Library Compiler™ Physical Libraries User Guide

Version X-2005.09, September 2005

SYNOPSYS®

Copyright Notice and Proprietary Information

Right to Copy Documentation

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

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

Destination Control Statement

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

Disclaimer

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

Registered Trademarks (®)

Synopsys, AMPS, Astro, Behavior Extracting Synthesis Technology, Cadabra, CATS, Certify, CHIPit, Design Compiler, DesignWare, Formality, HDL Analyst, HSIM, HSPICE, Identify, iN-Phase, Leda, MAST, ModelTools, NanoSim, OpenVera, PathMill, Physical Compiler, PrimeTime, SCOPE, Simply Better Results, SiVL, SNUG, SolvNet, Syndicated, Synplicity, Synplify, Synplify Pro, Synthesis Constraints Optimization Environment, TetraMAX, the Synplicity logo, UMRBus, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

Trademarks (™)

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

Service Marks (^{SM})

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

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

Contents

5. Parasitic RC Estimation in the Physical Library

Appendix A. Calculating the Antenna

Appendix B. Generating Physical Library Reports

Index

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 Manual

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

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

Following is a description of each volume's contents:

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

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

Audience

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

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

Related Publications

For additional information about Library Compiler, see Documentation on the Web, which is available through SolvNet at the following address:

https://solvnet.synopsys.com/DocsOnWeb

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

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

Conventions

The following conventions are used in Synopsys documentation.

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

Customer Support

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

Accessing SolvNet

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

To access SolvNet, go to the SolvNet Web page at 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 or in the footer.

Contacting the Synopsys Technical Support Center

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

- Open a call to your local support center from the Web by going to https://solvnet.synopsys.com, entering your user name and password and then clicking “Enter a Call to the Support Center.”
- Send an e-mail message to your local support center.
  - E-mail support_center@synopsys.com 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 the continental United States.
  - Call (650) 584-4200 from Canada.
  - Find other local support center telephone numbers at http://www.synopsys.com/Support/GlobalSupportCenters/Pages

1

Developing a Physical Library

The physical library specifies the information required for floor planning, RC estimation and extraction, placement, and routing.

You use the physical library syntax (.plib) to model your physical library.

This chapter includes the following sections:

Creating the Physical Library

This section describes how to name your source file and library, and how to define the units of measure for properties in your library.

Naming the Source File

The recommended file name suffix for physical library source files is .plib.

Example

```
myLib.plib
```

Naming the Physical Library

You specify the name for your physical library in the `phys_library` group, which is always the first executable line in a library source file.

Syntax

```
phys_library(library_name_id) {
          ...
}
```

Use the `comment`, `date`, and `revision` attributes to document your library source file.

Example

```
phys_library(sample) {
   comment : "Copyright Synopsys, Inc. 2002" ;
   date : "1st Jan 2002" ;
   revision : "Revision 2.0.5" ;
}
```

Defining the Units of Measure

Use the `phys_library` group attributes described in Table 1-1 to specify the units of measure for properties such as capacitance and resistance. The unit statements must precede other definitions, such as the technology data, design rules, and macros.

Syntax

```
phys_library (library_name_id) {
   ...
   attribute_name : value_enum ;
   ...
}
```

Example

```
phys_library(sample) {
   capacitance_unit : 1pf ;
   distance_unit : 1um ;
   resistance_unit : 1ohm ;
   ...
}
```

Table 1-1 lists the attribute names and values that you can use to define the units of measure.

Table 1-1 Units of Measure

Property	Attribute name	Legal values
Capacitance	`capacitance_unit`	1pf, 1ff, 10ff, 100ff
Distance	`distance_unit`	1um, 1mm
Resistance	`resistance_unit`	1ohm, 100ohm, 10ohm, 1kohm
Time	`time_unit`	1ns, 100ps, 10ps, 1ps
Voltage	`voltage_unit`	1mV, 10mV, 100mV, 1V
Current	`current_unit`	100uA, 100mA, 1A, 1uA, 10uA, 1mA, 10mA
Power	`power_unit`	1mw
Database distance resolution	`dist_conversion_factor`	Any multiple of 100

2

Defining the Process and Design Parameters

The physical library specifies the information required for floor planning, RC estimation and extraction, placement, and routing.

You use the physical library syntax (.plib) to model your physical library.

This chapter includes the following sections:

Defining the Technology Data

Technology data includes the process and electrical design parameters. Site-array and cell data refer to the technology data; therefore, you must define the layer data before you define site-array and cell data.

Defining the Architecture

You specify the architecture and the layer information in the `resource` group inside the `phys_library` group.

Syntax

```
phys_library(library_name_id) {
   resource(architecture_enum) {
      ...
   }
}
```

architecture

The valid values are `std_cell` and `array`.

Example

```
phys_library(mylib) {
   ...
   resource(std_cell) {
      ...
   }
}
```

Defining the Layers

The layer definition is order dependent. You define the layers starting with the layer closest to the substrate and ending with the layer furthest from the substrate.

Depending on their purpose, the layers can include

- Contact layer
- Overlap layer
- Routing layer
- Device layer

Contact Layer

Contact layers define the contact cuts that enable current to flow between the device and the first routing layer or between any two routing layers; for example, cut01 between poly and metal1, or cut12 between metal1 and metal2. You define the contact layer by using the `contact_layer` attribute inside the `resource` group.

Syntax

```
resource(architecture_enum) {
   contact_layer(layer_name_id)
   ...
}
```

Example

```
contact_layer(cut01) ;
```

Overlap Layer

An overlap layer provides accurate overlap checking of rectilinear blocks.You define the overlap layer by using the `overlap_layer` attribute inside the `resource` group.

Syntax

```
resource(architecture_enum) {
   overlap_layer(layer_name_id)
   ...
}
```

Example

```
resource(std_cell) {
      overlap_layer(mod) ;
      ...
}
```

Routing Layer

You define the routing layer and its properties by using the `routing_layer` group inside the `resource` group.

Syntax

```
resource(architecture_enum) {
   routing_layer(layer_name_id) {
      attribute : value_float ;
      ...
   }
}
```

Example

```
resource(std_cell) ; {
   routing_layer(m1) {  /* metal1 layer definition */
```

```
      cap_per_sq : 3.200e-04 ;
      default_routing_width : 3.200e-01 ;
      res_per_sq : 7.000e-02 ;
      routing_direction : horizontal ;
      pitch : 9.000e-01;
      spacing : 3.200e-01 ;
      cap_multiplier : ;
      shrinkage : ;
      thickness : ;
   }
}
```

Table 2-1 lists the attributes you can use to specify routing layer properties. For more information about any of these attributes, see the *Library Compiler Physical Libraries Reference Manual*.

Note:
All numerical values are floating-point numbers.

Table 2-1 Routing Layer Simple Attributes

Attribute name	Valid values	Property
`default_routing_width`	> 0.0	Minimum metal width allowed on the layer; the default width for regular wiring
`cap_per_sq`	> 0.0	Capacitance per square unit between a layer and a substrate, used to model wire-to-ground capacitance
`res_per_sq`	> 0.0	Resistance per square unit
`coupling_cap`	> 0.0	Coupling capacitance between parallel wires on the same layer
`fringe_cap`	> 0.0	Fringe (sidewall) capacitance per unit length of a routing layer
`routing_direction`	horizontal, vertical	Preferred routing direction
`pitch`	> 0.0	Routing pitch
`spacing`	> 0.0	Default different net spacing (edge-edge) for regular wiring on a layer
`cap_multiplier`	> 0.0	Cap multiplier; accounts for changes in capacitance due to nearby wires

Table 2-1 Routing Layer Simple Attributes (Continued)

Attribute name	Valid values	Property
shrinkage	> 0.0	Shrinkage of metal EffWidth = MetalWidth – Shrinkage
thickness	> 0.0	Thickness
height	>0.0	The distance from the top of the substrate to the bottom of the routing layer
offset	> 0.0	The offset from the placement grid to the routing grid
edgecapacitance	> 0.0	Total peripheral capacitance per unit length of a wire on the routing layer
inductance_per_dist	> 0.0	Inductance per unit length of a routing layer
antenna_area_factor	> 0.0	Antenna effect; to limit the area of wire segments

Specifying Net Spacing

Use the `ranged_spacing` complex attribute to specify the different net spacing for special wiring on the layer. You can also use this attribute to specify the minimum spacing for a particular routing width range of the metal. You can use more than one `ranged_spacing` attribute to specify spacing rules for different ranges.

Each `ranged_spacing` attribute requires floating-point values for the minimum width for the wiring range, the maximum width for the wiring range, and the minimum spacing for the net.

Syntax

```
resource(architecture_enum) {
   routing_layer(layer_name_id) {
      ...
      ranged_spacing(value_float, value_float, value_float) ;
      ...
   }
}
```

Example

```
routing_layer(m1) {
   ...
   ranged_spacing(1.60, 2.40, 1.20) ;
   ...
 }
```

Device Layer

Device layers make up the transistors below the routing layers—for example, the poly layer and the active layer. To define the device layer, use the `device_layer` attribute inside the `resource` group.

Wires are not allowed on device layers. If pins appear in the device layer, you must define vias to permit the router to connect the pins to the first routing layer.

Syntax

```
resource(architecture_enum) {
   device_layer(layer_name_id) ;
   ...
}
```

Example

```
resource(std_cell) {
   device_layer (poly) ;
   ...
}
```

Defining Vias

A via is the routing connection for wires in each pair of connected layers. Vias typically comprise three layers: the two connected layers and the cut layer between the connected layers.

Naming the Via

You define the via name in the `via` group inside the `resource` group.

Syntax

```
resource(architecture_enum) {
   via(via_name_id) {
      ...
   }
}
```

Example

```
resource(std_cell) {
   ...
   via(via23) {
     ...
   }
   ...
}
```

Defining the Via Properties

You define the via properties by using the following attributes inside the `via` group.

- `is_default`
- `top_of_stack_only`
- `resistance`

Syntax

```
via(via_name_id) {
   is_default : Boolean ;
   top_of_the_stack : Boolean ;
   resistance : float ;
   ...
}
```

Example

```
via(via23) {
   is_default : TRUE;
   top_of_stack_only : FALSE;
   resistance : 1.0;
   ...
}
```

Table 2-2 lists the properties you can define with the via attributes.

Table 2-2 Defining Via Properties

Attribute name	Valid values	Property
`is_default`	TRUE, FALSE	Default via for a given layer pair
`top_of_stack_only`	TRUE, FALSE	Use only on top of a via stack
`resistance`	floating-point number	Resistance per contact-cut rectangle

Defining the Geometry for Simple Vias

Define the via geometry (or geometries) by using `via_layer` groups inside a `via` group. Each `via_layer` group defines the via geometry for one layer. Use the name of the layer as the `via_layer` group name.

The layer1 and layer2 layers are the adjacent routing layers, where layer1 is closer to the substrate. The contact layer is the cut layer between layer1 and layer2.

For rectilinear vias, you define the geometry by using more than one rectangle function for the corresponding layer.

Syntax

```
via_layer(layer1_name_id) {
   rectangle(x11_float, y11_float, x21_float, y21_float) ;
                  /* 1 or more rectangles */
}
via_layer(contact_name_id) {
   rectangle(x1c_float, y1c_float, x2c_float, y2c_float) ;
                  /* 1 or more rectangles */
}
via_layer(layer2_name_id) {
   rectangle(x12_float, y12_float, x22_float, y22_float) ;
                  /* 1 or more rectangles */
}
```

where (*x11*, *y11*), (*x21*, *y21*), (*x1c*, *y1c*), (*x2c*, *y2c*), (*x21*, *y12*), and (*x22*, *y22*) are the coordinates of the opposite corners of the rectangle.

Example

```
via(via 45) {
   is_default : TRUE ;
   resistance : 1.5 ;
   via_layer(metal4) {
      rectangle(-0.3, -0.3, 0.3, 0.3) ;
   }
   via_layer(cut45) {
      rectangle(-0.18, -0.18, 0.18, 0.18) ;
   }
   via_layer(meta15) {
      rectangle(-0.27, -0.27, 0.27, 0.27) ;
   }
}
```

Defining the Geometry for Special Vias

Special vias are vias that have

- Fewer than three layers, with one layer being a contact layer
- More than three layers

Vias With Fewer Than Three Layers

To define vias that have fewer than three layers, use the `via_layer` group, as shown below.

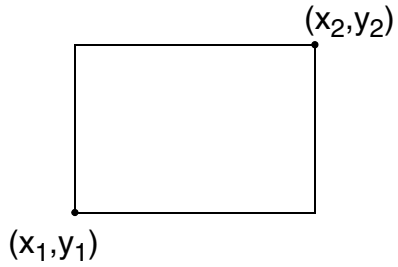
Syntax

```
via_layer(via_name_id) {
   rectangle(x1_float, y1_float, x2_float, y2_float) ;

}
```

where (*x1*, *y1*) and (*x2*, *y2*) are the coordinates (floating-point numbers) for the opposite corners of the rectangle, as shown in Figure 2-1.

Figure 2-1 Coordinates of a Rectangle



Example

```
via_layer(cut23) {
   rectangle(-0.18, -0.18, 0.18, 0.18) ;
}
```

Vias With More Than Three Layers

For vias with more than three layers, use multiple `via_layer` groups. You can have more than one `via_layer` group in your physical library.

Syntax

```
via_layer (via_name_id) {
   rectangle(x1_float, y1_float, x2_float, y2_float) ;

}
```

where (*x1*, *y1*) and (*x2*, *y2*) are the coordinates (floating-point numbers) for the opposite corners of the rectangle.

Example

```
via(via123) {
   resistance : 1.5 ;
   via_layer(met1) {
      rectangle(-0.3. -0.3, 0.3, 0.3):
   }
   via_layer(cut12) {
      rectangle(-0.2. -0.2, 0.2, 0.2):
   }
   via_layer(met2) {
      rectangle(-0.3. -0.3, 0.3, 0.3):
   }
   via_layer(met23) {
      rectangle(-0.2. -0.2, 0.2, 0.2):
   }
   via_layer (met3) {
      rectangle(-0.3, -0.3, 0.3, 0.3) ;
```

```
   }
}
```

Referencing a Foreign Structure

Use the `foreign` group to specify which GDSII structure (model) to use when you place an instance of the via. You also use this group to specify the orientation and the offset with respect to the GDSII structure origin.

Note:
Only one foreign reference is allowed for each via.

Syntax

```
foreign(foreign_structure_name_id) {
   orientation : N  |  E  |  W  |  S  |  FN  |  FE  |  FW  |  FS  ;
   origin(x_float, y_float) ;
}
```

where *x* and *y* represent the offset distance.

Example

```
via(via34) {
   is_default : TRUE ;
   resistance : 2.0e-02 ;
   foreign(via34) {
      orientation : FN ;
      origin(-1, -1) ;
   }
   ...
}
```

Defining the Placement Sites

For each class of cells (such as cores and pads), you must define the available sites for placement. The methodology you use for defining placement sites depends on whether you are working with standard cell technology or gate array technology.

Standard Cell Technology

For standard cell technologies you define the placement sites by defining the site name in the `site` group inside the `resource` group, and by defining the site properties using the following attributes inside the `site` group:

- The `site_class` attribute specifies the site class. Two types of placement sites are supported:
  - Core (core cell placement)

  - Pad (I/O placement)

- The `symmetry` attribute specifies the site symmetry with respect to the x- and y-axes.

  Note:
  If you do not specify the `symmetry` attribute, the site is considered asymmetric.

- The `size` attribute specifies the site size.

Syntax

```
resource(architecture_enum) {
   site(site_name_id) {
      site_class : core | pad ;
      symmetry : x | y | r | xy | rxy ;
      size(x_size_float, y_size_float) ;
   }
}
```

site_name

The name of the library site. Common practice is to describe the function of the site (core or pad) with the site name.

You can assign one of the following values to the `symmetry` attribute:

x

Specifies symmetry about the x-axis

y

Specifies symmetry about the y-axis

r

Specifies symmetry in 90 degree counterclockwise rotation

xy

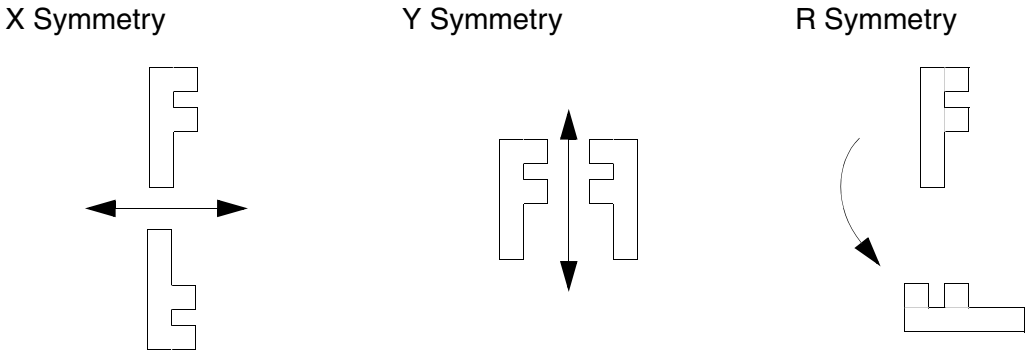
Specifies symmetry about the x-axis and the y-axis

rxy

Specifies symmetry about the x-axis and the y-axis and in 90 degree counterclockwise rotation increments

Figure 2-2 shows the relationship of the symmetry values to the axis.

Figure 2-2 Examples of X, Y, and R Symmetry



Gate Array Technology

Follow these guidelines when working with gate array technologies:

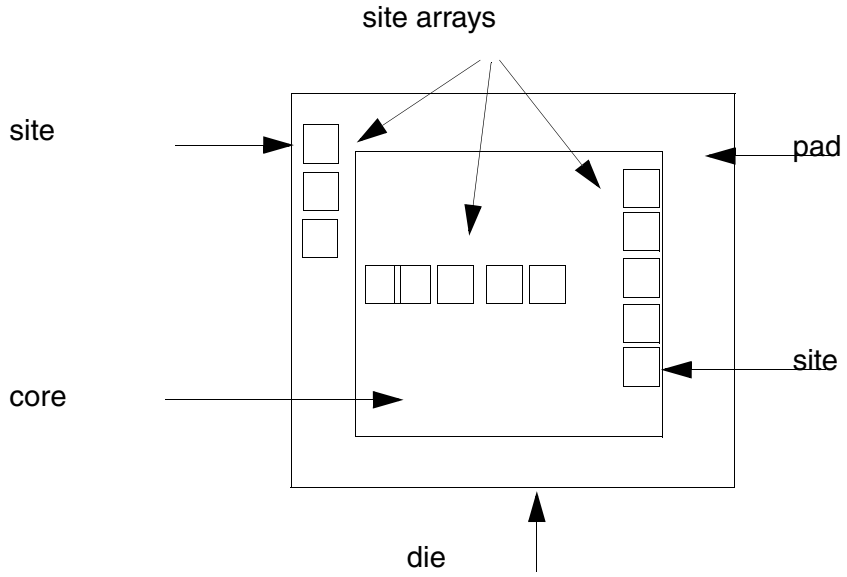
- Define the basic sites for the core and pad in the same way you would for standard cell technologies.
- Use the `array` group to define arrays for the site, the floorplan, and the detail routing grid descriptions. You define the `array` group inside the `resource` group.

Defining the Floorplan Set

A floorplan is an array of sites that allow or disallow the placement of cells. You define a `floorplan` group or multiple `floorplan` groups inside an `array` group.

A floorplan without a name becomes the default floorplan. Subsequently, when no floorplan is specified, the default floorplan is used. Figure 2-3 shows the elements of a floorplan on a die.

Figure 2-3 Elements of a Floorplan



Instantiating the Site Array

You instantiate arrays by using the `site_array` group inside the `floorplan` group. The orientation, availability for placement, origin, and the array pattern (that is, the number of rows and columns, as well as the row spacing and column spacing) are all defined in the `site_array` group.

Syntax

```
site(site_name_id) {
   stateless : pad | core;
   symmetry : x | y | r | xy | rxy ;
   size(x_size_float, y_size_float) ;
}
array(array_name_id) {
   ...
   floorplan(floorplan_name_id) {
      site_array(site_name_id) {
         orientation : N | E | W | S | FN | FE | FW | FS ;
         placement_rule : regular | can_place |
                          cannot_place ;
         origin(x_float, y_float) ;
         iterate(num_x_int, num_y_int,
               space_x_float, space_y_float) ;
      }
   }
}
```

Table 2-3 shows the values and description for each of the attributes you use to define placement sites.

Table 2-3 Placement Site Definitions

Attribute	Valid values	Description
site_class	pad	I/O cell placement site
	core	Core cell placement site
symmetry	x, y, r, xy, rxy	Symmetry
	width, height	Site dimensions
orientation	N, E, W, S, FN, FE, FW, FS	Orientation
placement_rule	can_place	Site array available for floorplan
	cannot_place	Site array not available for floorplan
	regular	Placement grid
origin	x, y	Coordinate of the origin of site array
iterate	num_x	Number of columns in the site array
	num_y	Number of rows in the site array
	space_x	Column spacing (float)
	space_y	Row spacing (float)

Example

```
site(core) {
   site_class : core ;
   symmetry : x ;
   size (1, 10) ;
}
array(samplearray) {
   ...
   floorplan() {  /* default floorplan */
      site_array(core) {  /* Core cells placement */
         orientation :  N ;
         placement_rule : can_place;   /* available for placement */
         origin(0, 0) ;
         iterate(2, 4, 1.5, 0) ;  /* site_array has 2 sites in x */
```

```
                        /*direction spaced 1.5 um apart, 4 */
                        /*sites in y direction, spaced */
                        /*1.5 um apart */
      }
   }
}
```

Defining the Global Cell Grid

You define the global cell grid overlaying the array by using the `routing_grid` attribute inside the `array` group. The router uses this grid during global routing.

Syntax

```
array(array_name_id) {
   routing_grid() {
      routing_direction : horizontal | vertical ;
      grid_pattern (start_float, grids_integer, spacing_float) ;
}
```

where

start

A floating-point number representing the starting-point coordinate

grids

An integer number representing the number of grids in the x and y directions

spacing

A floating-point number representing the spacing between the grids in the x and y directions

Example

```
array(samplearray) {
   routing_grid(0, 3, 1, 0, 3, 1) ;
      routing_direction(horizontal) ;
      grid_pattern(, ,) ;
      ...
}
```

Defining the Detailed Routing Grid

You specify the routing track grid for the gate array by using the `tracks` group inside the `array` group. In the `tracks` group, you specify the track pattern, the track direction, and the layers available for the associated tracks.

Note:

Define one `tracks` group for horizontal routing and one for vertical routing.

Syntax

```
array(array_name_id) {
   ...
   tracks() {
```

```
        layers : "layer_1", "layer_2", ..."layer_n" ;
        routing_direction : vertical  |  horizontal ;
        track_pattern(start_point_float, num_of_tracks_float,
                  space_between_tracks_float) ;
    }
}
```

where

start_point

A floating-point number representing the starting-point coordinate

num_of_tracks

A floating-point number representing the number of parallel tracks

space_between

A floating-point number representing the spacing between the tracks

Example

```
phys_library(example) {
   ...
   resource(array) {  /* gate array technology */
      ...
      array(samplearray) {
         ...
         tracks() {
            layers : "m1", "m3" ;
            routing_direction : horizontal ;
            track_pattern(1, 50, 10) ;
            /* 50 horizontal tracks 10 microns apart  */
         } /* end tracks */
         tracks() {
            layers :  "m1",  "m2" ;
            routing_direction : vertical ;
            track_pattern(1, 50, 10) ;
            /* 50 vertical tracks 10 microns apart  */
         }/* end tracks */
      } /* end array */
   } /* end resource */
   ...
} /* end phys_library */
```

3

Defining the Design Rules

Specify design rules for the technology, such as minimum spacing and width, by using the `topological_design_rules` group.

This chapter includes the following sections:

Defining the Design Rules

The following sections describe how you define the design rules for physical libraries.

Defining Minimum Via Spacing Rules in the Same Net

The design rule checker requires the value for the edge-to-edge minimum spacing between vias.

Use the `contact_min_spacing` attribute for defining the minimum spacing between contacts in different nets. This attribute requires the name of the two contact layers and the spacing distance. To specify the minimum spacing between the same contact, use the same contact layer name twice.

Syntax

```
topological_design_rules() {
   contact_min_spacing(contact_layer1id,
         contact_layer2id, spacingfloat) ;
   ...

}
```

Example

```
phys_library(sample) {
   ...
   topological_design_rules() {
      ...
      contact_min_spacing(cut01, cut12, 1) ;
      ...
   }
   ...
}
```

Defining Same-Net Minimum Wire Spacing

You can specify the minimum wire spacing between contacts in the same net by using the `same_net_min_spacing` attribute. To specify the minimum spacing between the same contact, use the same contact layer name twice.

Syntax

```
topological_design_rules() {
   same_net_min_spacing(layer1_nameid, layer2_nameid,
                  spacingfloat, ...) ;
      ...
}
```

Example

```
topological_design_rules() {
   same_net_min_spacing(m1, m1, 0.4, ...) ;
   same_net_min_spacing(m3, m3, 0.4, ...) ;
   ...
}
```

Defining Same-Net Stacking Rules

You can specify stacking for vias that share the same routing layer by setting the `is_stack` parameter in the `same_net_min_spacing` attribute to `TRUE`.

Syntax

```
topological_design_rules() {
   same_net_min_spacing(layer1_name_id, layer2_name_id,
                  spacing_float, is_stack_Boolean) ;
    ...
}
```

Example

```
topological_design_rules() {
   same_net_min_spacing(m1, m1, 0.4, TRUE) ;
   same_net_min_spacing(m3, m3, 0.4, FALSE) ;
   ...
}
```

Defining Nondefault Rules for Wiring

For all regular wiring, you define the default rules by using either the `layer` group or the `via` group in the `resource` group. You define the nondefault rules for wiring by using the `wire_rule` group in the `topological_design_rules` group as shown here:

```
phys_library(sample) {
   ...
   topological_design_rules() {
      ...
      wire_rule(rule1) {
         via(non_default_via12) {
            ...
         }
      }
   }
}
```

You define the width, different net minimum spacing (edge-to-edge), and the wire extension by using the `layer_rule` group. The width and spacing specifications override the default values defined in the `routing_layer` group. If you do not specify the extension, the tool applies a default extension. The value of the default extension is half the routing width for the layer used.

```
phys_library(sample) {
    ...
    topological_design_rules() {
       ...
       layer_rule(metal1) {
          /* non default regular wiring rules for metal1 */
          wire_width :  0.4 ;  /* default is 0.32 */
          min_spacing : 0.4 ;  /* default is 0.32 */
          wire_extension : 0.25 ;   /* default is 0.4/2 */
       } /*end layer rule */
    }
}
```

Use the `via` group in the `wire_rule` group to define nondefault vias associated with the routing layers. This via group is similar to the `via` group in the `resource` group except that the `is_default` attribute is absent. For regular wiring, the tool uses the via defined in the `wire_rule` group instead of the default via defined in the `resource` group whenever the wire width matches the width specified in the `via` or `layer` group.

```
phys_library(sample) {
   ...
   topological_design_rules() {
      ...
      wire_rule(rule1) {
         via(non_default_via12) {
            ...
         }
      }
   }
}
```

For nondefault regular wiring, you define the via and routing layer spacing and the stacking rules by using the `same_net_min_spacing` attribute inside the `wire_rule` group. This attribute overrides the default values in the `same_net_min_spacing` attribute inside the `topological_design_rules` group.

```
phys_library(sample) {
   ...
   topological_design_rules() {
   ...
      wire_rule(rule1) {
         same_net_min_spacing(m1, m1, 0.32, FALSE) ;
         same_net_min_spacing(m2, m2, 0.4, FALSE) ;
         same_net_min_spacing(cut01, cut01, 0.36, FALSE) ;
         same_net_min_spacing(cut12, cut12, 0.36, FALSE) ;
      } /* end wire rule */
```

```
   } /* end design rules */
} /* end phys_library */
```

Use the `vias` attribute in the `via_rule` group to specify a list of vias. The router selects the first via that satisfies the design rules.

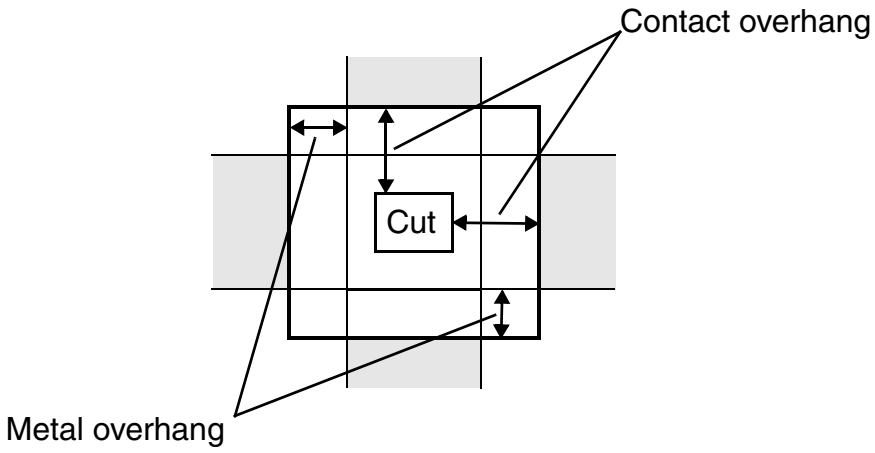
Defining Rules for Selecting Vias for Special Wiring

The `via_rule` group inside a `topological_design_rules` group defines vias used at the intersection of special wires in the same net.

You can specify multiple `via_rule` groups for a given layer pair. The rule that governs the selection of a `via_rule` group is the routing wire width range. When the width of a special wire is within the range specified, then the via rule is selected. When no via rule applies, then the default via rule is applied. The default via rule is created when you omit the routing wire width specification.

You also specify contact overhang and metal overhang, in both the horizontal and vertical directions, in the `via_rule` group. Contact overhang is the minimum amount of metal (wire) between the contact and the via edge. Metal overhang is at the edges of wire intersection. Figure 3-1 shows these relationships.

Figure 3-1 Contact Overhang and Metal Overhang



Syntax

```
topological_design_rules() {
   ...
   via_rule(via_rule_name_id) {
      vias : list_of_vias_id ;
      routing_layer_rule(routing_layer_name_id) {
      /* one for each layer associated with the via; */
```

```
      /* normally 2. */
      routing_direction : value_enum ;
      /* direction of the overhang */
      contact_overhang : value_float ;
      metal_overhang : value_float ;
         min_wire_width : value_float ;
         max_wire_width : value_float ;
      }
   }
}
```

Example

```
topological_design_rules() {
   ...
   via_rule(default_rule_for_m1_m2) {
   /* default via rule for the metal1, metal2 pair; */
   /* no wire width range is specified  */
      vias :  "via12, via23" ;
      /* select via12 or via23 - whichever satisfies */
      /* the design rules*/
      routing_layer_rule(metal1) {
         routing_direction : horizontal ;
         contact_overhang :  0.1 ;
         metal_overhang :  0 ;
      }
      routing_layer_rule(metal2) {
         routing_direction : vertical ;
         contact_overhang :  0.1 ;
         metal_overhang :  0 ;
      }
   }
   ...
}
```

Defining Rules for Generating Vias for Special Wiring

Use the `via_rule_generate` group to specify the rules for generating vias used at the intersection of special wires in the same net. You define this group inside the `topological_design_rules` group. You can specify multiple `via_rule_generate` groups for a given layer pair.

The rule that governs the selection of a `via_rule` group is the routing wire width range. When the width of the special wire is within the range specified, then the via rule is selected. When no via rule applies, then the default via rule is applied. The default via rule is created when you omit the routing wire width specification.

Use the `vias` attribute in the `via_rule_generate` group to specify a list of vias. The router selects the first via that satisfies DRC. You also specify contact overhang and metal overhang, in both the horizontal and vertical directions, in the `via_rule_generate` group. Contact overhang is the minimum amount of metal (wire) between the contact and the via edge. Metal overhang is at the edges of wire intersection.

You specify the contact layer geometry generation formula in the `contact_formula` group inside the `via_rule_generate` group. The number of contact cuts in the generated array is determined by the contact spacing, contact-cut geometry, and the overhang (both contact and metal).

Syntax

```
topological_design_rules() {
   ...
   via_rule_generate(via_rule_name_id) {
      routing_layer_formula(routing_layer_name_id) {
         /* one for each layer associated with the via */
         /* normally 2 */
         routing_direction : value_enum ;
         /* direction of the overhang */
         contact_overhang :  value_float ;
         metal_overhang : value_float;
         min_wire_width : value_float ;
         max_wire_width : value_float ;
      }
      contact_formula(contact_layer_name) {
         rectangle(x1_float, y1_float, x2_float, y2_float) ;
         /* specify more than 1 rectangle for */
         /* rectilinear vias */
         contact_spacing(x_spacing_float, y_spacing_float)
         resistance : value_float
      }
}
```

Example

```
phys_library(sample) {
   ...
   resource(std_cell) {  /* standard cell technology */
      ...
   } /* end resource */
   topological_design_rules() {  /* design rules */
      same_net_min_spacing(m1, m1, 0.32, FALSE) ;
      /* minimum spacing required between 2 metal1 layers in the same net */
      same_net_min_spacing(m2, m2, 0.4, FALSE) ;
      /* minimum spacing required between 2 metal2 layers in the same net */
      same_net_min_spacing(m3, m3, 0.4, FALSE) ;
      /* minimum spacing required between 2 metal3 layers in the same net */
      same_net_min_spacing(cut01, cut01, 0.36, FALSE) ;
      /* minimum spacing required between 2 contact cut01 layers in the same net */
      same_net_min_spacing(cut12, cut12, 0.36, FALSE) ;
      /* minimum spacing required between 2 contact cut12 layers in the same net */
      same_net_min_spacing(cut23, cut23, 0.36, FALSE) ;
```

```
        /* minimum spacing required between 2 contact cut23 layers in the same net */
        /* via generation rules */
        via_rule_generate(default_rule_for_m1_m2) {
            routing_layer_formula(metal1) {
                routing_direction : horizontal ;
                contact_overhang :  0.1 ;
                metal_overhang :  0.0 ;
            }
            routing_layer_rule(metal2) {
                routing_direction : vertical ;
                contact_overhang :  0.1 ;
                metal_overhang :  0 ;
            }
            contact_formula(cut12) { /* rule for generating contact cut array */
                rectangle(-0.2, -0.2, 0.2, 0.2) ;  /* cut shape */
                contact_spacing(0.8, 0.8) ;   /* center-to-center spacing */
                resistance : 1.0 ;  /* cut resistance */
            }
        } /* end via_rule_generate */
        via_rule_generate(default_rule_for_m2_m3) {
            routing_layer_formula(metal2) {
                routing_direction : vertical ;
                contact_overhang :  0.1 ;
                metal_overhang :  0.0 ;
            }
            routing_layer_rule(metal3) {
                routing_direction : horizontal ;
                contact_overhang :  0.1 ;
                metal_overhang :  0 ;
            }
            contact_formula(cut23) {  /* rule for generating contact cut array */
                rectangle(-0.2, -0.2, 0.2, 0.2) ;  /* cut shape */
                contact_spacing(0.8, 0.8) ;   /* center-to-center spacing */
                resistance : 1.0 ;  /* cut resistance */
            }
        } /* end via_rule_generate */
    } /* end design rules */
    macro(and2) {
        ...
    } /* end macro */
} /* end phys_library */
```

Defining Rules for Generating Vias for Default Wiring

Default signal wires are routing wires that have the minimal (default) routing width.

During compilation, Library Compiler looks for a via definition (in a `via` group) on every adjacent pair of routing layers in each `wire_rule` group. If it cannot find a via definition, Library Compiler looks for a predefined via generation rule (in a `via_rule_generate` group) that fits the routing width range for the two layers. If it finds a predefined rule, Library Compiler creates the via definition according to the rule. If neither a predefined via definition nor a via generation rule is found, the Library Compiler generates the 12 compliant vias shown in Figure 3-2. A compliant via is a via definition in which the upper metal layer, cut layer, and lower metal layer are in vertical alignment.

Library Compiler requires the following information to generate the default vias for default signal wires:

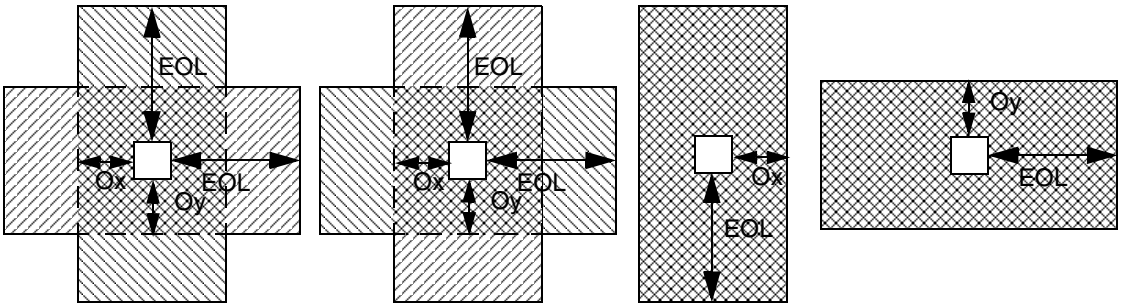
- The horizontal overhang and vertical overhang (Ox and Oy) on both the top and bottom layers
- The end-of-line (EOL) extension on the top and bottom layers
- The minimal area design rule (MAR) for the lower metal area

Using the above information, Library Compiler generates:

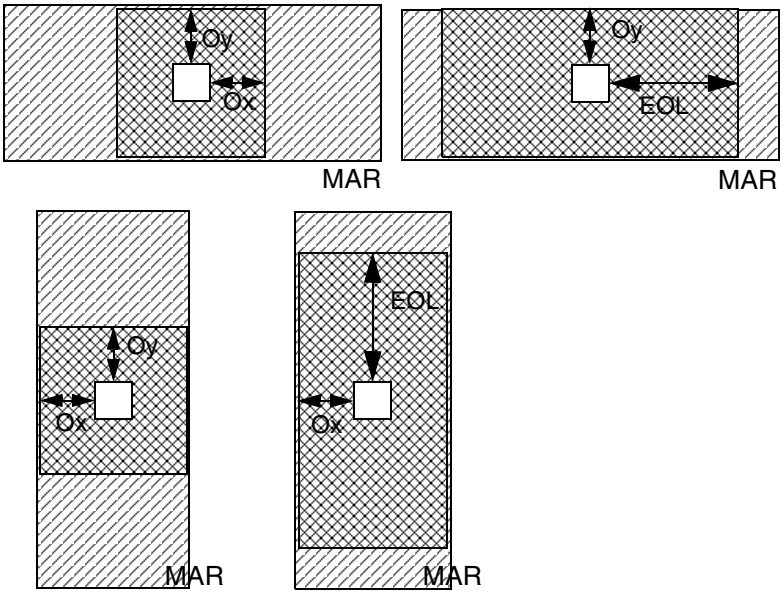
- One via with minimum metal extension (overhang) for both metal layers.
- Seven other vias with all possible combinations and orientations of end of line and minimum area extensions in vertical and horizontal directions, including all possible preferred and nonpreferred direction orientations.

Figure 3-2 Default Generated Vias

VIAs with end-of-line (EOL) extension:



VIAs with minimal-area-rule (MAR) on bottom layer:



Syntax

```
physical_library () {
   resource () {
      ...
      routing_layer () {
         ...
         min_area : float ; /* minimal area rule (MAR) */
      }
   }
   topological_design_rules() {
      default_via_generate ( name_id ) {
      via_routing_layer(layer_name_id) {
         overhang (float, float); /* Ox, Oy */
         end_of_line_overhang : float ; /* EOL */
```

```
      }
      via_routing_layer(layer_name_id) {
         overhang (float, float);
         end_of_line_overhang : float ;
      }
      via_contact_layer(layer_name) {
         rectangle (float, float, float, float) ;
         resistance : float ;
      }
   }
}
```

Example

```
physical_library () {
   resource (Cstd_cell") {
      ...
      routing_layer ("MET1") {
         ...
         min_area : 1.0 ; /* minimal area rule (MAR) */
      }
   topological_design_rules() {
      default_via_generate ( via1array ) {
      via_routing_layer("MET1C) {
         overhang (1.0, 1.0); /* Ox, Oy */
         end_of_line_overhang : 1.0 ; /* EOL */
      }
      via_routing_layer("MET2") {
         overhang (1.0, 1.0);
         end_of_line_overhang : 1.0 ;
      }
      via_contact_layer("VIA12") {
         rectangle (1.0, 1.0, 1.0, 1.0) ;
         resistance : 1.0 ;
      }
   }
}
```

Note:

Library Compiler preserves all user-defined via definitions on the contact layer.

Defining the Generated Via Size

Generated vias are a multiple of the minimum feature size. The lithographic grid determines the minimum feature size for the technology.

Syntax

```
min_generated_via_size(x_size_float, y_size_float) ;
```

4

Defining Cells

You use the `macro` group inside the `phys_library` group to describe macro-level information about the cells, such as symmetry, size, and origin, as well as pin information (geometry, position, and so forth).

This chapter includes the following sections:

Defining Cell Characteristics

The following sections describe how you define cell characteristics in a macro group.

Naming the Cell

For each cell in a library, define one `macro` group. Each macro () construct corresponds to a cell () construct in the technology library (.lib). Use the same name to identify the corresponding `macro` and `cell` groups.

Syntax

```
macro(cell_name_id) {
          ...
}
```

Example

```
phys_library(sample) {
      ...
   resource(std_cell) {
      ...
   }
   topological_design_rules() {
      ...
   }
   macro (and2) {
      ...
   }
      ...
}
```

Defining the Cell Type

You define the cell type by using the `cell_type` attribute in the `macro` group.

Syntax

```
macro(cell_name_id) {
   cell_type : value_enum ;
   ...
}
```

For a list of valid cell types, see the *Library Compiler Physical Libraries Reference Manual*.

Example

```
macro(nand4) {
   cell_type : core ;
      ...
```

```
}
```

Defining the Source

You define the cell source by using the `source` attribute inside the `macro` group.

Syntax

```
macro(cell_name_id) {
   ...
   source : value_enum ;
   ...
}
```

Valid values are `user` (specifies a regular cell), `generate` (specifies a parametric cell), and `block` (specifies a block cell).

Example

```
macro(nand4) {
   ...
   source : user ;
   ...
}
```

Specifying Electrically and Logically Equivalent Cells

You can specify electrical or logical equivalency between defined cells with the `eq_cell` and the `leq_cell` attributes inside the `macro` group.

Electrically equivalent cells should have the same functionality, pin order, and electrical characteristics (such as timing, power, and so forth). The placement tools use this information to select the correct cell from a set of electrically equivalent cells.

Logically equivalent cells should have the same functionality and pin order, but they do not need to have the same electrical characteristics.

Syntax

```
macro(cell_name_id) {
   eq_cell : eq_cell_name_id ;
   leq_cell : leq_cell_name_id ;
   ...
}
```

Example

```
phys_library(sample) {
   ...
   macro(and2a) {
      cell_type : core ;
```

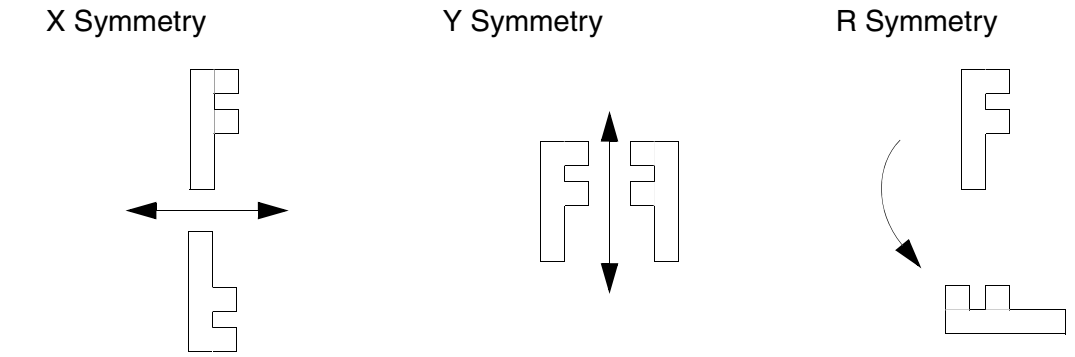
```
      leq_cell : and2x2 ;
      ...
   }
   macro(and2b) {
      cell_type : core ;
      eq_cell : and2a ;
      leq_cell : and2x2 ;
      ...
   }
   ...
}
```

Specifying Cell Symmetry

You specify cell symmetry by using the `symmetry` attribute inside the `macro` group. The cell symmetry must match the associated site symmetry. If you do not specify the `symmetry` attribute, the cell is considered asymmetric.

Figure 4-1 shows examples of X, Y, and R symmetry.

Figure 4-1 Examples of X, Y and R Symmetry



Syntax

```
macro(cell_name_id) {
          symmetry : value_enum ;
          ...
}
```

where *value* is one of the following:

x

Specifies symmetry about the x-axis

y

Specifies symmetry about the y-axis

r

Specifies symmetry in 90 degree counterclockwise rotation

xy

Specifies symmetry about the x-axis and the y-axis

rxy

Specifies symmetry about the x-axis and the y-axis and in 90 degree counterclockwise rotation increments

Example

```
macro(inverter) {
   cell_type : core ;
   in_site : core ;
   symmetry : x ;
   ...
}
```

Specifying the Cell Origin

You specify the cell origin by using the `origin` attribute inside the `macro` group. You define the other coordinates (such as sites, ports, and obstructions) with respect to the values you specify in the `origin` attribute.

Syntax

```
macro(cell_name_id) {
   ...
   origin(x_float, y_float) ;
   ...
}
```

Example

```
macro(mux21) {
   cell_type : core ;
   symmetry : xy ;
   in_site : core ;
   origin(0.0, 0.0) ;
   ...
}
```

Specifying the Cell Size

You specify the cell size using the `size` attribute inside the `macro` group. The values you specify describe the minimum bounding rectangle for the cell. Set this to a multiple of the placement grid. For standard cells, the height should be equal to the associated site height and the width should be a multiple of the site width.

Syntax

```
macro(cell_name_id) {
   ...
   size(x_size_float, y_size_float) ;
   ...
}
```

Example

```
macro(mux21) {
   cell_type : core ;
   symmetry : xy ;
   in_site : core ;
   origin(0.0, 0.0) ;
   size(5, 10) ;
      ...
}
```

Associating the Placement Sites

For standard cells, use a row of a previously defined site. For gate array architecture, use a site array.

Standard Cells

Use the `in_site` attribute inside the `macro` group to specify the site associated with the cell. The cell class and symmetry must match the site class and symmetry.

Syntax

```
macro(cell_name_id) {
   ...
   in_site : name_id) {
}
```

Example

```
phys_library(sample) {
   ...
   resource(std_cell) {  /* standard cell technology */
      ...
      site(core) {
         site_class : core ;
         symmetry :  x ;
         size(1, 2) ;
      }
   }
   macro(and2a) {
      cell_type : core ;
      in_site : core ;
      ...
   }
```

```
        macro(and2b) {
           cell_type : core ;
           eq_cell : "and2a" ;
           in_site : core ;
           ...
        }
     }
```

Gate Arrays

To specify the site array associated with the cell, use the `site_array` group inside the `macro` group. The site requirement is defined as a subset of an existing site array (defined in the technology section). The cell class and symmetry must match the site class and symmetry.

Syntax

```
macro(cell_name_id) {
   site_array(site_name_id) ;
   ...
}
```

Example

```
phys_library(sample) {
    ...
    resource(array) {  /* gate array technology */
        ...
        array(corearray) {
            ...
            floorplan() {  /* default floorplan */
                site_array(core_site) {  /* Core cells placement */
                    ...
                }
            }
        }
    }
    ...
    macro(and2a) {
        ...
        site_array(core) {  /* associating the core array site */
            ...
        }
    }
    ...
}
```

Defining the Pins in a Cell

To specify pin properties such as direction and geometry, use the `pin` group inside the `macro` group.

Defining the Pin Attributes

You must define one `pin` group for each pin in the cell.

Syntax

```
macro(cell_name_id) {
   ...
   pin(pin_name_id) {
      ...
   }
}
```

For information about the attributes and groups in the `pin` group, see the *Library Compiler Physical Libraries Reference Manual*.

Example

```
macro(and2) {
   pin(A) {
      ...
   }
   ...
}
```

Referencing a Foreign Structure

Use the foreign group to specify which GDSII structure (model) to use when you place an instance of the cell and this pin. You also use this group to specify the orientation and the offset with respect to the GDSII structure origin.

Syntax

```
pin(pin_name_id) {
   ...
   foreign(foreign_structure_name_id) {
      orientation : N  |  E  |  W  |  S  |  FN  |  FE |  FW  |  FS ;
      origin(x_float, y_float) ;
   }
   ...
}
```

For more information about the foreign group attributes, see the *Library Compiler Physical Libraries Reference Manual*.

Example

```
pin(A) {
   ...
   foreign(XOR2X1) {
      orientation : FN ;
      origin(1, 1) ;
   }
   ...
```

```
}
```

Defining the Port Geometry for the Pin

You specify the geometry by using the `port` group inside the `pin` group. You can specify the shape as path or polygon or rectangle functions, or you can specify an array of paths, polygons, or rectangles.

Syntax

```
pin() {
   port(pin_name) {
   geometry(layer_name_id) {
      path(width, x1, y1, ..., xi, yi, ..., xn, yn) ;
                              /* n >= 2*/
      polygon(x1, y1, x2, y2, x3, y3, x4, y4,  ..., xi,
            yi, ..., xn, yn) ; /* n >= 3*/
      rectangle(x1, y1, x2, y2) ;
      path_iterate(width, num_x, num_y, space_x,
            space_y, x1, y1, ..., xi,
            yi, ..., xn, yn) ;/* n >= 2*/
      polygon_iterate(num_x, num_y, space_x,
            space_y, x1, y1, x2, y2,
            x3, y3, x4, y4, ..., xi, yi,
            ..., xn, yn) ;    /* n >= 3*/
      rectangle_iterate(num_x, num_y, space_x, space_y,
            x1, y1, x2, y2) ;
   }
   ...
}
```

For more information about the `geometry` group attributes, see the *Library Compiler Physical Libraries Reference Manual*.

Example

```
port() {
   geometry(cut01) {
      rectangle(0.1, 0,1, 0,9, 0,9) ;
   }
   geometry(m1) {
      rectangle(0, 0, 1, 1) ;
      rectangle(2, 0, 3, 1) ;
      rectangle(0, 2, 1, 3) ;
      rectangle(2, 2, 3, 3) ;
   }
}
```

The example can also be written as

```
port() {
   geometry(cut01) {
      rectangle(0.1, 0,1, 0,9, 0,9) ;
   }
```

```
   geometry(m1) {
      rectangle_iterate(2, 2, 1, 1, 0, 0, 1, 1) ;
   }
}
```

Placing the Vias Inside the Port

You use the `via` and `via_iterate` attributes to specify the vias that are placed inside the port. You define these attributes inside the `port` group.

The `via` attribute instantiates a via defined in the `resource` group at the given coordinates.

The `via_iterate` attribute instantiates an array of vias in a particular pattern.

Syntax

```
port() {
   via(via_name_id, x_float, y_float) ;
   via_iterate(num_x_float, num_y_float,
         space_x_float, space_y_float, via_name_id,
         x_float, y_float) ;
```

Example

```
port() {
  ...
  via(via12, 0, 0) ;
  via(via12, 0, 100) ;
  via(via12, 100, 0) ;
  via(via12, 100, 100) ;
}
```

The example can also be written as

```
port() {
   ...
   via_iterate(2, 2, 100, 100, via12, 0, 0) ;
}
```

Defining Obstructions on the Cell

To specify the obstructions and blockages on the cell, use the `obs` group inside the `macro` group.

Syntax

```
macro(cell_name_id) {
   ...
   obs() {
      ...
```

```
   }
   ...
}
```

Defining the Obstruction Geometry

To specify the geometry of the obstruction on the cell, use the `geometry` group inside the `obs` group. You can describe the shape of the obstruction as a path, rectangle, or polygon function. Alternatively, you can specify an array of paths, rectangles, or polygons.

Syntax

```
geometry(layer_name_id) {
   path(width, x1, y1, ..., xi, yi, ..., xn, yn) ;
                              /* n >= 2 */
   polygon(x1, y1, x2, y2, x3, y3,..., xi, yi,
         ..., xn, yn) ;       /* n >= 3 */
   rectangle(x1, y1, x2, y2) ;
   path_iterate(width, num_x, num_y,
            space_x, space_y, x1, y1,
            ..., xi, yi, ..., xn, yn) ;  /* n >= 1 */
   polygon_iterate(num_x, num_y, space_x, space_y,
               x1, y1, x2, y2, x3, y3,
               x4, y4, ..., xi, yi, ..., xn, yn) ;
                              /* n >= 3 */
   rectangle_iterate(num_x, num_y, space_x, space_y, x1,
                        y1, x2, y2) ;
}
```

For more information about the `geometry` group attributes, see the *Library Compiler Physical Libraries Reference Manual*.

Example

```
obs() {
   geometry(m1) {
      rectangle(0, 0, 1, 1) ;
      rectangle(2, 0, 3, 1) ;
      rectangle(0, 2, 1, 3) ;
      rectangle(2, 2, 3, 3) ;
      path(0.44, 0.53, 2.32, 3.45, 2.32) ;
   }
}
```

The example can also be written as

```
obs() {
   geometry(m1) {
      rectangle_iterate(2, 2, 1, 1, 0, 0, 1, 1) ;
   }
}
```

Placing the Vias Inside an Obstruction

You use the `via` and `via_iterate` attributes to specify the vias that are placed inside an obstruction on a cell. You define these attributes inside the `obs` group.

The `via` attribute instantiates a via defined in the `resource` group at the given coordinates.

The `via_iterate` attribute instantiates an array of vias in a particular pattern.

Syntax

```
obs() {
   ...
   via(via_name_id, x_float, y_float) ;
   via_iterate(num_x_float, num_y_float,
            space_x_float, space_y_float, via_name_id,
            x_float, y_float) ;
}
```

Example

```
obs() {
  ...
  via(via12, 0, 0) ;
  via(via12, 0, 100) ;
  via(via12, 100, 0) ;
  via(via12, 100, 100) ;
}
```

The example can also be written as

Example

```
obs() {
   ...
   via_iterate(2, 2, 100, 100, via12, 0, 0) ;
}
```

5

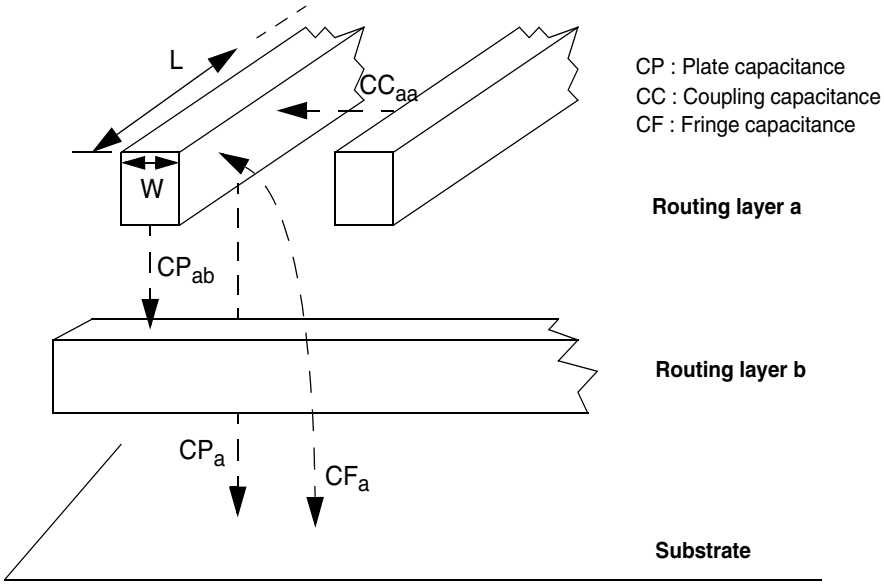
Parasitic RC Estimation in the Physical Library

This chapter includes the following sections:

Modeling Parasitic RC Estimation

Figure 5-1 provides an overview of the measures used in the parasitic RC estimation model.

Figure 5-1 Parasitic RC Estimation Model



The following sections provide information about the variables and equations you use to model parasitic RC estimation.

Variables Used in Parasitic RC Estimation

The following sections list and describe the routing layer and routing wire variables you need to define in the RC estimation model.

Variables for Routing Layers

Define the following set of variables for each `routing_layer` group in your physical library.

Variable	Description
res_per_sq	Resistance per square of a res_per_sq routing layer.
cap_per_sq	Substrate capacitance per cap_per_sq square of a poly or metal layer (CP layer).
coupling_cap	Coupling capacitance per unit length between parallel wires on the same layer (CC layer).
fringe_cap	Fringe (sidewall) capacitance per unit length of a routing layer (CF layer).
edgecapacitance	Total fringe capacitance per unit length of routing layer. Specifies capacitance due to fringe, overlapping, and coupling effect.
inductance_per_dist	Inductance per unit length of a routing layer.
shrinkage	Distance that wires on the layer will shrink or expand on each side from the design to the fabricated chip. Note that negative numbers indicate expansion and positive number indicate shrinkage.
default_routing_width	Default routing width for wires on the layer.
height	Distance from the top of the substrate to the bottom of the routing layer.
thickness	Thickness of the routing layer.
plate_cap	Capacitance per unit area when the first layer overlaps the second layer. This function specifies an array of values indexed by routing layer order (CP layer, layer).

Variables for Estimated Routing Wire Model

Define the following set of variables for each `routing_wire_model` group in your physical library. Each `routing_wire_model` group represents a statistics-based design-specific estimation of interconnect topology.

overlap_wire_ratio

Percentage of the wiring on the first layer that overlaps the second layer. This function specifies all overlap_wire_ratio values in an n*(n-1) sized array, where n is the number of routing layers. For example, the overlap_wire_ratio values for the first routing layer (routing layer 1) are specified in overlap_wire_ratio[0] to overlap_wire_ratio[n-2]. The values for routing layer 2 are specified in overlap_wire_ratio[n-1] to overlap_wire_ratio[2(n-1)].

adjacent_wire_ratio

Percentage of wiring on the layer that runs adjacent to and has minimum spacing from wiring on the same layer. This function specifies percentage values of adjacent wiring for all routing layers. For example, two parallel adjacent wires with the same length would have an adjacent_wire_ratio of 50 percent.

wire_ratio_x

Percentage of total wiring in the horizontal direction that you estimate will be on each layer. The function carries an array of floating-point numbers, following the order of routing layers. That is, there will be three floating-point numbers in the array if there are three routing layers. These numbers should add up to 1.00.

wire_ratio_y

Percentage of total wiring in the vertical direction that you estimate will be on each layer. The function carries an array of floating-point numbers, following the order of routing layers. That is, there will be three floating point numbers in the array if there are three routing layers. And these numbers should add up to 1.00.

wire_length_x, wire_length_y

Estimated wire lengths in horizontal and vertical direction for a net.

Equations for Parasitic RC Estimation

Parasitic calculation is based on your estimates of routing topology prior to detailed routing. The following sections describe how to determine those estimates.

Capacitance per Unit Length for a Layer

Use the following equations to estimate capacitance per unit length for a given layer.

$$cap\_per\_dist_{layer} = W * cap\_per\_area_{layer} + fringe\_cap_{layer} + coupling\_cap\_per\_dist_{layer}$$

where

$$W = (default\_wire\_width \mid actual\_wire\_width) - shrinkage$$

$$cap\_per\_area_{layer} = 1 - SUM\_overlap\_wire\_ratio\_under_{layer} * cap\_per\_sq_{layer} + SUM_{i=other\_layer}[overlap\_wire\_ratio_{j,layer}] * plate\_cap_{layer,i}$$

where

$$SUM\_overlap\_wire\_ratio\_under_{layer} = SUM_{j=layer\_underneath}[overlap\_wire\_ratio_{j,layer}]$$

Note:

The above equation represents the sum of all the `overlap_wire_ratio` values between the current layer and each layer underneath the current layer.

$$coupling\_cap\_per\_dist_{layer} = 2 * adjacent\_wire\_ratio_{layer} * coupling\_cap_{layer}$$

Resistance and Capacitance for Each Routing Direction

Use the following equations to estimate capacitance and resistance values based on orientational routing wire ratios.

```
capacitance x = cap_per_dist x * wire_length_x
capacitance y = cap_per_dist y * wire_length_y


resistance x = res_per_sq x * wire_length x / width x
resistance y = res_per_sq y * wire_length y / width y
```

where

```
cap_per_dist x = SUM[wire_ratio_x layer * cap_per_dist layer]
cap_per_dist y = SUM[wire_ratio_y layer * cap_per_dist layer]


res_per_sq x = SUM[ wire_ratio_x layer * res_per_sq layer ]
res_per_sq y = SUM[ wire_ratio_y layer * res_per_sq layer ]
width x = SUM[ wire_ratio_x layer * W layer ]
width y = SUM[ wire_ratio_y layer * W layer ]
```

.plib Format

To provide layer parasitics for RC estimation based on the equations shown in this section, define them in the following .plib format.

```
physical_library(name){
    ...
    resistance_lut_template (template_name_id) {
        variable_1: routing_width | routing_spacing ;
        variable_2: routing_width | routing_spacing ;
        index_1 ("float, float, float, ...") ;
        index_2 ("float, float, float, ...") ;
    }
    resource(technology) {
        field_oxide_thickness : float ;
        field_oxide_permitivity : float ;
        ...
        routing_layer(layer_name_id) {
            cap_multiplier : float ;
            cap_per_sq : float ;
            coupling_cap : float ;
            default_routing_width : float ;
            edgecapacitance : float ;
            fringe_cap : float ;
            height : float ;
            inductance_per_dist : float ;
            min_area : float ;
            offset : float ;
            oxide_permittivity : float ;
            oxide_thickness : float ;
            pitch : float ;
            ranged_spacing(float, ..., float) ;
            res_per_sq : float ;
            routing_direction : vertical | horizontal ;
            shrinkage : float ;
            spacing : float ;
            thickness : float ;
            wire_extension : float ;
            lateral_oxide (float, float) ;
            resistance_table (template_name_id) {
                index_1 ("float, float, float, ...") ;
                index_2 ("float, float, float, ...") ;
                values ("float, float, float, ...") :
            }

        } /* end routing_layer */

        plate_cap(value, value, value, value, value, ...) ;
        /* capacitance between wires on lower and upper layer */
        /* MUST BE DEFINED BEFORE ANY routing_wire_model GROUP DEFINITION */
        /* AND AFTER ALL *_layer() DEFINITIONS */
        routing_wire_model(name) {
            /* predefined routing wire ratio model for R/C estimation */
            overlap_wire_ratio(value, value, value, value, value, ...) ;
            /* overlapping wiring percentage between wires on different layers. */
            /* Value between 0 and 100.0 */
```

```
        adjacent_wire_ratio(value, value, value, ...) ;
        /* Adjacent wire percentage between wires on same layers. */
        /* Value between 0.0 and 100.0 */
        wire_ratio_x(value, value, value, ...) ;
        /* x wiring percentage on each routing layer. */
        /* Value between 0.0 and 100.0 */
        wire_ratio_y(value, value, value, ...) ;
        /* y wiring percentage on each routing layer. */
        /* Value between 0.0 and 100.0 */
        wire_length_x : float  ;
        /* estimated length for horizontal wire segment */
        wire_length_y : float  ;
        /* estimated length for vertical wire segment */
    }
}
topological_design_rules() {
    ...
    default_via_generate( ) {
        via_routing_layer ( ) {
            end_of_line_overhang : ;
            overhang ( ) :
        }
        via_contact_layer ( ) {
            end_of_line_overhang : ;
            overhang ( ) :
            rectangle(float, float, float, float) ;
            resistance : float ;
        }
    }
}
process_resource () {
    process_routing_layer () {
        res_per_sq : float;
        cap_per_sq : float ;
        coupling_cap : float ;
        /* coupling effect between parallel wires on same layer */
        fringe_cap : float  ; /* sidewall capacitance per unit length */
        edgecapacitance: float ; /* lumped fringe capacitance */
        inductance_per_dist : float ;
        shrinkage : float ;  /* delta width */
        default_routing_width : float; /* width */
        height : float ; /* height from substrate */
        thickness : float ; /* interconnect thickness */
        lateral_oxide_thickness : float ;
        oxide_thickness : float ;
    }
    process_via () {
        .resistance : float ;
    }
    process_array () {
        default_capacitance : float ;
    }
    process_wire_rule () {
        process_via () {
            resistance : float ;
        }
    }
}
```

```
   macro() {
   ...
   }
}
```

During a psyn_shell session, you can provide a more accurate wire-ratio model to .plib by using the `update_lib` command. The new .plib file contains the wire_ratio model:

```
resource (technology) {
   routing_wire_model(name) {
      overlap_wire_ratio(value, value, value, ...);
      adjacent_wire_ratio(value, value, value, ...);
      wire_ratio_x(value, value, value, ...);
      wire_ratio_y(value, value, value, ...);
      wire_length_x : float;
      wire_length_y : float;
   }
}
```

A

Calculating the Antenna

Antennas are inherent in the manufacturing process.

Deep submicron topologies require a more accurate calculation of the antenna than submicron topologies.

The physical library provides for two antenna calculation models:

- Linear
- Piecewise Linear

Linear Model for Antenna Calculation

The linear model lets you determine the antenna using only segment-length and top-area based calculations. To determine the antenna, you specify the antenna limits for individual pins and how the routing contributes to the antenna.

The following list shows the parameters you can specify and their uses. For more information about how to specify these parameters, see the *Library Compiler Physical Libraries Reference Manual*.

- Global pin parameters

  Use these attributes to define default values for all the input, output and I/O pins for all the cells in the library:

  - `antenna_input_threshold`
  - `antenna_output_threshold`
  - `antenna_inout_threshold`

- Pin-specific parameters

  Use the following attribute to define antenna limits that override the global default values.

  Use this second set of attributes to define how the intra-cell geometries contribute to the antenna.

  - `antenna_metal_area`

Piecewise Linear Model

The piecewise linear model lets you calculate the ratio between a metal layer and gate diffusion area.

The following list shows the parameters you can specify and their uses. For more information about specifying these parameters, see the *Library Compiler Physical Libraries Reference Manual*.

- Pin-specific parameter

  Use this attribute to define how the intra-cell geometries contribute to the antenna:

  - `antenna_metal_area`

Antenna Rule Syntax

The following examples shows the syntax for implementing the piecewise linear model calculation.

```
phys_library(...) {
    ...
    /* template */
    antenna_lut_template (template_name_id) {
        variable_1: antenna_diffusion_area ;
        index_1 (); /* default indices */
    }

    resource (...) {
        ...
    }
    topological_design_rules() {
        ...
        antenna_rule(antenna_rule_name_id) {
            apply_to: gate_area | diffusion_area;
            geometry_calculation_method : all_geometries | connected_only ;
            pin_calculation_method : all_pins | each_pin ;
            routing_layer_calculation_method : side_wall_area | top_area |
                                    side_wall_and_top_area ;
            antenna_ratio (template_name_id) {
                index_1 ("float, float, float, ..."); /* optional */
                values ("float, float, float, ...");
            }
            layer_antenna_factor (layer_name_string, antenna_factor_float);
        }
    ...
    }
    ...
}
```

where

apply_to

Specifies the type of pin geometry that the rule applies to.

geometry_calculation_method

Used with the `pin_calculation_method` attribute to specify which geometries are applied to which pins. See Table A-1 for a matrix of the options.

pin_calculation_method

Used with the `geometry_calculation_method` attribute to specify which geometries are applied to which pins. See Table A-1 for a matrix of the options.

routing_layer__calculation_method

Specifies which property of the routing segments to use to calculate antenna contributions.

antenna_ratio

Specifies the piecewise linear table for antenna calculations.

layer_antenna_factor

Specifies a factor in each routing or contact layer that is multiplied by either the area or the length of the routing segments to get their contribution.

Table A-1 Calculating Geometries on Pins

geometry_calculation_method values	pin_calculation_method values	
	all_pins	each_pin
all_geometries	All the geometries are applied to all pins. The connectivity analysis is not performed. Pins share antennas.	All the geometries of the net are applied to every pin on the net separately. The connectivity analysis is not performed. Antennas are not shared by connected pins. **This is the most pessimistic calculation.**
connected_only	Considers connected geometries as well as sharing. **This is the most accurate calculation.**	Only the geometries connected to the pin are considered. Sharing of antennas is not allowed.

Specifying Antenna Rules

The following examples show how to specify various antenna rules.

Example A-1 shows how to specify an antenna rule on layer metal3 that uses the side wall area of the segments.

Example A-1 Specifying the Side Wall Area of the Segments

```
antenna_lut_template (antenna_template_1) {
   variable_1: antenna_diffusion_area;
   index_1: (0, 2.4, 4.8);
}

antenna_rule(antenna_metal3_only) {
   apply_to: gate_area;
   geometry_calculation_method: connected_only;
   pin_calculation_method: all_pins;
   routing_layer_calculation_method: side_wall_area;
   antenna_ratio (antenna_template_1) {
      values(10, 100, 1000);
   }
```

```
      layer_antenna_factor (metal3, 1);
   }
```

Example A-2 shows how to specify an area-based rule based on M1-M2 vias.

Example A-2 Specifying a Via Area-Based Rule

```
antenna_rule(m1_m2_via_area_only) {
   apply_to: gate_area;
   geometry_calculation_method: connected_only;
   pin_calculation_method: all_pins;
   antenna_ratio (antenna_template_1) {
      values (2, 20, 200);
   }
   layer_antenna_factor(m1_m2, 1);
}
```

Example A-3 shows how to specify a cumulative antenna rule that uses the top area of the segments on layers m1 and m2 (segments and vias).

Example A-3 Specifying a Cumulative Antenna Rule for Multiple Layers

```
antenna_rule(antenna_cumulative_upto_m2) {
   apply_to: gate_area;
   geometry_calculation_method: connected_only;
   pin_calculation_method: all_pins;
   routing_layer_calculation_method: top_area;
   antenna_ratio_table (antenna_template_1) {
      values(10, 100, 1000);
   }
   layer_antenna_factor(metal1, 1);
   layer_antenna_factor(metal2, 1);
   layer_antenna_factor(m1_m2, 1);
}
```

Per Layer Calculations

You can specify layer by layer antenna checks by creating one antenna rule for every routing layer. All the rules specified in the library are used during antenna detection.

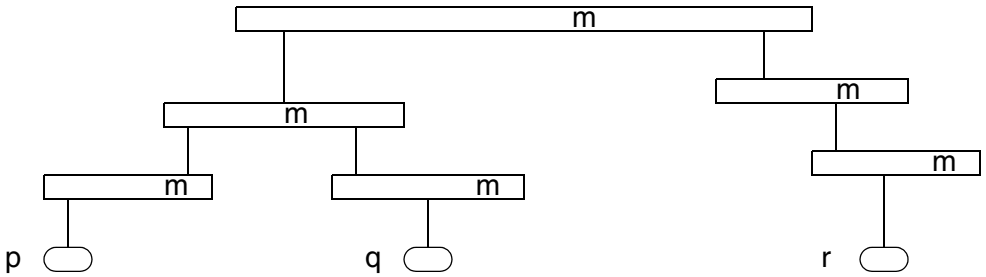
Cumulative Calculations

You can specify cumulative antenna checks by specifying more than one layer in the antenna rule. In the case of cumulative calculation, the antenna is calculated and checked in incremental mode, layer by layer for the reason that an upper layer could add a connection to a large protective diode such that the total cumulative result would be acceptable. However, cumulative antenna for layers below this layer could exceed the limit and have no connection to a protective diffusion area.

Topology Example With Antenna Rule Calculations

Figure A-1 shows the topology of a net with three pins: p, q and r.

Figure A-1 Cumulative Calculations for Antenna Accumulation



The cumulative calculations for antenna accumulation are shown below. The calculations for total_diffusion_area and max_antenna_ratio are similar.

Example A-4 Cumulative Calculations for Antenna

```
Assumption: layer_antenna_factor(all_layers) = 1;

pin_accumulated_ratio(p) = antenna_metal_area(p) / gate_area(p);
pin_accumulated_ratio(q) = antenna_metal_area(q) / gate_area(q);
pin_accumulated_ratio(r) = antenna_metal_area(r) / gate_area(r);

At Layer m1:
    pin_clusters = {(p), (q), (r)};
    pin_accumulated_ratio(p) += area(m1_1) / gate_area(p);
    pin_accumulated_ratio(q) += area(m1_2) / gate_area(q);
    pin_accumulated_ratio(r) += area(m1_3) / gate_area(r);

At Layer m2:
    pin_clusters = {(p,q), (r)}
    pin_acumulated_ratio(p) += area(m2_1) / gate_area(p,q);
    pin_acumulated_ratio(q) += area(m2_1) / gate_area(p,q);
    pin_acumulated_ratio(r) += area(m2_2) / gate_area(r);

At Layer m3:
    pin_clusters = {(p,q,r)}
    pin_accumulated_ratio(p) += area(m3_1) / gate_area(p,q,r);
    pin_accumulated_ratio(q) += area(m3_1) / gate_area(p,q,r);
    pin_accumulated_ratio(r) += area(m3_1) / gate_area(p,q,r);
```

Handling Black Box Cells, Blocks, and Modules

There are two ways you can perform antenna checking when there are blocks in the design.

- Case 1: Protect the block sufficiently such that antenna checking stops as soon as the routing hits a block pin.
- Case 2: Account for the routing inside the block.

Case 1 is easily accomplished by modeling a block's pin with an infinite diffusion capacity.

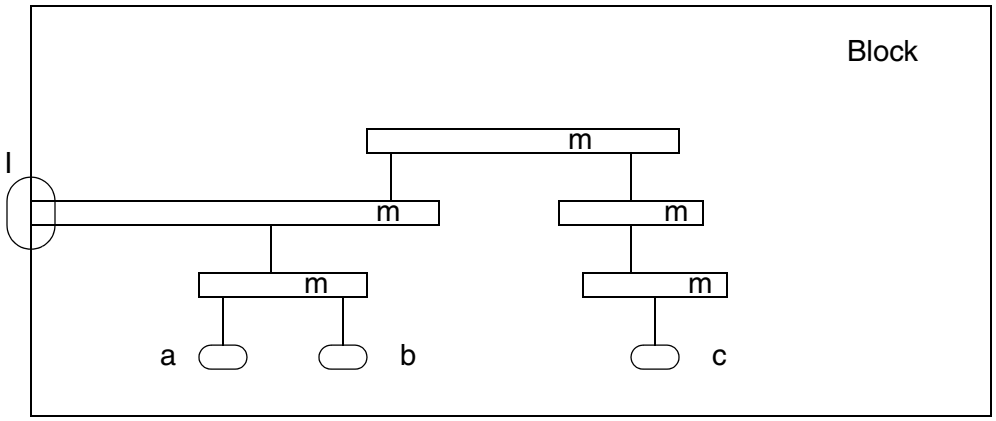
Extrapolating Information From the Pin Model

Case 2 requires that you extrapolate from the block's physical pin model because the routing inside a block is not visible.

The block's pin provides a bridge connecting the block's internal and external routing. The segments inside the block that are below the pin's layer are protected by the bridge.

Figure A-2 shows a block that has an input pin (I) and internal routing.

Figure A-2 A Block's Internal Routing.



To enable antenna calculations in a design that has this block, you need to define the following properties for the input pin (I).

```
Layer m1: Nothing.

Layer m2:
    total_diffusion_area = diffusion_area(a) + diffusion_area(b);
    total_gate_area = gate_area(a) + gate_area(b);
    antenna_metal_area = area(m21);
    partial_ratio = MAX(pin_accumulated_ratio(a, m1), \
                        pin_accumulated_ratio(b, m1));

Layer m3:
    total_diffusion_area = diffusion_area(a) + diffusion_area(b) \
                               + diffusion_area(c);
    total_gate_area = gate_area(a) + gate_area(b) + gate_area(c);
    antenna_metal_area = area(m31);
    partial_ratio = partial_accumulated_ratio(c, m2);
```

The `partial_ratio` for layer m2 is calculated as follows: only pins a and b are connected using m2. Because there will be some external m2 metal unconnected to pin I of the block, you cannot accurately calculate the partial ratio for pins a and b taking m2 area into account. Therefore you must restrict the partial ratio calculations to layers below m2 until the antenna calculations are completed for the design that contains this block. Then you can update the partial ratio with the m2 information.

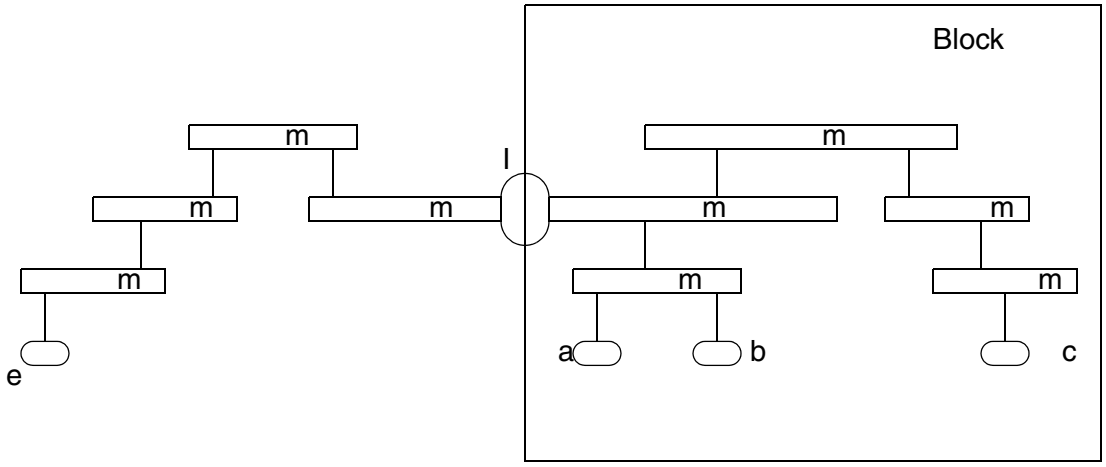
The `partial_ratio` for layer m3 is calculated as follows: the new pin that gets connected is c. Because pin c is the only pin, calculate its accumulated ratio using layers m2 and m1 and use the result to represent the partial_ratio.

The `pin_accumulated_ratio` for an internal pin is calculated as shown during antenna calculations in Example A-1 and Figure A-1. These values are calculated during the block's creation, when antennas are checked on the block. When antennas are checked on the block during and after the block's routing, the antenna model for the block should be created by the antenna checker.

Antenna calculations in the presence of blocks are similar. The total gate and diffusion area, as well as the partial ratio from the block's pin are picked up from the model and added to the calculations described earlier.

Figure A-3 shows a block connected to external routing.

Figure A-3 A Block Connected to External Routing



Example A-5 shows cumulative calculations for antenna accumulation for this block. The calculations for `total_diffusion_area` and `max_antenna_ratio` are similar.

Example A-5 Cumulative Calculations for Antenna Accumulation

```
     Assumption: layer_antenna_factor(all_layers) = 1

At Layer m1:
    pin_clusters = {(e)};
    pin_accumulated_ratio(e) += area(m1_e1) / gate_area(e);

At Layer m2:
    pin_clusters = {(e), (I[m2])}
    pin_accumulated_ratio(e) += area(m2_e1) / gate_area(e);
    I_m2_area = area(m2_e2) + antenna_metal_area(I, m2);
    pin_accumulated_ratio(I[m2]) += partial_ratio(I, m2) \
```

```
                                + (I_m2_area / total_gate_area(I, m2));

At Layer m3:
    pin_clusters = {(e, I[m2], I[m3])}
    total_gate_area = gate_area(e) + total_gate_area(I, m3);
    m3_area = area(m3_e1) + antenna_metal_area(I, m3);
    pin_accumulated_ratio(e) += m3_area / total_gate_area;
    pin_accumulated_ratio(I[m2]) += m3_area / total_gate_area;
    pin_accumulated_ratio(I[m3]) = partial_ratio(I, m3) \
                                    + m3_area / total_gate_area;
```

At each layer block pin, I is considered as if it were a new pin; that is, pins I[m2] and I[m3] are separate pins for antenna calculations. In this case these pins represent the worst case candidate pins at each layer inside the block at block pin I.

B

Generating Physical Library Reports

You can use the Library Compiler `report_lib` command with the `-physical` option command, as shown below, to generate physical library reports.

```
report_lib -physical mylib
```

Sample Reports

Example B-1 shows the information the report_lib command generates and the format. Example B-2 shows an example report.

Example B-1 Physical Library Report Format

```
****************************************
Report : library
Library: test
Version: 2001.08
Date   : Mon Jan 28 17:31:04 2002
****************************************

Library Type            : physical_library
Tool Created            : 2001.08
Date Created            : Tue Feb 27 17:24:30 2001
Library Version         :00000
   site      class     size(x, y)
   --------------------------------------------------------------------------------
   CORE      core      (0,0)


   layer     layer_type    direction     pitch         width         spacing
   --------------------------------------------------------------------------------
   POLY1     masterslice   -             -             -             -
   CUT01     cut           -             -             -             -
   METAL1    routing       x    000
   VIA12     cut           -             -             -             -
   METAL2    routing       x      000
   VIA23     cut           -             -             -             -
   METAL3    routing       x    000
   VIA34     cut           -             -             -             -
   METAL4    routing       x      000
   VIA45     cut           -             -             -             -
   METAL5    routing       x    000
   VIA56     cut           -             -             -             -
   METAL6    routing       x      000
   OVERLAP   overlap       -             -             -             -

================================================================================

routing_wire_model( test ):

  overlap_wire_ratio (in percentage):
   layer1    layer2    value
   --------------------------------------------------------------------------------
   METAL1    METAL2    0.0
   METAL1    METAL3    0.0
   METAL1    METAL4    0.0
   METAL1    METAL5    0.0
   METAL1    METAL6    0.0
   METAL2    METAL1    0.0
   METAL2    METAL3    0.0
   METAL2    METAL4    0.0
   METAL2    METAL5    0.0
   METAL2    METAL6    0.0
```

```
 METAL3    METAL1    0.0
 METAL3    METAL2    0.0
 METAL3    METAL4    0.0
 METAL3    METAL5    0.0
 METAL3    METAL6    0.0
 METAL4    METAL1    0.0
 METAL4    METAL2    0.0
 METAL4    METAL3    0.0
 METAL4    METAL5    0.0
 METAL4    METAL6    0.0
 METAL5    METAL1    0.0
 METAL5    METAL2    0.0
 METAL5    METAL3    0.0
 METAL5    METAL4    0.0
 METAL5    METAL6    0.0
 METAL6    METAL1    0.0
 METAL6    METAL2    0.0
 METAL6    METAL3    0.0
 METAL6    METAL4    0.0
 METAL6    METAL5    0.0

adjacent_wire_ratio (in percentage):
 layer     value
 ------------------------------------------------------------------------------
 METAL1    0
 METAL2    0
 METAL3    0
 METAL4    0
 METAL5    0
 METAL6    0

wire_ratio (in percentage):
 layer     value(x, y)
 ------------------------------------------------------------------------------
 METAL1    (0 , 0)
 METAL2    (0 , 0)
 METAL3    (0 , 0)
 METAL4    (0 , 0)
 METAL5    (0 , 0)
 METAL6    (0 , 0)

wire_length (x, y) = ( 0,0)

capacitance_per_distance:
 layer     value
 ------------------------------------------------------------------------------
 METAL1    0
 METAL2    0
 METAL3    0
 METAL4    0
 METAL5    0
 METAL6    0
capacitance_derating_factor:
 layer     value
 ------------------------------------------------------------------------------
 METAL1    0
 METAL2    0
 METAL3    0
```

```
   METAL4    0
   METAL5    0
   METAL6    0

  resistance_per_sq (x, y) = ( 0, 0)
  capacitance_per_dist (x, y) = (0,0)
  lumped_preroute_capacitance (x, y) = (0, 0)
  lumped_preroute_resistance  (x, y) = (0,0)

================================================================================
   cell      cell_type           in_site   size(x, y)
   ------------------------------------------------------------------------------
   ADDFHX1   core                CORE      (0,0)
   ADDFHX2   core                CORE      (0,0)
   ADDFHX4   core                CORE      (0,0)
   ADDFHXL   core                CORE      (0,0)
   ADDFX1    core                CORE      (0,0)
```

Example B-2 Physical Library Report Example

```
****************************************
Report : library
Library: mylin
Version: 2002.05
Date   : Mon Dec 10 18:15:50 2001
****************************************

Library Type            : physical_library
Tool Created            : 2002.05
Date Created            : Tue May 22 14:23:01 2001
Library Version         : 1.000000
   layer     layer_type    direction     pitch         width         spacing
   ------------------------------------------------------------------------
   A1 masterslice   -             -             -             -
   V0 cut           -             -             -             -
   M1        routing       horizontal    6.000e-01     3.000e-01     1.600e-01
   V1        cut           -             -             -             -
   M2        routing       vertical      6.000e-01     3.000e-01     1.600e-01
   V2        cut           -             -             -             -
   M3        routing       horizontal    6.000e-01     3.000e-01     1.600e-01
   OVERLAP   overlap       -             -             -             -

 LIST OF USER DEFINED AND GENERATED VIAS
   ------------------------------------------------------------------------
 M1_A1 ------------ ---------  IS_DEFAULT resistance =  5.00000e+00
         A1  (-2.2000e-01, -2.2000e-01, 2.2000e-01, 2.2000e-01)
         CONT  (-1.6000e-01, -1.6000e-01, 1.6000e-01, 1.6000e-01)
         M1  (-2.2000e-01, -2.2000e-01, 2.2000e-01, 2.2000e-01)
 M2_M1 ------------ ---------  IS_DEFAULT resistance =  3.00000e+00
         M1  (-2.2000e-01, -2.2000e-01, 2.2000e-01, 2.2000e-01)
         V1  (-1.6000e-01, -1.6000e-01, 1.6000e-01, 1.6000e-01)
         M2  (-2.6000e-01, -2.6000e-01, 2.6000e-01, 2.6000e-01)
 SP_M2_M1 ------------ ---------  ---------- resistance =  N/A
         M1  (-2.6000e-01, -2.6000e-01, 2.6000e-01, 2.6000e-01)
         V1  (-1.6000e-01, -1.6000e-01, 1.6000e-01, 1.6000e-01)

================================================================================
```

```
site      class     size(x, y)
--------------------------------------------------------------------------
CORE      core      (8.000e-01, 9.600e+00)
CORNER    pad       (3.720e+02, 3.720e+02)
PAD       pad       (5.000e-01, 3.720e+02)


cell      cell_type           in_site   size(x, y)
--------------------------------------------------------------------------
ad01d0    core                CORE      (1.360e+01, 9.600e+00)
ad01d1    core                CORE      (1.520e+01, 9.600e+00)
ad01d2    core                CORE      (1.680e+01, 9.600e+00)
 iocorner bottomright_endcap CORNER     (3.720e+02, 3.720e+02)
iofill    pad                 PAD       (8.000e-01, 3.720e+02)
iofilla   pad                 PAD       (4.000e-01, 3.720e+02)
```

Index

L

M

N

O

P

R

S