold_falling";
```

```
                        rise_constraint ("vio_3_3_1") {
                             index_1("…");
                             index_2("…");
                             values("…", "…", "…");
            }
                        fall_constraint ("vio_3_3_1") {
                             index_1("…");
                             index_2("…");
                             values("…", "…", "…");
            }
        }
    }
    …
}
```

The `setup_falling` and `hold_falling` values in the .lib example correspond to `negedge CK` in the following Verilog example. The `rise_constraint` value corresponds to `posedge D` in the example, and `fall_constraint` corresponds to `negedge D`.

Edge Statements in $setup and $hold Timing Checks Verilog Example

```
$setuphold(negedge CK , posedge D , tsetup_D_CK_0 ,thold_CK_D_0,
notifier);
$setuphold(negedge CK , negedge D , tsetup_D_CK_1 ,thold_CK_D_1,
```

Recovery

The following rules apply to recovery timing checks:

- The `rise_constraint` group or the `intrinsic_rise` attribute denotes `posedge` for the start pin (control pin).
- The `fall_constraint` group or the `intrinsic_fall` attribute denotes `negedge` for the start pin (control pin).
- A value of `recovery_rising` for the `timing_type` attribute denotes `posedge` for the end pin (clock pin).
- A value of `recovery_falling` for the `timing_type` attribute denotes `negedge` for the end pin (clock pin).

Removal

The following rules apply to removal timing checks:

- A value of `removal_rising` for the `timing_type` attribute denotes `posedge` for the start pin (clock pin).
- A value of `recovery_falling` for the `timing_type` attribute denotes `negedge` for the end pin (clock pin).

Edge Statements in $recover and $removal Timing Check .lib Example

```
pin (R) {
                direction : "input";
                rise_capacitance : 0.021705;
                rise_capacitance_range(0.020768,0.022129);
                capacitance : 0.021528;
                fall_capacitance : 0.021528;
                fall_capacitance_range(0.019914,0.023608);
                timing () {
                     related_pin : "CK";
                     timing_type : "recovery_falling";
                     rise_constraint ("vio_3_3_1") {
                          index_1("…");
                          index_2("…");
                          values("…", "…", "…");
                     }
                }
                timing () {
                     related_pin : "CK";
                     timing_type : "removal_falling";
                     rise_constraint ("vio_3_3_1") {
                          index_1("…");
                          index_2("…");
                          values("…", "…", "…");
                     }
                }
                timing () {
                     related_pin : "R";
                     timing_type : "min_pulse_width";
                     fall_constraint ("constraint_3_0_1") {
                          index_1("…");
                          values("…");
                     }
                }
           }
```

The `recovery_falling` and `removal_falling` values in the .lib example correspond to `negedge CK` in the following Verilog example and the `rise_constraint` value corresponds to `posedge R`.

Edge Statements in $recover and $removal Timing Check Verilog Example

```
$recrem(posedge R, negedge CK, trecovery_R_CK_0, tremoval_CK_R_0,
```

Skew

The following rules apply to skew timing checks:

- A value of `skew_rising` for the `timing_type` attribute denotes `posedge` for the start pin, which is the start clock pin, the `related_pin` in the timing group.

- A value of skew_falling for the timing_type attribute denotes negedge for the start pin, which is the start clock pin, the related_pin in the timing group.
- The rise_constraint group or intrinsic_rise attribute denotes posedge for the end pin, which is the end clock pin, the parent_pin of timing group.
- The fall_constraint group or intrinsic_fall attribute denotes negedge for the end pin, which is the end clock, pin, or parent_pin of the timing group.

Pulsewidth

Liberty provides the following ways to specify pulse width and to deduce the edge on a pulse width constraint pin.

For a simple attribute in a pin group,

- The min_pulse_width_high attribute denotes posedge on the constraint pin.
- The min_pulse_width_low attribute denotes negedge on the constraint pin.

For a min_pulse_width group in the pin group,

- The constraint_high attribute denotes posedge on the constraint pin.
- The constraint_low attribute denotes negedge on the constraint pin.

For a timing group with a value of min_pulse_width for the timing_type attribute,

- The rise_constraint group denotes posedge on the constraint pin.
- The fall_constraint group denotes negedge on the constraint pin.

Period

Liberty provides the following ways to specify period and to deduce the edge on the period constraint pin:

- For the minimum_period attribute, which is a simple attribute in a pin group,
  - The edges for period checks cannot be identified from the attribute itself. First, the Verilog model generator checks for a timing group in the output pin groups with a value of rising_edge or falling_edge for the timing_type attribute, where the value of the related_pin attribute is the same clock pin as in the period checks.
  - A value of rising_edge denotes posedge, and a value of falling_edge denotes negedge.
  - If a similar timing group cannot be found, the Verilog model generator issues a warning and assumes a posedge for the period check.
- For the minimum_period group in the pin group, the Verilog model generator checks for the existence of the constraint attribute.

Note:
For edge determination, the same rules as for the `minimum_period` attribute apply.

- For a `timing` group with a `minimum_period` value for the `timing_type` attribute,
  - The `rise_constraint` group denotes `posedge` on the constraint pin.
  - The `fall_constraint` group denotes `negedge` on the constraint pin.

Conditional Timing Checks

For setup, hold, recovery, removal, and skew timing checks, the following attributes are related to pin conditions: `when`, `when_start`, `when_end`, `sdf_cond`, `sdf_cond_start`, and `sdf_cond_end`. The `sdf_cond_start` attribute maps to the start pin condition; the `sdf_cond_end` attribute maps to the end pin condition. When none of these attributes exist, no condition is applied to the corresponding pin, whether it is a start pin or an end pin.

For pulse width, the Verilog model generator checks the following conditions for timing:

- If there is a simple attribute in pin group.
- If there is no condition for a constraint pin.
- For a `min_pulse_width` group in pin group, the Verilog model generator checks to see if the `when` and `sdf_cond` attributes are related to the constraint pin condition. It also checks to see if the `sdf_cond` attribute is used directly as the constraint pin condition if it exists.
- When the `timing` group has a value of `min_pulse_width` for the `timing_type` attribute, the Verilog model generator checks to make sure that the `when` and `sdf_cond` attributes are related to the constraint pin condition. It also checks to make sure that the `sdf_cond` attribute is used directly as the constraint pin condition if it exists.

For period, the Verilog model generator checks the following conditions for timing:

- If there is a simple attribute in pin group.
- If there is no condition for a constraint pin.
- For a `minimum_period` group in pin group, the Verilog model generator checks to see if the `when` and `sdf_cond` attributes are related to the constraint pin condition. It also checks to see if the `sdf_cond` attribute is used directly as the constraint pin condition if it exists.
- When the `timing` group has a value of `minimum_period` for the `timing_type` attribute, the Verilog model generator checks to make sure that the `when` and `sdf_cond` attributes are related to the constraint pin condition. It also checks to make sure that the `sdf_cond` attribute is used directly as the constraint pin condition if it exists.

As with the state dependent path delay, if the `sdf_cond`, `sdf_cond_start`, or `sdf_cond_end` attributes are recognized as “single-bit signal,” the Verilog model generator adds a function description for this one-bit signal by instantiating a gate, based on the corresponding logic of the `when`, `when_start`, or `when_end` attributes. For more information, see “State-Dependent Path Delay” on page B-51.

Duplicate Timing Checks

If timing groups in Liberty have multiple timing checks with the same type, same edge identifier, pin, and condition for both the start and end pins, these timing checks are considered to be duplicate timing checks. The Verilog model generator avoids generating duplicate timing checks. However, if potential duplicate timing checks occur, the Verilog model generator generates only one entry for the timing check and skips the later duplicate timing checks.

UDP Definition

The UDP definition syntax is as follows:

```
primitive UDP_identifier (UDP_pin_list);  //UDP header;
UDP_pin_declaration;  //UDP pin declaration;
table                                 //UDP body or UDP state
table_content
endtable
endprimitive
```

The *`UDP_identifier`* in the UDP header is in the format `UDP_`*`cell_port`*, where the arguments are as follows:

- *`cell`* and *`port`* describe the behavior of the UDP. The *`port`* value can be either output pin or internal node.
- *`UDP_ pin_list`* is a comma-separated list containing the output and input pins. One UDP has exactly one output pin. The output pin is the first pin in the port list.

For timing check violations, a `notifier` pin is explicitly added to the end of the pin list as an input signal. The order of the input pins in the pin list determines the order of the inputs in the UDP state table definition. For more information, see the example at the end of this section.

The UDP pin declaration is as follows:

```
output q;
reg q;
input input_pin[, input_pin…], notifier
```

where the *`input_pin`* values are named `in1`, `in2`, and so on.

The UDP state table is as follows:

```
table
input_state[ input_state…] : current_state : next_state;
…
Endtable
```

The arguments are as follows:

- *input_state*

  Specifies a space-separated list of input values. Valid characters for the input values are 0, 1, r, f, b, ?, x and *.

- *current_state*

  Specifies the output current state value. Valid characters are 0, 1, x, ?, and b.

- *next_state*

  Specifies the output next state value. Valid characters are 0, 1, x, ?, and -. Each row defines the output, based on the current state, particular combinations of input values, and one input transition at the most. The order of the input state fields of each row of the state table is taken directly from the port list in the UDP definition header. Any event on notifier puts the output in the x state.

Three-valued logic tends to make pessimistic estimates of the output when one or more inputs are unknown. UDPs can be used to reduce this pessimism. The Verilog model generator provides the ability to write pessimism reduction in UDP.

To reduce pessimism in UDP, the general rule is that “x” represents either 0 or 1. If an input pin can be set to either 0 or 1 while having no impact on the output state, even if the state of the input is unknown, there won't be any impact on the output state. There are other rules for certain conditions, such as ignoring certain edges, which must be added specially.

Example

```
primitive UDP_DFF1X1_Q (q, in1, in2, notifier);
output q;
reg q;
input in1, in2, notifier;
table
CP     D    notifier         : Q    Q+1 ;
(01)   0     ?         :?     : 0 ;
(01)   1     ?         :?     : 1 ;
(1?)   ?     ?         :?     : - ;
(?0)   ?     ?         :?     : - ;

?      *     ?         :?     : - ;
?      ?     *         :?     : x ;
//Added as part of pessimism reduction
x1    0      ?        : 0   : 0 ;
```

```
0x    0      ?         : 0    : 0 ;
x1    1      ?         : 1    : 1 ;
0x    1      ?         : 1    : 1 ;
endtable

endprimitive
```

Creating the Testbench

Library Compiler can automatically generate testbenches with expected outputs for the entire library during Verilog generation. To automatically generate library-level testbenches with input stimuli and expected outputs for the cell during Verilog generation, set the `veriloglib_tb_compare` variable to a value from 0 to 5, as shown in the following example:

```
set veriloglib_tb_compare =  [0 | 1 | 2 | 3 | 4 | 5 ]
```

The values are described as follows:

`0` (the default)

If you set the variable to 0, the default setting, no testbench is created.

`1`

If you set the variable to 1, each time an input changes, Library Compiler checks to ensure that the result from the previous input matches the expected result. You should define enough time between the input vectors for the outputs to propagate and stabilize. This check is useful for unit-delay structural libraries.

`2`

If you set the variable to 2, each time an expected output changes, Library Compiler checks to ensure that the actual output has changed or is changing at the same time. This check is useful if you are unable to define some of the expected outputs. It can also be used if the expected output signals are timed pessimistically and if changes usually occur later. The actual outputs can have many unpredictable changes that are not detected. This check reports known pins and does not check unknown pins.

`3`

If you set the variable to 3, each time an expected output changes, Library Compiler checks to ensure that the actual output makes an identical change at the same time, verifying the correct pin state and transition time.

`4`

If you set the variable to 4, Library Compiler checks each expected output against the corresponding actual output to ensure they are identical at the same time. When the actual output pins are active, Library Compiler checks to ensure that the expected output signals are the same at the same time.

`5`

If you set the variable to 5, Library Compiler checks each expected output against the corresponding actual output to ensure they are identical at the same time. When the actual output pins are active, Library Compiler checks to ensure that the expected output signals are the same at the same time.

When you set `veriloglib_tb_compare` to a value from 1 to 5, enabling testbench generation, Library Compiler creates the following files:

- Testbench files

  The testbench files, generated in Verilog format, specify the SDF file for back annotation and the hierarchical path of the cell instance. They do this by using the `$sdf_annotate` function. Library Compiler reads the input stimulus file, applies the input stimulus, writes out the simulation output changes, compares the output with any expected output, and reports the differences. Library Compiler creates or defines a netlist of the Verilog model instances being tested and writes out a testbench file for each library that is tested and a testbench file for each cell. Testbench files use the contents of the input stimulus file as a test vector.

  The testbench file name format is *library*_tb.v for an entire library and *library_cell*_tb.v for a specific cell. The simulation output file name format is *library*_tb.out or *library_cell*_tb.out. The output file format is compatible with the input stimulus file. The SDF file name format is o_*cell*.sdf and the module instantiation name format is *cell*_inst. The timescale is hard-coded as `` `timescale 1 ns/10 ps``.

- Input stimulus files

  The *library*_tb.sen and *library_cell*_tb.sen input stimulus files contain a pin list with expected outputs in the following format for each cell in the library:

  ```
  Input1; Input2; Input3; Output1; Output2; /*pin order list*/
  $
  absolute-time1 input-stimulus1 #expected-outputs1
  absolute-time2 input-stimulus2 #expected-outputs2
  absolute-time3 input-stimulus3 #expected-outputs3
  ```

  where `absolute-time` is expressed in `time_units`, as defined in the testbench file, and the characters used for `input-stimulus` and `expected-outputs` are in IEEE Std 1164.1 format.

  The Verilog model generator creates one .sen file for an entire library when testbenches are written out. The *library*_tb.v file uses the contents of the .sen file as the test vectors. The .sen files that the Verilog model generator writes out contain the absolute time and input stimulus only. (You can add expected outputs and comments later.)

  The pin list must begin with a blank line and terminate with a dollar sign ($). The stimulus vectors are then listed after the dollar sign with the time they are to be applied. The input and output pin list is optional. Testbench generation does not output a pin list. By default, the stimulus file lists pin names in the order they are specified in the *cell*.v file. You can define a different pin order in your individual cell testbenches by including the pin order at the beginning of the input stimulus file before the dollar sign.

- Script files

  The Verilog model generator writes out the following script files:

  - *library*.dc.tcl, a dc_shell script that generates an SDF file for each cell. The dc_shell script is as follows:

    ```
    set link_library lib.db
    read_verilog -netlist wrapper_library.v
    set designs [get_designs *]
    foreach_in_collection ds $designs {
    set d_name [get object_name $ds]
    current_design $d_name
    write_sdf $d_name.sdf
    }
    ```

    The Verilog model generator automatically creates a wrapper_*library*.v file. The dc_shell script reads the wrapper file, which constructs one module, or design, for each cell. The modules are also packed into one wrapper_*library*.v file. You must create a .db file from the input .lib file before using the dc_shell script. The input .lib file generates a *lib*.db file during Verilog generation.

  - *library*.pt.tcl, a pt_shell script that generates an SDF file for each cell. The pt_shell script is as follows:

    ```
    set link_library library.db
    set i 0
    read_verilog wrapper_library.v
    set designs [get_designs *]
    foreach_in_collection ds $designs {
          set d_name [get_object_name $ds]
          if {$i==0} {
                set ld $d_name
          }
          else {
                lappend ld $d_name
          }
          incr i
    }
    foreach dl_name $ld {
          read_verilog wrapper_library_$dl_name.v
          link_design $dl_name
          write_sdf $dl_name.sdf [-include SETUPHOLD] [ -include RECREM]
    }
    quit
    ```

The Verilog model generator automatically creates a wrapper_*library*.v file. The pt_shell script reads the wrapper file, which constructs one module, or design, for each cell. Each cell also has a specific wrapper_*library*_o_*cell*.v file. You must create a .db file from the input .lib file before using the pt_shell script. The input .lib file generates a *lib*.db file during Verilog generation.

If the `veriloglib_combine_timingcheck` variable is set to the `setuphold` value, `write_sdf` will include the `-include SETUPHOLD` value. If `veriloglib_combine_timingcheck` is set to `recrem`, `write_sdf` will include the `-include RECREM` value.

- *library* .csh, a C shell script that runs library verification.

  The C shell script calls the VCS simulator to run testbench simulation. You can use the following options with the `library .csh` command:

  ```
  library .csh -tb
  ```

  The `-tb` option skips the Verilog model compilation step. If you have already run the `.csh` script one time, the library has already been compiled. Using the `-tb` option allows you to skip this step and save time.

  ```
  library .csh -sp
  ```

  The `-sp` option omits the portion of the script that runs the testbench on the entire library. Using the `-sp` option allows you to skip the entire library check. The `-lib` option specifies whether a specific cell is tested. The script includes a cell list. If you want to test only a few cells, you can edit the script and remove the cells you do not want to test.

  ```
  library .csh -lib
  ```

  The `-lib` option omits the portion of the script that runs the testbenches on specific cells. Using the `-lib` option allows you to skip the check on individual cells.

During Verilog testbench generation, Library Compiler creates a `testbench` directory under the Verilog model directory and puts all generated files into that directory. (The default directory is `library_verilog` under the current working directory.) If a `testbench` directory already exists, the newly generated files overwrite the existing files in the directory and Library Compiler issues a warning message.

When the testbench compares the expected and actual output, an X (unknown) is equal only to another X if the `veriloglib_tb_x_eq_dontcare` variable is set to false, as shown:

```
set veriloglib_tb_x_eq_dontcare = false
```

If `veriloglib_tb_x_eq_dontcare` is set to `true`, an X is a “don't care” value and is equal to any logic value. If you do not define `veriloglib_tb_x_eq_dontcare`, the default is `false`.

Index