cadence

Using Encounter™ RTL Compiler

Product Version 5.2
December 2005

Cadence Design Systems, Inc., 555 River Oaks Parkway, San Jose, CA 95134, USA

Contents

Preface

About This Manual

This manual describes how to use RTL Compiler.

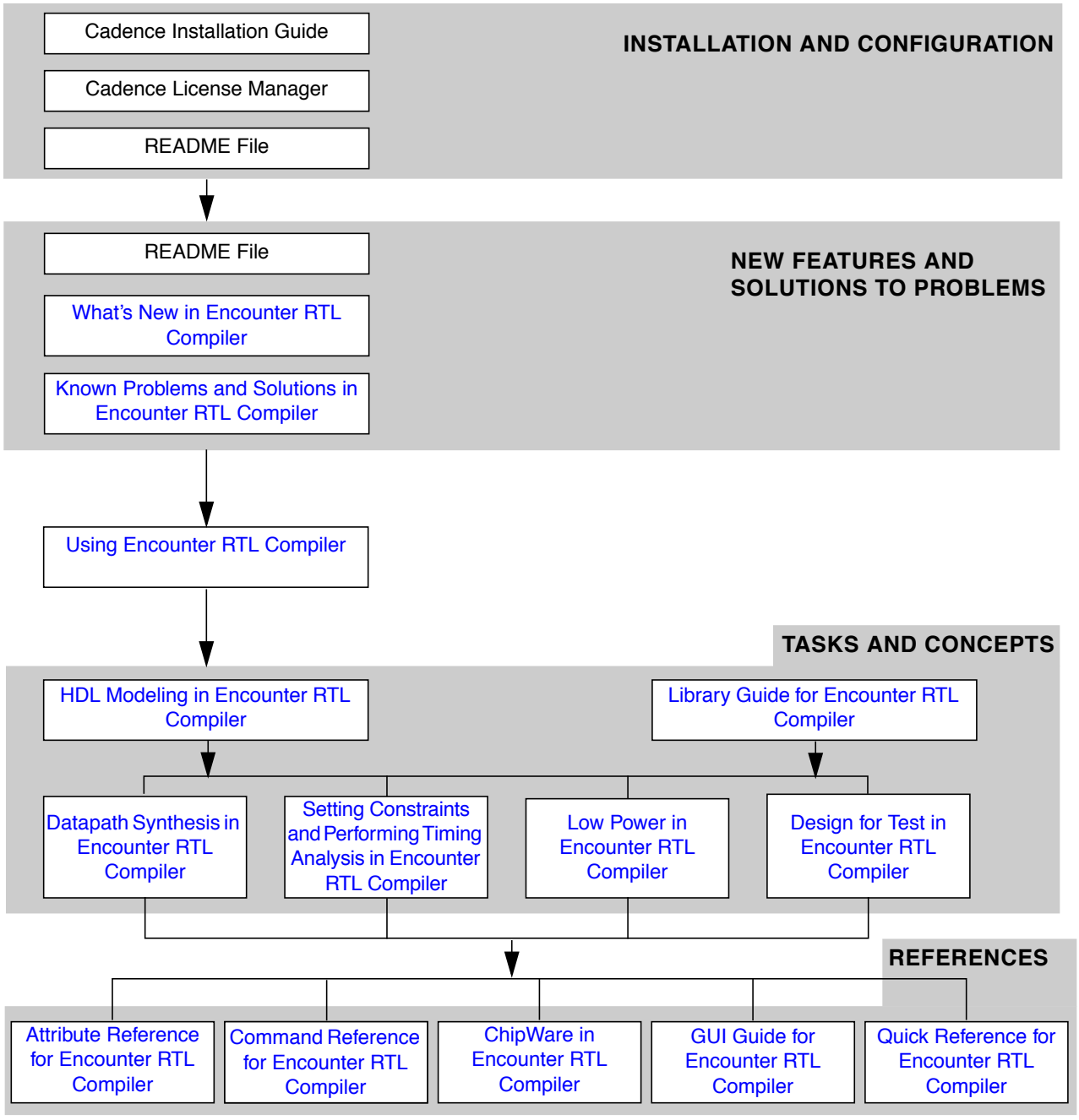
Additional References

The following sources are helpful references, but are not included with the product documentation:

- TclTutor, a computer aided instruction package for learning the Tcl language: http://www.msen.com/~clif/TclTutor.html.
- TCL Reference, *Tcl and the Tk Toolkit*, John K. Ousterhout, Addison-Wesley Publishing Company
- IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language (IEEE Std.1364-1995)
- IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language (IEEE Std. 1364-2001)
- IEEE Standard VHDL Language Reference Manual (IEEE Std. 1076-1987)
- IEEE Standard VHDL Language Reference Manual (IEEE Std. 1076-1993)

Note: For information on purchasing IEEE specifications go to http://shop.ieee.org/store/ and click on *Standards.*

How to Use the Documentation Set



Reporting Problems or Errors in Manuals

The Cadence Online Documentation System, CDSDoc, lets you view, search, and print Cadence product documentation. You can access CDSDoc by typing `cdsdoc` from your Cadence tools hierarchy.

Clicking the *Feedback* button lets you send e-mail directly to Cadence Technical Publications. Use it if you find:

- An error in the manual
- An omission of information in a manual
- A problem displaying documents

Customer Support

Cadence offers live and online support, as well as customer education and training programs.

SourceLink Online Customer Support

SourceLink® online customer support offers answers to your most common technical questions. It lets you search more than 40,000 FAQs, notifications, software updates, and technical solutions documents that give you step-by-step instructions on how to solve known problems. It also gives you product-specific e-mail notifications, software updates, service request tracking, up-to-date release information, full site search capabilities, software update ordering, and much more.

For more information on SourceLink go to:

http://www.cadence.com/support/sourcelink.aspx

Other Support Offerings

- **Call centers**—Provide live customer support from Cadence experts who can answer many questions related to products and platforms.
- **Software downloads**—Provide you with the latest versions of Cadence products.
- **Education services**—Offers instructor-led classes, self-paced Internet, and virtual classroom.

- **University software program support**—Provides you with the latest information to answer your technical questions.

For more information on these support offerings go to:

http://www.cadence.com/support

Messages

From within RTL Compiler there are two ways to get information about messages.

- Use the `report messages` command.

  For example:

  ```
  rc:/> report messages
  ```

  This returns the detailed information for each message output in your current RTL Compiler run. It also includes a summary of how many times each message was issued.

- Use the `man` command.

  Note: You can only use the `man` command for messages within RTL Compiler.

  For example, to get more information about the "TIM-11" message, type the following command:

  ```
  rc:/> man TIM-11
  ```

If you do not get the details that you need or do not understand a message, either contact Cadence Customer Support to file a PCR or email the message ID you would like improved to:

rc_msg_improvement@cadence.com

Man Pages

In addition to the Command and Attribute References, you can also access information about the commands and attributes using the man pages in RTL Compiler. Man pages contain the same content as the Command and Attribute References. To use the man pages from the UNIX shell:

1. Set your environment to view the correct directory:

   ```
   setenv MANPATH $CDN_SYNTH_ROOT/share/synth/man
   ```

2. Enter the name of the command or attribute that you want either in RTL Compiler or within the UNIX shell. For example:
   - `man check_dft_rules`
   - `man cell_leakage_power`

Command-Line Help

You can get quick syntax help for commands and attributes at the RTL Compiler command-line prompt. There are also enhanced search capabilities so you can more easily search for the command or attribute that you need.

Note: The command syntax representation in this document does not necessarily match the information that you get when you type `help` *`command_name`*. In many cases, the order of the arguments is different. Furthermore, the syntax in this document includes all of the dependencies, where the help information does this only to a certain degree.

If you have any suggestions for improving the command-line help, please e-mail them to:

synthesis_help@cadence.com

Getting the Syntax for a Command

Type the `help` command followed by the command name.

For example:

```
rc:/> help path_delay
```

This returns the syntax for the `path_delay` command.

Getting the Syntax for an Attribute

Type the following:

```
rc:/> get_attribute attribute name * -help
```

For example:

```
rc:/> get_attribute max_transition * -help
```

This returns the syntax for the `max_transition` attribute.

Searching for Attributes

You can get a list of all the available attributes by typing the following command:

```
rc:/> get_attribute * * -help
```

Searching For Commands When You Are Unsure of the Name

You can use help to find a command if you only know part of its name, even as little as one letter.

- If you only know the first few letters of a command you can get a list of commands that begin with that letter.

  For example, to get a list of commands that begin with “ed”, you would type the following command:

  ```
  rc:/> ed* -h
  ```

- You can type a single letter and press `Tab` to get a list of all commands that contains that letter.

  For example:

  ```
  rc:/> c <Tab>
  ```

  This returns the following commands:

  ```
  ambiguous "c": cache_vname calling_proc case catch cd cdsdoc change_names
  check_dft_rules chipware clear clock clock_gating clock_ports close cmdExpand
  command_is_complete concat configure_pad_dft connect_scan_chains continue
  cwd_install ...
  ```

- You can also type a sequence of letters and press `Tab` to get a list of all commands that contain those letters.

  For example:

  ```
  rc:/> path_ <Tab>
  ```

  This returns the following commands:

  ```
  ambiguous "path_": path_adjust path_delay path_disable path_group
  ```

Documentation Conventions

Text Command Syntax

The list below defines the syntax conventions used for the RTL Compiler text interface commands.

`literal`	Nonitalic words indicate keywords you enter literally. These keywords represent command or option names.
`arguments and options`	Words in italics indicate user-defined arguments or information for which you must substitute a name or a value.
|	Vertical bars (OR-bars) separate possible choices for a single argument.
`[ ]`	Brackets indicate optional arguments. When used with OR-bars, they enclose a list of choices from which you can choose one.
`{ }`	Braces indicate that a choice is required from the list of arguments separated by OR-bars. Choose one from the list. `{ argument1 | argument2 | argument3 }`
`{ }`	Braces, used in Tcl commands, indicate that the braces must be typed in.
`...`	Three dots (...) indicate that you can repeat the previous argument. If the three dots are used with brackets (that is, `[argument]...`), you can specify zero or more arguments. If the three dots are used without brackets (`argument...`), you must specify at least one argument.
#	The pound sign precedes comments in command files.

1

Introduction

RTL Compiler is a fast, high capacity synthesis solution for demanding chip designs. Its patented core technology, “global focused synthesis,” produces superior logic and interconnect structures for nanometer-scale physical design and routing. RTL Compiler complements the existing Cadence solutions and delivers the best wires for nanometer-scale designs.

RTL Compiler produces designs for processors, graphics, and networking applications. Its globally focused synthesis results in rapid timing closure without compromising run time. RTL Compiler’s high capacity furthermore enhances designer productivity by simplifying constraint definition and scripting.

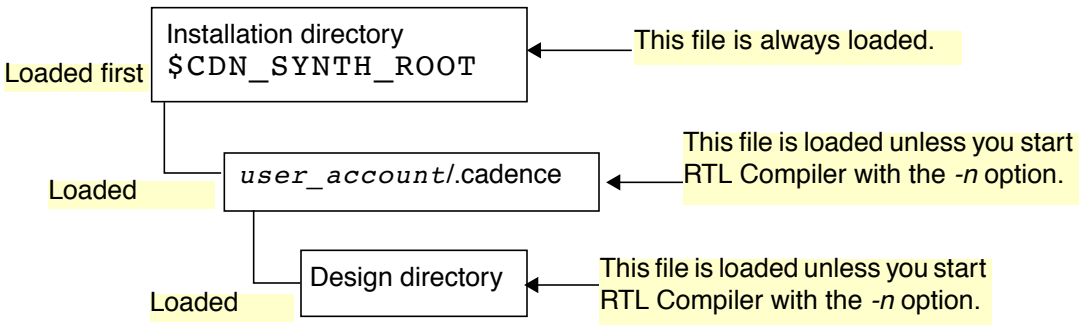
Using the .synth_init Initialization File

This section describes the location and use of the `.synth_init` initialization file. The `.synth_init` file may be located in three different directories:

- Installation root directory — This usually contains the site-specific setup. This file is always loaded.
- Under the user’s `.cadence` directory — Make a directory named `.cadence` under the user’s home directory. This contains the user-specific setup. This file in not loaded if you launch RTL Compiler with the `-n` option.
- Current design directory — This contains a project-specific setup. This file in not loaded if you launch RTL Compiler with the `-n` option.

Figure 1-1 illustrates the possible locations and loading priorities of the `.synth_init` file.

Figure 1-1 Locations of the .synth_init file



Working in the RTL Compiler Shell

Interaction with RTL Compiler occurs within the RTL Compiler shell. It is an environment similar to that of UNIX and shares many characteristics with the UNIX environment.

Note: Once you are in RTL Compiler you no longer have access to the UNIX operating system until your RTL Compiler session has ended or a new terminal session is launched. For example, once your are in RTL Compiler, the `cd` command will change your directory in the RTL Compiler Design Information Hierarchy and *not* the UNIX directory tree.

Navigation

RTL Compiler uses the Design Information Hierarchy to interface with its database. The Design Information Hierarchy is very similar to the UNIX directory structure. Therefore, familiar navigation commands are available to navigate the hierarchy.

The following command indicates that your current directory within the Design Information Hierarchy is `/designs`:

```
rc:/designs> pwd

/designs
```

The following command changes the current directory to the root (“`/`”) directory:

```
rc:/designs> cd /
```

The following command lists the contents of the root (“`/`”) directory:

```
rc:/> ls /

./    designs/    hdl_libraries/    libraries/    messages/
```

For more information regarding RTL Compiler commands including other navigation commands, consult the *RTL Compiler Command Reference*. For more information regarding the Design Information Hierarchy, refer to RTL Compiler Design Information Hierarchy.

Objects and Attributes

In RTL Compiler, objects are general terms for items within the Design Information Hierarchy. For example, an object can be a design, subdesign, library, directory (including the root directory), port, pin, and so on.

The nature of an object can be changed by attributes. That is, objects can behave differently according to which attributes have been placed on them. As an example of showing the relationship between objects and attributes: If you take an “apple” as an object, you can assign it the attribute of being “green” in color and “smooth” in texture.

See a full list of available attributes in the *Attribute Reference for Encounter RTL Compiler*. Changing the settings of these attributes is performed by using the `set_attribute` command.

Output Redirection

All commands in the RC shell output their data to the standard output device (`stdout`). To save a record of the data produced, you need to redirect the command’s output to a file. This redirection has the same form as the standard UNIX redirection:

- One greater-than sign (>) writes output to the specified file, overwriting any existing file.
- Two greater-than signs (>>) appends output to an existing file, or creates a new file if none exists.

The following example redirects the output from a timing report into a file:

```
rc:/> report timing > timing.rpt
```

This example appends the timing report to an existing file:

```
rc:/> report area  >> design.rpt
```

Additional examples of command redirection are shown in the following section.

Scripting

Scripting is the most efficient way of automating the tasks that are performed with any tool. To support scripting at both a basic and advanced level, RTL Compiler uses the standard scripting language, Tool Control Language (Tcl).

In most cases, an RTL Compiler script consists of a series of RTL Compiler commands listed in a file, in the same format that is used interactively. This script file is executed by using either the –`f` command line switch from your UNIX environment or the `include` command from within RTL Compiler.

The following example, `design1.g`, is a simple script that loads a technology library, loads a design, sets the constraints, synthesizes, maps, and finally writes out the design:

```
set_attribute library tech.lib
read_hdl design1.v
elaborate
set clock [define_clock –period 2500 –name clock1 [clock_ports]]
external_delay –input 0 –clock $clock /designs/*/ports_in/*
external_delay –output 0 –clock $clock /designs/*/ports_out/*
synthesize -to_mapped
report timing >  design1.rpt
report area   >> design1.rpt
write_hdl  > design1_net.v
quit
```

- Run this script from your UNIX command line by typing the following command:

  ```
  unix:/> rc –f design1.g
  ```

- Alternatively, run the script within RTL Compiler by typing the following command:

  ```
  rc:/> include design1.g
  ```

A collection of helpful Tcl scripts accompany RTL Compiler to aid your synthesis sessions. Load all the scripts by typing the following command within RTL Compiler:

```
rc:/> include load_etc.tcl
```

The `load_etc.tcl` file, which loads all the relevant Tcl scripts, is located at:

```
$CDN_SYNTH_ROOT/lib/etc
```

RTL Compiler by default includes the `load_etc.tcl` file in the search path. Therefore, it is unnecessary to explicitly specify the path name.

Using SDC Interactively

Use SDC interactively by adding the `dc::` prefix to any SDC, as shown in the following example:

```
dc::set_output_delay 1.0 -clock foo [dc::get_ports boo*]
```

The following command uses the `dc::` prefix and the `-help` option to return the syntax for a specific SDC command:

```
rc:/> dc::set_clock_latency -help
```

Important

When you are mixing SDC and RTL Compiler commands, be aware that the units for capacitance and delay are different. For example, in the following command, the SDC `set_load` command expects the load in pF, but the RTL Compiler command `get_attribute` will return the load in fF:

```
dc::set_load [get_attribute load slow/INVX1/A] [dc::all_outputs]
```

This causes the capacitance set on all outputs to be off by a factor of 1000.

Getting Help

Online help is available to explain RTL Compiler commands, attributes, and messages.

Help Command

Use the `help` command to obtain a brief description of a command without its syntax:

```
rc:/> help synthesize

That command is:
  Synthesis:
    synthesize  synthesize the design
```

Using the help command alone will return a complete list of all RTL Compiler commands:

```
rc:/> help
```

For more information on the `help` command, see the *RTL Compiler Command Reference*.

The -help Option

The `-help` option provides a brief description of a command or attribute and its syntax. You can also use the option with wildcards to return a comprehensive list of all available attributes. For example, the following command returns a complete list of writeable attributes:

```
rc:/> set_attribute * * -help
```

The first wildcard star (“*”) represents the attribute name, while the second represents the object. If you want to return a complete list of both write and read-only attributes, type the following command:

```
rc:/> get_attribute * * -help
```

The following command returns help descriptions on all `retime` attributes:

```
rc:/> get_attribute retime * -help
```

The following command shows the syntax for the `elaborate` command:

```
rc:/> elaborate -help
  elaborate: elaborate a design
Usage: elaborate [-parameters <integer>+] [<string>+]
    -parameters <integer>+:
        list of design parameters
    <string>+:
        elaborate toplevel module
```

The following command uses the `dc::` prefix and the `-help` option to return the syntax for a specific SDC command:

```
rc:/> dc::set_clock_latency -help
```

RTL Compiler Messages: Errors, Warnings, and Information

If there are any issues during an RTL Compiler session, messages categorized as *Errors*, *Warnings*, or *Information* will be outputted. Error messages stop the current session; warnings and information messages allow the process continue.

The following messages are examples of warning and information messages:

```
Warning: Variable is read but never written [ELAB-VLOG-16]
Warning: Variable has multiple drivers [ELAB-VLOG-14]
Info   : Unused variable [ELAB-VLOG-33]
```

You can pass the `help` argument to the `get_attribute` command to obtain information about particular messages. For example, the following command returns information about the synthesis message `SYN-100`:

```
rc:/> get_attribute help [find / -message TIM-11]

Use 'report timing -lint' for more information.
```

All messages are located in the `/messages` directory within RTL Compiler.

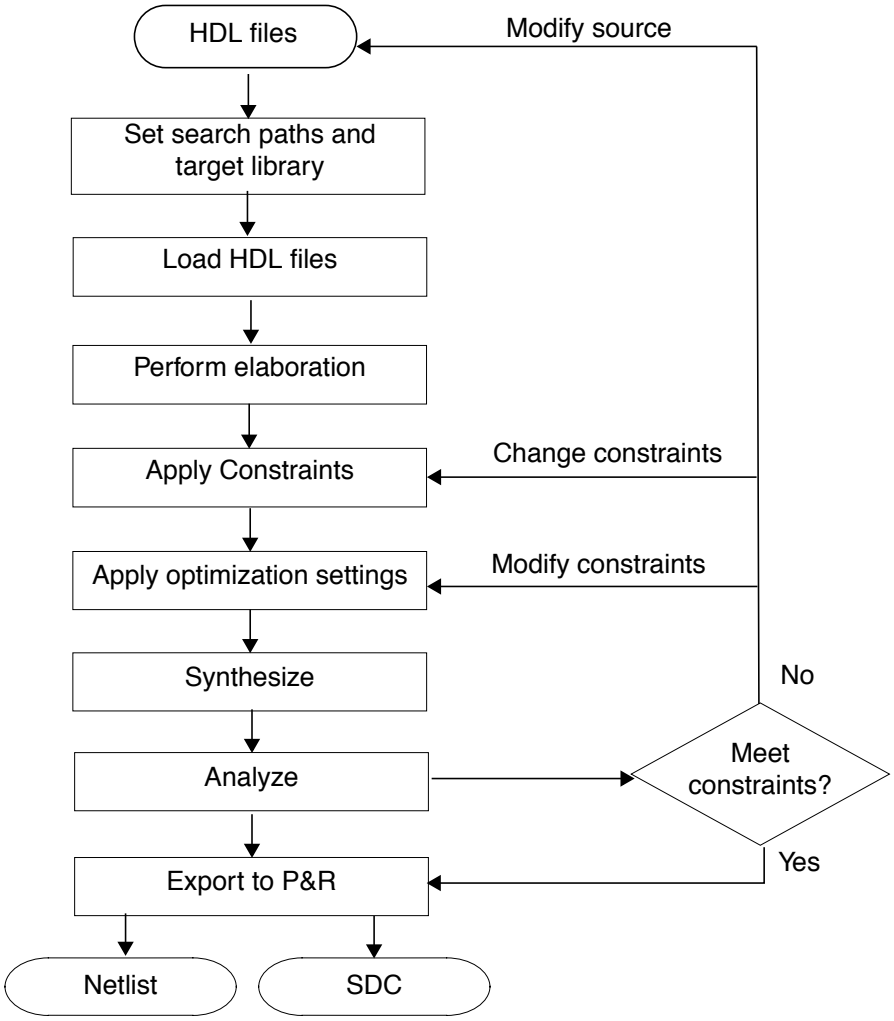
2

The RTL Compiler Work Flow

Overview

Figure 2-1shows the RTL Compiler work flow. This section briefly and sequentially describes all the tasks within the work flow, from setting the target technology library and reading in the HDL file to writing out the synthesized netlist or SDC file. The subsequent chapters discuss each of these steps in more detail. A separate chapter is dedicated to describing the Design Information Hierarchy, which is the data structure used to interface with the RTL Compiler database.

Figure 2-1 RTL Compiler Work Flow



Tasks

Starting RTL Compiler

The `rc` command starts RTL Compiler from the UNIX environment. The syntax of the `rc` command is:

```
rc [-no_custom] [-32] [-64][-use_license RTL_Compiler_Ultra
|RTL_Compiler_Verification | First_Encounter_GXL | SOC_Encounter_GXL | FE_GPS |
SOC_Encounter_GPS | Virtuoso_Digital_Implement][-files script_file]
[-execute command][-lsf_cpu integer] [lsf_queue queue_name] [-cmdfile <string>]
[-logfile log_file_name] [-nologfile] [-vdi] [-version]
```

To invoke RTL Compiler:

- Type the following command at the UNIX prompt to launch RTL Compiler in 32-bit mode with the `RTL_Compiler_Ultra` license:

  ```
  unix>  rc -32
  ```

  Alternatively, you can just type the `rc` command because the 32-bit mode is the default mode:

  ```
  unix> rc
  ```

- Type the following command if you want the launching of RTL Compiler to fail if no `RTL_Compiler_Ultra` license is available:

  ```
  unix> rc -use_license RTL_Compiler_Ultra
  ```

  If you specify multiple licenses, only the last one will be used.

- The following commands have the same effect. Therefore, you should use one or the other and not both in conjunction:

  ```
  unix> rc -use_license Virtuoso_Digital_Implement
  ```

  is the same as:

  ```
  unix> rc -vdi
  ```

- Type the following command at the UNIX prompt to launch RTL Compiler in 64-bit mode:

  ```
  unix> rc -64
  ```

  The initial splash screen will tell you whether you are in 64 or 32 bit mode.

- Type the following command to simultaneously invoke RTL Compiler as a background process and execute a script:

  ```
  unix> rc < script_file_name &
  ```

- Type the following command to simultaneously invoke RTL Compiler, execute script, and exit if any problems are encountered with the script:

  ```
  unix> rc -files script_file_name < /dev/null
  ```

- Type the following command to simultaneously set the script search path and invoke RTL Compiler:

```
unix> rc -execute "set_attribute script_search_path pathname"
```

- Type the following command to simultaneously set the a Tcl variable, invoke RTL Compiler, and launch a script:

  ```
  unix> rc -files script_file_name -execute "set variable_name value"
  ```

- RTL Compiler supports super-threading on LSF. Use the `-lsf_cups` option to specify the number of processes to send to LSF and the `-lsf_queue` option to specify a particular LSF queue:

  ```
  unix> rc -lsf_cups 4 -lsf_queue teagan_queue
  ```

  If the `super_thread_servers` and `bsub_options` attributes are specified within a RTL Compiler session, they will override the `-lsf_cpus` and `-lsf_queue` options. In the following example, two processes will be sent to LSF and the `stormy queue` used:

  ```
  unix> rc -lsf_cups 4 -lsf_queue teagan_queue
  ...
  rc:/> set_attribute super_thread_servers {lsf lsf}
  rc:/> set_attribute bsub_options stormy_queue
  ```

Tip

You can abbreviate the options for the `rc` command as long as there are no ambiguities with other options. In the following example, `-ver` would imply the `-version` option:

```
unix> rc -ver
```

Just using `rc -v` would not work because there is more than one option that starts with the letter "v."

Generating Log Files

By default, RTL Compiler generates a log file named `rc.log`. The log file contains the entire output of the current RTL Compiler session. You can set the level of verbosity in the log file with the `information_level` attribute, as described in Setting Information Level and Messages on page 32.

You can customize the log file name while invoking RTL Compiler or during the synthesis session. The following examples simultaneously customize the log file name and execute a script file. RTL Compiler will overwrite any log file with the same name.

- Start RTL Compiler with the `-logfile` option:

  ```
  unix> rc -f script_file_name -logfile log_file_name
  ```

- Start RTL Compiler as a background process and write out the log file:

```
unix> rc < script_file_name > log_file_name &
```

- Suppress the generation of any log file by using the `-nologfile` option when invoking RTL Compiler.

  ```
  unix> rc -f script_file_name -nologfile
  ```

- Customizes the log file within an RTL Compiler session through the `stdout_log` attribute:

  ```
  rc:/> set_attribute stdout_log log_file_name
  ```

Generating the Command File

By default, RTL Compiler generates a command history file named `rc.cmd`, which contains a record of all the commands that were issued in a particular session. This file is created in addition to the log file.

To customize the command file name, use the `command_log` attribute within a RTL Compiler session. The following example changes the default name of `rc.cmd` to `rc_command_list.txt`:

```
rc:/> set_attribute command_log rc_command_list.txt
```

Setting Information Level and Messages

You can control the amount of information RTL Compiler writes out in the output logfiles.

- To specify the verbosity level, type the following command:

  ```
  rc:/> set_attribute information_level value /
  ```

  where `value` is an integer value between `0` (minimum) and `9` (maximum). The recommended level is `6`. The `information_level` attribute is a root-level attribute. Therefore, like all root level attributes, it needs to be set on the root level ("`/`") like the above example.

Tip

For analysis and debugging, set the information level to `9`.

Loading the Encounter Configuration File

RTL Compiler can load the Encounter configuration file through the `read_encounter conf` command. The configuration file contains Tcl variables that describe design information such as the netlist or RTL, technology libraries, LEF information, constraints, and capacitance

tables. It also contains operating environment information such as how to handle black-boxes, high fanout nets, and unconstrained ports. Encounter configuration files have the `.config` extension:

```
rc:/> read_encounter_conf teagan.conf
```

Because of the information that an Encounter configuration file contains, you may proceed directly to synthesis (Performing Synthesis on page 37) after loading the file.

Specifying Explicit Search Paths

You can specify the search paths for libraries, scripts, and HDL files. The default search path is the directory in which RTL Compiler is invoked.

To set the search paths, type the following `set_attribute` commands:

```
rc:/> set_attribute lib_search_path path /
rc:/> set_attribute script_search_path path /
rc:/> set_attribute hdl_search_path path /
```

where *`path`* is the full path of your target library, script, or HDL file locations.

The slash (“/”) in these commands refers to the root-level RTL Compiler object that contains all global RTL Compiler settings.

Setting the Target Technology Library

After you set the library search path, you need to specify the target technology library for synthesis using the `library` attribute.

- To specify a single library:

  ```
  rc:\> set_attribute library lib_name.lbr /
  ```

  RTL Compiler will use the library named `lib_name.lbr` for synthesis. RTL Compiler can also accommodate the `.lib` (Liberty) library format. In either case, ensure that you specify the library at the root-level (“/”).

  Note: If the library is not in a previously specified search path, specify the full path, as follows:

  ```
  rc:/> set_attribute library /usr/local/files/lib_name.lbr
  ```

- To specify a single library compressed with gzip:

  ```
  rc:/> set_attribute library lib_name.lbr.gz /
  ```

- To append libraries:

```
rc:/> set_attribute library {{lib1.lib lib2.lib}}
```

After `lib1.lib` is loaded, `lib2.lib` is appended to `lib1.lib`. This appended library retains the `lib1.lib` name.

Importing the LEF Files

LEF files are ASCII files that contain library information such as layer, via, placement site type, and macro cell definitions. This information is usually divided into two separate LEF files for easier manageability. However, this division is not required. The two LEF files can be divided in the following manner:

- *Technology LEF file* — Contains all of the LEF technology information for a design, such as placement and routing design rules and process information for layers.
- *Cell Library LEF file* — Contains the macro and standard cell information for a design.

RTL Compiler supports LEF 5.5. Refer to the *LEF/DEF Language Reference* for more information on LEF files.

To import LEF files, use the `lef_library` attribute. The following example imports two LEF files:

```
rc:/> set_attribute lef_library {a.lef b.lef}
```

Use the `get_attribute` command to return a list of imported LEF files:

```
rc:/> get_attribute lef_library
a.lef
b.lef
```

Those pins that are found only in the LEF library and not in the timing library (physical pins) are automatically ignored.

Extracting Capacitance Information

Use the `cap_table_file` attribute to load the capacitance table file. Although a capacitance table file is not required, it will generally provide better results than synthesizing the design with only the LEF file.

```
rct:/> set_attribute cap_table_file tech.cap
```

RTL Compiler ignores the `EXTENDED` section of the capacitance table file.

Note: Always read the capacitance table file after the LEF files.

Loading the HDL Files

Use the `read_hdl` command to read HDL files into RTL Compiler. When you issue a `read_hdl` command, RTL Compiler reads the files and performs syntax checks.

➤ To load one or more Verilog files:

- You can read the files sequentially:

```
read_hdl file1.v
read_hdl file2.v
read_hdl file3.v
```

- Or you can load the files simultaneously:

```
read_hdl { file1.v file2.v file3.v }
```

Your files may have extra, hidden characters (e.g. line terminators) if they are transferred from Windows/Dos to UNIX. Be sure to eliminate them because RTL Compiler will issue an error when it encounters these characters.

For more information on loading HDL files, see Loading HDL Files on page 102.

Performing Elaboration

Elaboration is only required for the top-level design. The `elaborate` command automatically elaborates the top-level design and all of its references. During elaboration, RTL Compiler performs the following tasks:

- Builds data structures
- Infers registers in the design
- Performs high-level HDL optimization, such as dead code removal
- Checks semantics

Note: If there are any gate-level netlists read in with the RTL files, RTL Compiler automatically links the cells to their references in the technology library during elaboration. You do not have to issue an additional command for linking.

At the end of elaboration, RTL Compiler displays any unresolved references (immediately after the key words `Done elaborating`):

```
Done elaborating '<top_level_module_name>'.
```

```
Cannot resolve reference to <ref01>
Cannot resolve reference to <ref02>
Cannot resolve reference to <ref03>
...
```

After elaboration, RTL Compiler has an internally created data structure for the whole design so you can apply constraints and perform other operations.

For more information on elaborating a design, see Performing Elaboration on page 127.

Applying Constraints

After loading and elaborating your design, you must specify constraints. The constraints include:

- Operating conditions
- Clock waveforms
- I/O timing

You can apply constraints in several ways:

- Type them manually in the RTL Compiler shell.
- Include a constraints file.
- Read in SDC constraints.

"Applying Constraints" on page 147 gives a broader overview on constraints. For detailed information, see *Setting Constraints and Performing Timing Analysis in Encounter RTL Compiler*.

Applying Optimization Constraints

In addition to applying design constraints, you may need to use additional optimization strategies to get the desired performance goals from synthesis.

With RTL Compiler, you can perform any of the following optimizations:

- Remove designer-created hierarchies (ungrouping)
- Create additional hierarchies (grouping)
- Synthesize a sub-design
- Create custom cost groups for paths in the design to change the synthesis cost function

For example, the timing paths in the design can be classified into the following four distinct cost groups:

- Input-to-Output paths (I2O)
- Input-to-Register paths (I2C)
- Register-to-Register (C2C)
- Register-to-Output paths (C2O)

For each path group, the worst timing path drives the synthesis cost function. For more information on optimization strategies and related commands, see "Defining Optimization Settings" on page 153.

Performing Synthesis

After the constraints and optimizations are set for your design, you can proceed with synthesis by issuing the `synthesize` command.

➤ To synthesize your design using the `synthesize` command, type the following command:

```
rc:\> synthesize -to_mapped
```

For details on the `synthesize` command, see "Synthesizing your Design" on page 172.

After synthesis, you will have a technology-mapped gate-level netlist.

Analyzing the Synthesis Results

After synthesizing the design, you can generate detailed timing and area reports using the various `report` commands:

- To generated a detailed area report, use `report area`.
- To generate a detailed gate selection and area report, use `report gates`.
- To generate a detailed timing report, including the worst critical path of the current design, use `report timing`.

For more information on generating reports for analysis, see "Generating Reports" on page 209 and "Analysis Commands" in the *Command Reference for Encounter RTL Compiler*.

Writing Out Files for Place and Route

The last step in the flow involves writing out the gate-level netlist, SDC, or Encounter configuration file for processing in your place and route tool. For more information on this topic, see Interfacing to Place and Route on page 223.

Note: By default, the `write` commands write output to `stdout`. If you want to save your information to a file, use the redirection symbol (>) and a filename.

➤ To write the gate-level netlist, use the `write_hdl` command.

Because `write_hdl` directs the output to `stdout`, use file redirection to create a design file on disk, as shown in the following example:

```
rc:/> write_hdl > design.v
```

This command writes out the gate-level netlist to a file called `design.v`.

➤ To write out the design constraints, use the `write_script` command, as shown in the following example:

```
rc:/> write_script > constraints.g
```

This command writes out the constraints to the file `constraints.g`.

➤ To write the design constraints in SDC format, use the `write_sdc` command, as shown in the following example:

```
rc:/> write_sdc > constraints.sdc
```

This command writes the design constraints to a file called `constraints.sdc`.

Note: Because some place and route tools require different structures in the netlist, you may need to make some adjustments either before synthesis or before writing out the

netlist. For more information about these issues, see "Interfacing to Place and Route" on page 223.

➤ To write the Encounter configuration file, use the `write_encounter` command:

```
rc:/> write_encounter design design_name
```

Exiting RTL Compiler

There are three ways to exit RTL Compiler when you finish your session:

- Use the `quit` command.
- Use the `exit` command.
- Use the `Control-c` key combination twice in succession to exit the tool immediately.

Using Design for Test and Low Power Synthesis in RTL Compiler Ultra

To learn more about using design for test and low power synthesis in RTL Compiler, see the following users guides that are included in your software installation package:

- *Using Design for Test in Encounter RTL Compiler Ultra*
- *Using Low Power in Encounter RTL Compiler Ultra*

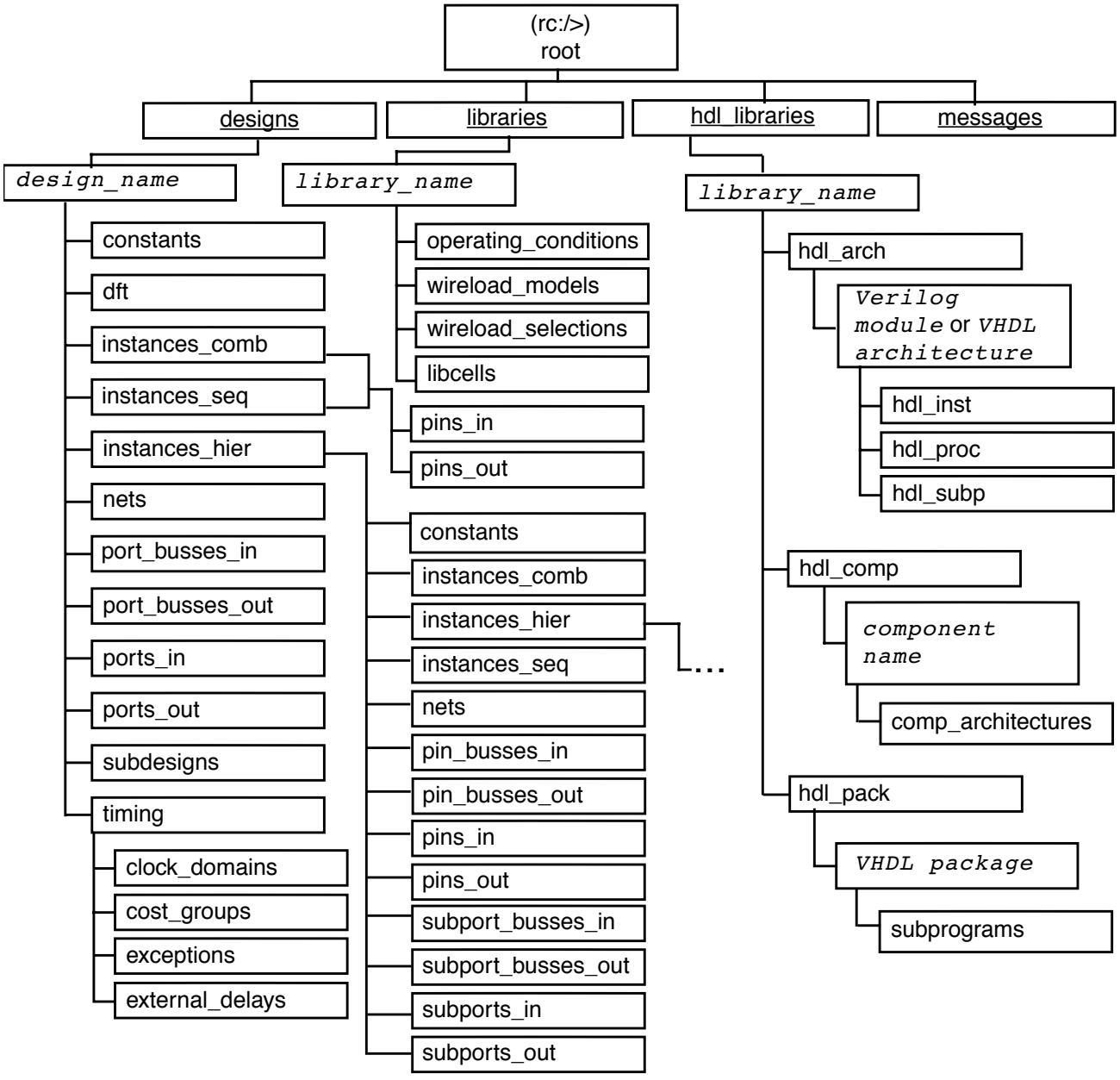
3

RTL Compiler Design Information Hierarchy

Overview

The Design Information Hierarchy contains the design data. When an RTL Compiler session is started, the basic information hierarchy is automatically created in memory. The top-level directories are empty before you load your designs and libraries. After the design and libraries are loaded and elaborated, new hierarchical levels are created within this hierarchy, as shown in Figure 3-1

Figure 3-1 Design Information Hierarchy



Setting the Current Design

All RTL Compiler operations are performed only on the current design. If you have only one top-level design, then RTL Compiler automatically treats this as the current design. If you have more than one top-level design, then you need to specify the current design.

➤ To set the current design, navigate to the top-level design in the design directory:

```
rc:/> cd /designs/top_level_design/
```

After you navigate to the directory of the design that you want to set as current, you can specify constraints or perform other tasks on that design. For example, to preserve the `FSH` subdesign from optimization, type the following command:

```
rc:/> cd /designs/SEQ_MULT/subdesigns/FSH
rc:.../FSH> set_attribute preserve true
```

Alternatively, you can use the `find` command to access the object, without changing the directory as follows:

```
rc:/> set_attribute preserve true [find / -subdesign FSH]
```

Specifying Hierarchy Names

You can control the hierarchy names that RTL Compiler implicitly creates for internally generated modules such as arithmetic, logic, and register-file modules.

➤ To specify the prefix for all implicitly created modules, type the following command:

```
rc:/> set_attribute gen_module_prefix name_prefix /
```

RTL Compiler uses the specified `gen_module_prefix` for all internally generated modules. By default, RTL Compiler does not add any prefix to internally generated modules. This attribute is valid only at the root-level (“/”).

Note: You must use this command before loading your HDL files.

Describing the Design Information Hierarchy

The following sections describe the hierarchy components and how they interact with each other.

See Navigating a Sample Design on page 75 for an example design.

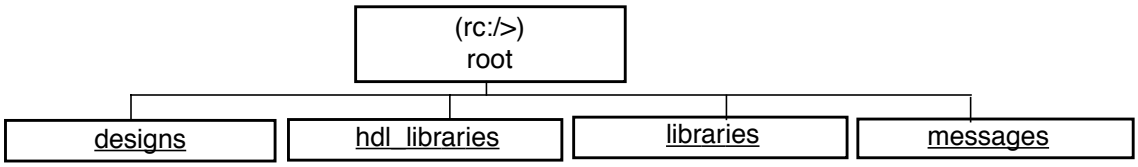
Note: In RTL Compiler, anything you can manipulate, such as designs, processes, functions, instances, clocks, or ports are considered “objects”.

- Working in the Top-Level (root) Directory on page 45
- Working in the Designs Hierarchy on page 47
- Working in the Library Directory on page 56
- Working in the hdl_libraries Directory on page 60

Working in the Top-Level (root) Directory

Root is a special object that contains all other objects represented as a ‘tree’ underneath it. The root object is always present in RTL Compiler and is represented by a “`/`”, as shown in Figure 3-2.

Figure 3-2 Top-Level Directory



➤ To quickly change to the root directory, type the `cd` command without any arguments:

```
rc:/designs/ksteal> cd
rc:/>
```

The top-level (root) directory of the RTL Compiler design data structure contains the following sub-directories:

- `designs`

  Contains all the designs and their associated components. This directory is populated and used *after* elaboration. See Working in the Designs Hierarchy on page 47 for detailed information.

- `hdl_libraries`

  Contains all the ChipWare, third party libraries, and designs. The design information is located under the `default` directory if the `-lib` option was not specified with the `read_hdl` command. Otherwise, the design information is located under the library specified with the `read_hdl` command. In either case, this directory is only available *before* elaboration.

  There are three directories under each `hdl_libraries` subdirectory. The three directories are: `architectures`, `components`, and `packages`.

  - `../architectures`

    If the design was described in Verilog, the `architectures` directory refers to the Verilog `module`. This directory contains all Verilog modules or VHDL architectures and entities that were read using the `read_hdl` command.

  - `.../components`

    Contains all ChipWare components added by RTL Compiler. Component information such as bindings, implementations, parameters, and pins can be found under this directory.

  - `.../packages`

    Contains all VHDL packages, and does not apply to Verilog designs.

    See Working in the hdl_libraries Directory on page 60 for detailed information.

    You can ungroup modules, including user defined modules, during elaboration in the `/hdl_libraries` directory. That is, you can control the Design Information Hierarchy immediately after loading the design. See Ungrouping Modules During and After Elaboration on page 64 for detailed information.

- `libraries`

  Contains all the specified technology libraries. See Working in the Library Directory on page 56 for detailed information.

- `messages`

  Contains all messages displayed during an RTL Compiler session.

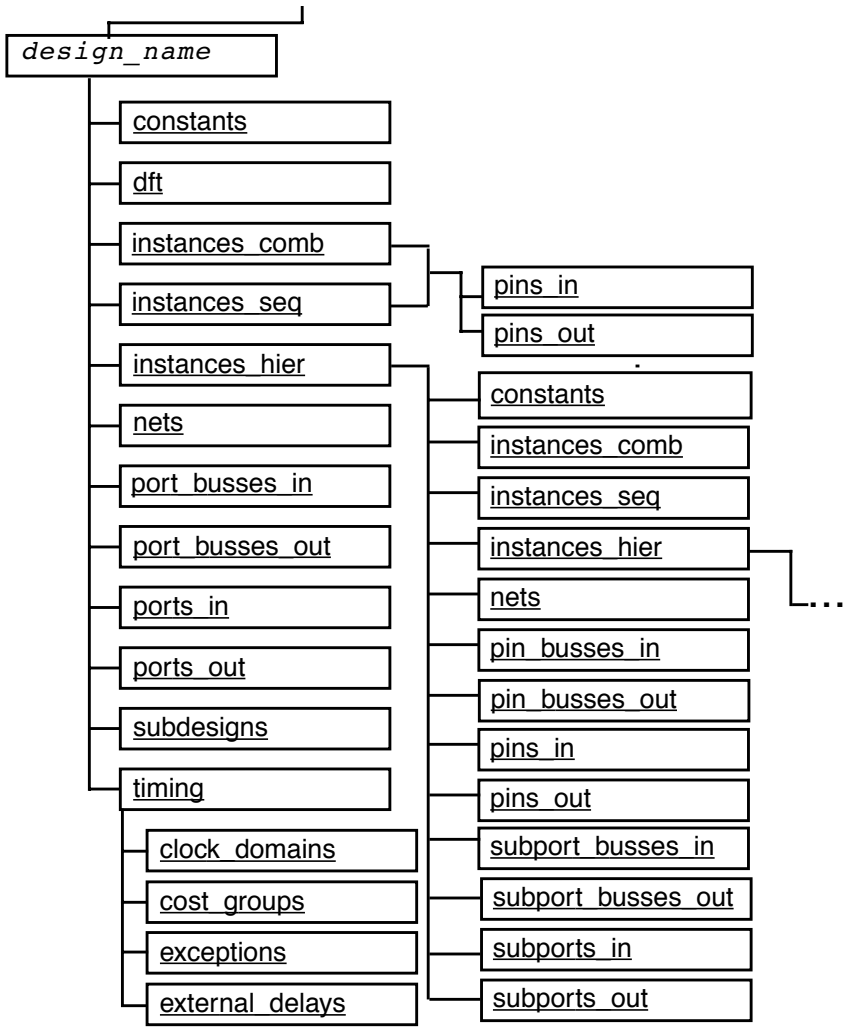
Working in the Designs Hierarchy

➤ Change your current location in the hierarchy to the `designs` directory:

```
rc:/> cd designs
```

A `design` corresponds to a module in Verilog that is not instantiated. In other words, it is the top-level Verilog module. During elaboration, the `designs` hierarchy is populated with the following components for each *design_name*, as shown in Figure 3-3.

Figure 3-3 Designs Hierarchy



The following describes the directories under the `designs` subdirectory.

- `constants`

Each level of hierarchy has its own dedicated logic constants that can only be connected to other objects within that level of hierarchy, such as `logic0` and `logic1` pins.The `logic0` and `logic1` pins are visible in the directory so that you can connect to them and disconnect from them. They are in the `constants` directory and are called `1` and `0`. The following example shows how the top-level `logic1` pin appears in a design called `add`:

```
/designs/add/constants/1
```

The following example shows how a `logic0` pin appears deeper in the hierarchy:

```
/designs/add/instances_hier/add_b/constants/0
```

- `dft` (Design for Test) contains the following subdirectories:
  - `report` contains two subdirectories:
    - `actual_scan_chains` contain the names and information pertaining to the final scan chains connected in the design.
    - `actual_scan_segments` contain the names and information pertaining to the final scan segments connected in the design.
  - `scan_chains` contain the names and information of any user-defined chains.
  - `scan_segments` contain the names and information of any user-defined segments.
  - `test_clock_domains` contain the names and information of any DFT test clocks, either identified by the DFT rule checker or defined by the user.
  - `test_signals` contain the names and information of any user-defined `shift_enable` and `test_mode` signals and also any `test_mode` signals that are detected and auto-asserted by the DFT rule checker.

  See “Attributes in the *Design for Test in Encounter RTL Compiler Ultra* for a list of all the DFT attributes per object type.

- An `instance` corresponds to a Verilog instantiation. There are four kinds of instances:
  - Instantiated subdesigns

    Instantiated subdesigns are hierarchical instances, which are in the `instances_hier` directory. The following is an example of an instantiated subdesign where `sub` is defined as a Verilog module:

    ```
    sub s(.in(in), .out(out));
    ```

    If this instantiation is performed directly inside of the `my_chip` module it would be listed in the design hierarchy as follows:

    ```
    /designs/my_chip/instances_hier/s
    ```

Identify the subdesign that an instance instantiates using the subdesign attribute. The above example would have its `subdesign` attribute set to the following:

```
/designs/my_chip/subdesigns/sub
```

- Instantiated primitives

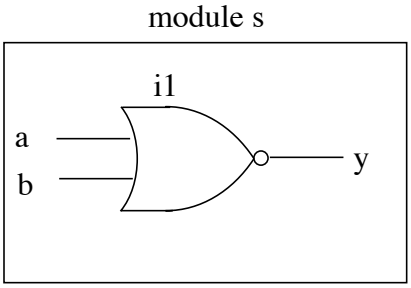
  Instantiated primitives are leaf level instances, which means that there are no instances beneath them. Instantiated primitives are in the `instances_comb` directory if they are combinational or in the `instances_seq` directory if they are sequential. Combinational means that the gate output is purely a function of the current values on inputs, such as a NAND gate or an inverter. Sequential means that the gate has some kind of internal state and typically a clock input, such as a RAM, flip-flop, or latch. The following example is an instantiated primitive, which uses `nor` as one of the special Verilog primitive function keywords:

  ```
  nor i1(a, b, c);
  ```

  If this instantiation is performed directly inside of module `s`, as shown in Figure 3-4, it would be listed in the design hierarchy as follows:

  ```
  /designs/my_chip/instances_hier/s/instances_comb/i1
  ```

Figure 3-4 Instantiated Primitive



- Instantiated library cells

  Instantiated library cells are also referred to as leaf level instances because there are no instances beneath them.These instances are in the `instances_comb` directory if they are combinational or in the `instances_seq` directory if they are sequential. The following is an example of an instantiated library cell, where `INVX1` is the name of a cell defined in the technology library:

  ```
  INVX1 i1(.A(w1), .Y(w2));
  ```

  If this instantiation is performed directly inside of module `s`, it would be listed in the design hierarchy as:

  ```
  /designs/my_chip/instances_hier/s/instances_comb/i1
  ```

  To view the libcell corresponding to a combinational or sequential instance, use the `get_attribute` command. For example:

```
get_attribute libcell /designs/..../il
```

- Unresolved references

  Unresolved references are also referred to as hierarchical instances, because usually a Verilog module is plugged in for them later in the flow. The following is an example of an unresolved reference, where `unres` is not in the library or defined as a Verilog module:

  ```
  unres u(.in(in), .out(out));
  ```

  If this instantiation is performed directly inside of module `s`, it would be listed in the design hierarchy as:

  ```
  /designs/my_chip/instances_hier/s/instances_hier/u
  ```

  In this case, querying the `unresolved` attribute on the instance would return a `true` value.

- `nets` refers to a `wire` in Verilog. If you have `wire w1;` within the `my_chip` module it would be listed in the design hierarchy as follows:

  ```
  /designs/my_chip/nets/w1
  ```

- `pins_in/pins_out` is a single 1-bit connection point on an instance. If you have the following pins:

  ```
  sub s(.in(in), .out(out))
  ```

  instantiated inside the `my_chip` module, and `in` is defined in module `s` as a bus with a `3:0` range and `out` is defined as a single bit, the following pins would be listed in the design hierarchy as:

  ```
  /designs/my_chip/instances_hier/s/pins_in/in[0]
  /designs/my_chip/instances_hier/s/pins_in/in[1]
  /designs/my_chip/instances_hier/s/pins_in/in[2]
  /designs/my_chip/instances_hier/s/pins_in/in[3]
  /designs/my_chip/instances_hier/s/pins_out/out
  ```

- `pin_bus` is a bussed connection point. Similar to the pin example above, the following `pin_busses` would be listed in the design hierarchy as:

  ```
  /designs/my_chip/instances_hier/s/pin_busses_in/in
  /designs/my_chip/instances_hier/s/pin_busses_out/out
  ```

  Note: If an instance connection point is not bussed because it is a single bit, it will still appear as a `pin_bus` object and a single pin object.

- `port` is a single 1-bit connection point on a design. If you have the following Verilog design:

  ```
  module my_chip(a, b, c, d);
  ```

```
input [2:0] a;
input b
...
```

This will produce ports and would be listed in the design hierarchy as follows:

```
/designs/my_chip/ports_in/a[0]
/designs/my_chip/ports_in/a[1]
/designs/my_chip/ports_in/a[2]
/designs/my_chip/ports_in/b
```

- `port_bus` represents all bussed input and output ports of a top-level design. For example, RTL Compiler displays the port and bus inputs in the `alu` design:

```
rc:/> ls -long /designs/alu/port_busses_in/

/designs/my_chip/ports_in/a[0]
/designs/my_chip/ports_in/a[1]
/designs/my_chip/ports_in/a[2]
/designs/my_chip/ports_in/b
```

Note: If an instance connection point is not bussed because it is a single bit, it will still appear as a `port_bus` object and a single port object.

- `subport` is a single bit-wise connection point within a module that has been instantiated. If module `sub` is defined in Verilog as follows:

```
module sub(in, out);
  input [1:0] in;
  output out;
```

and module `sub` is instantiated within the `my_chip` design as follows:

```
sub s(.in(in), .out(out))
```

then the following subports are listed in the design hierarchy as follows:

```
/designs/my_chip/instances_hier/s/subports_in/in[0]
/designs/my_chip/instances_hier/s/subports_in/in[1]
/designs/my_chip/instances_hier/s/subports_out/out
```

See Difference Between a Subport and a Pin on page 53 for more information.

- `subport_bus` are bussed connection points within a module. Similar to the above `subport` example, `subport_bus` objects are listed in the design hierarchy as follows:

```
/designs/my_chip/instances_hier/s/subport_busses_in/in
/designs/my_chip/instances_hier/s/subport_busses_out/out
```

Note: You will see the same list of signals in the pin, ports, and the `pin_busses`/`port_busses` directories if there are non-bussed connections. In the `subport` and the `subport_bus` example above, object `out` would appear in both directories

because there is both a `subport` called `out` and a `subport_bus` called `out`.

- `subdesigns` are Verilog modules that have been instantiated within another Verilog module. If the following instantiation appears within the `my_chip` module or recursively within any module that is instantiated within the `my_chip` module as follows:

  ```
  sub s(.in(in), .out(out));
  ```

  then the following subdesign object is listed in the design hierarchy as:

  ```
  /designs/my_chip/subdesigns/sub
  ```

  ➤ To list the instances that refer to (instantiate) a subdesign, use the `instances` attribute. For example, querying the `instances` attribute on the above subdesign will return a Tcl list that contains the following instance:

  ```
  /designs/my_chip/instances_hier/s
  ```

  It may also contain other instances if subdesign `s` was instantiated multiple times.

  See Subdesigns on page 77 for information on how to find `subdesigns` in the `design` data structure.

- `timing` contains the following timing and environment constraint subdirectories.

  ➤ To list all the object types in the hierarchy that you can set a constraint on, use the `set_attribute -help` command as follows:

  ```
  set_attribute -h *
  ```

  For example, the following command lists all the attributes that you can set on a `port`:

  ```
  set_attribute -h port *
  ```

  - `clock` refers to a defined clock waveform. Clock objects are created using the `define_clock` command.

    The SDC equivalent is the `create_clock -domain` command.

  - `clock_domain` refers to clocks that are grouped together because they are synchronous to each other, letting you perform timing analysis between these clocks. RTL Compiler only computes constraints among clocks in the same clock domain. By default, RTL Compiler assumes that every clock object belongs to a single clock domain. Create a `clock_domain` using the `define_clock` command or the SDC equivalent `create_clock` command. See Creating Clock Domains in *Setting Constraints and Performing Timing Analysis in Encounter RTL Compiler* for detailed information.

  - `cost_group` refers to a group of timing paths with a single timing optimization objective. During optimization, RTL Compiler tries to minimize the worst negative slack among all paths in each cost group. By default, all timing paths in a design are included in a cost group called `default`. As cost groups are created, the

corresponding signals are removed from the `default` group. Create cost groups using the `define_cost_group` command. Assign timing paths to a particular cost group using the `path_group` command, or the SDC equivalent `group_path` command. By default, RTL Compiler creates a separate cost group for each clock created using the SDC `create_clock` command. See Creating Path Groups and Cost Groups in *Setting Constraints and Performing Timing Analysis in Encounter RTL Compiler* for more information.

- `exceptions` refer to timing exceptions. A timing exception is a directive to indicate special treatment for a set of timing paths. Create timing exceptions using the following commands:
  - `multi_cycle` or the SDC equivalent `set_multicycle_path` command.
  - `path_adjust`
  - `path_delay` or SDC equivalent `set_max_delay` command.
  - `path_disable` or the SDC equivalent `set_false_path` command.
  - `path_group` or the SDC equivalent `group_path` command.
  - `define_cost_group`
  - `specify_paths`

  See Path Exceptions in *Setting Constraints and Performing Timing Analysis in Encounter RTL Compiler* for detailed information.

- `external_delays` refer to the delay between an input or output port in the design and a particular edge on a clock waveform, such as input and output delays. Create external delays using the `external_delay` command. See Defining External Clocks in *Setting Constraints and Performing Timing Analysis in Encounter RTL Compiler* for more information.

Difference Between a Subport and a Pin

It is important to understand the difference between a subport and a pin to manipulate the design hierarchy. A subport is used to define the connections with nets within a module and a pin is used to define the connections of the given module within its immediate environment. In the context of the top-level design, a hierarchical pin is like any other pin of a combinational or sequential instance. From the perspective of the given module, its `subports` are similar to ports through which it will pass and receive data.

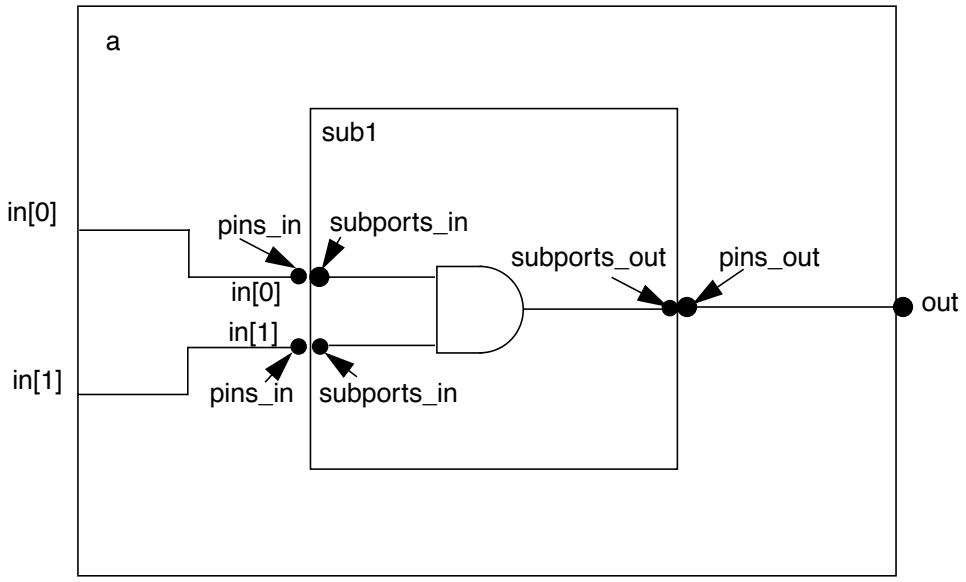
The following example shows the difference between a subport and a pin. Example 3-1 describes a Verilog design.

Example 3-1 Verilog Design

```
module a (in, out);
    input [1:0] in;
    output out;
    sub sub1 (.in1(in), .out1(out));
endmodule

module sub (in1, out1);
    input [1:0] in1;
    output out1;
    assign out1 = in1[0] && in1[1];
endmodule
```

Figure 3-5 shows a schematic representation of this example:

Figure 3-5 Schematic of Subports and Pins



During elaboration, RTL Compiler generates subports and pins in the design information hierarchy. As shown in Example 3-2, If you check the attributes for `subports_in` and `pins_in` for the `sub` module, you will see the following information:

Example 3-2 Subport and Pin Attributes in the Design Hierarchy

```
rc:/designs/a/instances_hier/sub1/subports_in> ls -a
Total: 3 items
./
in1[0]      (subport)
    Attributes:
     bus = /designs/a/instances_hier/sub1/subport_busses_in/in1
     direction = in
     net = /designs/a/instances_hier/sub1/nets/in1[0]
in1[1]      (subport)
    Attributes:
     bus = /designs/a/instances_hier/sub1/subport_busses_in/in1
     direction = in
     net = /designs/a/instances_hier/sub1/nets/in1[1]
rc:/designs/a/instances_hier/sub1/pins_in> ls -a
Total: 3 items
./
in1[0]      (pin)
    Attributes:
     direction = in
     net = /designs/a/nets/in[0]
in1[1]      (pin)
    Attributes:
     direction = in
     net = /designs/a/nets/in{1}
```

In particular, the nets are connected to `subports_in` and `pins_in`. The net connected to `subports_in/in1[0]` is defined at a level of hierarchy within the `sub1` hierarchical instance; hence, this net describes connections within the `sub` module. The net connected to `pins_in/in1[0]` is defined at the level of the module encapsulating `sub1`, in this case, the top level design (`/designs/a`). Therefore, this net describes the connections of the `sub` module and its environment.

This is similar to the behavior for `pins_out` and `subports_out`.

Working in the Library Directory

A library is an object that corresponds to a technology library, which appears in the `.lib` file as a `library` group as follows:

```
library("my_technology") {}
```

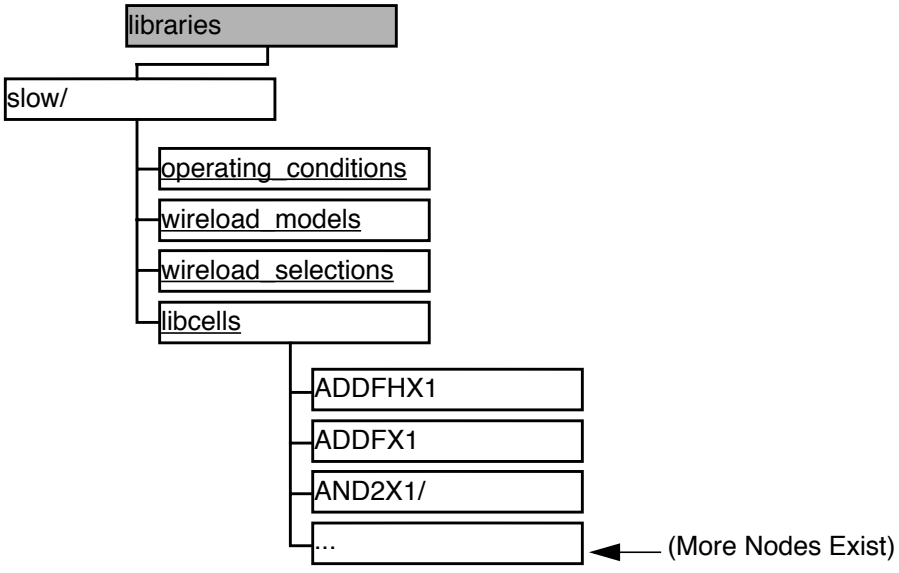
The technology library is listed in the library directory as follows:

```
/libraries/my_technology
```

After you load the design and libraries, new hierarchical levels are created within this information hierarchy. For example, as shown in Figure 3-6, if you look into the libraries directory using the `cd` command and list the contents using the `ls -long` command, there is only one library (`slow`) in the /`libraries` directory:

```
rc: cd libraries/
rc:/libraries> ls -l
Total: 2 items
./
slow/    (library)
```

Figure 3-6 Library Directory Example



If you change your directory into this library using the `cd` command and list the contents using the `ls -long` command, the following contents are listed:

```
Total: 5 items
./    (library)
```

```
libcells/
operating_conditions/
wireload_models/
wireload_selections/
```

RTL Compiler creates the library structure with the following subdirectories and fills in their associated information:

- `libcells/`

  Library cells and their associated attributes that RTL Compiler uses during mapping and timing analysis.

  - `libarc` corresponds to a timing path between two pins of a library cell. In the technology `.lib` file this appears as a `timing` group:

    ```
    timing() {}
    ```

  - `libpin` corresponds to a library pin within a library cell. It appears in the `.lib` file as a `pin` group as follows:

    ```
    pin("A") {}
    ```

    This may produce an object as follows:

    ```
    /libraries/my_technology/libcells/INVX1/A
    ```

➤ To get detailed information about a pin or cell, use the `ls` command with the `-long` and `-attribute` options:

```
rc:/> ls -l -a cell_name/pin_name
```

RTL Compiler displays the functionality (how the pin value is assigned), timing arcs in reference to other pins, and other data.

For example, the following command displays data about pin `Y`.

```
rc:/libraries/slow/libcells/NAND4BBX1> ls -l -a Y
```

This displays the following information:

```
Total: 2 items
./        (libpin)
    All attributes:
        async_clear = false
        async_preset = false
        clock = false
        enable = false
        fanout_load = 0 fanout_load units
        function = (!(AN' BN' C D))
        higher_drive = /libraries/slow/libcells/NAND4BBX2/Y
        incoming_timing_arcs = 4
```

```
        input = false
        load = 0.0 ff
        lower_drive = /libraries/slow/libcells/NAND4BBXL/Y
        outgoing_timing_arcs = 0
        output = true
tristate = false
    Additional information:
        inarcs/
```

The timing arcs directory (`inarcs`) contains the timing lookup table data from the technology library that RTL Compiler uses for timing analysis.

For a library cell, RTL Compiler displays the area value, whether the cell is a flop, latch, or tristate cell, and whether it is prevented from being used during mapping.

For example:

```
Total: 6 items
./        (libcell)
    All attributes:
        area = 26.611
        avoid = false
        timing_model = false
        buffer = false
        combinational = true
        flop = false
        inverter = false
        latch = false
        preserve = false
        sequential = false
        tristate = false
        usable = true
        …
```

➤ To get more information on any library cell (for example, the `NAND4BBXL` library cell), `cd` into the directory and list its contents:

```
rc:/libraries/slow/libcells> cd NAND4BBXL/
rc:/libraries/slow/libcells/NAND4BBXL> ls -l
```

This displays information similar to the following:

```
Total: 6 items
./    (libcell)
AN/    (libpin)
BN/    (libpin)
```

```
C/    (libpin)
D/    (libpin)
Y/    (libpin)
```

- `operating_conditions/`

  Operating conditions for which the technology library is characterized.

- `wireload_models/`

  The available wire-load models in the technology library.

  See Finding and Listing Wire-Load Models on page 71 for information on finding and listing library wire-load model specifications.

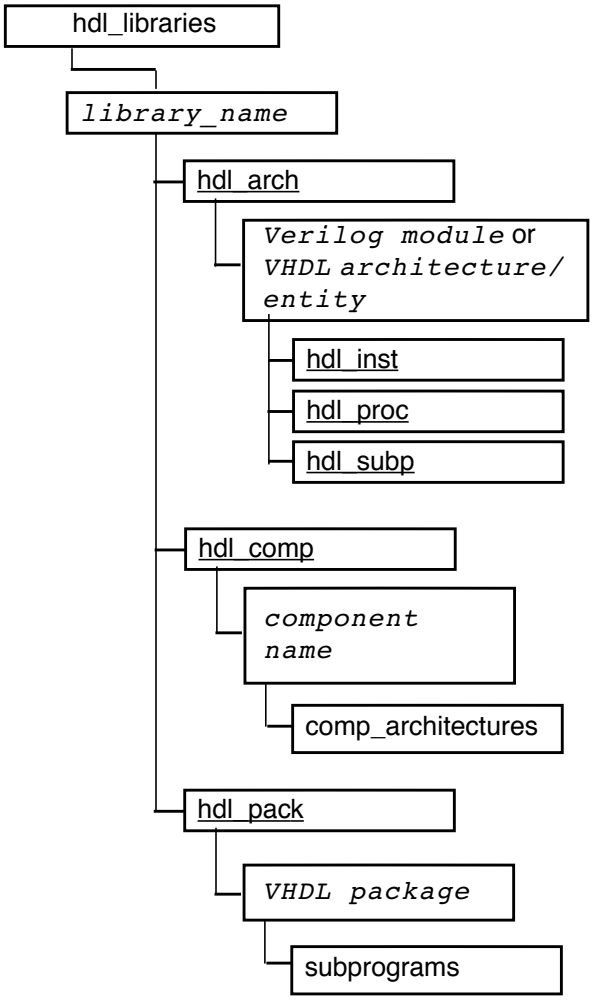
- `wireload_selections/`

  The user selected wire-load models.

Working in the hdl_libraries Directory

The `hdl_libraries` directory contains the following object types, as shown in Figure 3-7.

Figure 3-7 hdl_libraries Directory



➤ To get a list of the HDL library directories, type the `ls` command in the directory. For example:

```
rc:/hdl_libraries> ls
Total: 10 items
./
AWARITH/    (hdl_lib)
AWLOGIC/    (hdl_lib)
AWSEQ/      (hdl_lib)
CW/         (hdl_lib)
```

```
DW01/          (hdl_lib)
DW02/          (hdl_lib)
DW03/          (hdl_lib)
DWARE/         (hdl_lib)
GTECH/         (hdl_lib)
```

The (`hdl_lib`) to the right of each directory indicates the library type. The following is a complete list of HDL library object types:

- `hdl_lib`

  Refers to the HDL libraries in the directory named:

  `/hdl_libraries`

- `hdl_arch`

  Refers to the VHDL architecture and Verilog module in the directory named:

  `/hdl_libraries/`*`library_name`*`/architectures`

  VHDL architectures are named using an *`entityname`* (*`architecture_name`*) convention while Verilog modules are named using a *`modulename`* convention.

  - `hdl_inst`

    Refers to the HDL instance in the directory named:

    `/hdl_libraries/library_name/architectures/`
    `module_or_architecture name``/instances`

  - `hdl_proc`

    Refers to the HDL process in the directory named:

    `/hdl_libraries/default/architectures/`
    `module_or_architecture name``/processes`

    Unnamed processes are named using a *noname@linesourcelinenumber* naming convention.

  - `hdl_subp`

    Refers to the HDL subprogram in the directory named:

    `/hdl_libraries/default/architectures/`
    `module_or_architecture name``/subprograms`

    Overloaded subprograms are named using a *functionname@linesourcelinenumber* naming convention. Overloaded

subprograms are widely used subprograms that perform similar actions on arguments of different types, as shown in Example 3-3

Example 3-3 Overloaded Subprograms

```
-- Id: A.3
function "+" (L, R: UNSIGNED) return UNSIGNED;
-- Result subtype: UNSIGNED(MAX(L'LENGTH, R'LENGTH)-1 downto 0).
-- Result: Adds two UNSIGNED vectors that may be of different lengths.

-- Id: A.4
function "+" (L, R: SIGNED) return SIGNED;
-- Result subtype: SIGNED(MAX(L'LENGTH, R'LENGTH)-1 downto 0).
-- Result: Adds two SIGNED vectors that may be of different lengths.

-- Id: A.5
function "+" (L: UNSIGNED; R: NATURAL) return UNSIGNED;
-- Result subtype: UNSIGNED(L'LENGTH-1 downto 0).
-- Result: Adds an UNSIGNED vector, L, with a non-negative INTEGER, R.

-- Id: A.6
function "+" (L: NATURAL; R: UNSIGNED) return UNSIGNED;
-- Result subtype: UNSIGNED(R'LENGTH-1 downto 0).
-- Result: Adds a non-negative INTEGER, L, with an UNSIGNED vector, R.

-- Id: A.7
function "+" (L: INTEGER; R: SIGNED) return SIGNED;
-- Result subtype: SIGNED(R'LENGTH-1 downto 0).
-- Result: Adds an INTEGER, L(may be positive or negative), to a SIGNED
-- vector, R.

-- Id: A.8
function "+" (L: SIGNED; R: INTEGER) return SIGNED;
-- Result subtype: SIGNED(L'LENGTH-1 downto 0).
-- Result: Adds a SIGNED vector, L, to an INTEGER, R.
```

- `hdl_comp`

  Refers to the ChipWare component in the directory named:

  `/hdl_libraries/`*`library_name`*`/components/`

- `hdl_impl`

  Refers to the ChipWare component architecture in the directory named:

  `/hdl_libraries/`*`library_name`*`/components/`*`component_name`*`/comp_architectures`

- `hdl_pack`

  Refers to the VHDL package in the directory named:

  `/hdl_libraries/`*`library_name`*`/packages`

To learn how to find information about an HDL object, see Finding Specific Objects and Attribute Values on page 70.

Manipulating Objects in the Design Information Hierarchy

Attributes exist on each object type so that you can manipulate your design before elaboration. Refer to the *Encounter RTL Compiler Ultra Attribute Reference* for a complete list. You can also ungroup modules during and after elaboration, and you can use Tcl commands to manipulate objects in the Design Information Hierarchy.

Ungrouping Modules During and After Elaboration

Ungrouping Modules During Elaboration

You can ungroup modules, including user defined modules, during elaboration in the `/hdl_libraries` directory, which lets you control the Design Information Hierarchy immediately after loading the design. The `/hdl_libraries` directory contains specific object types that correlate to particular data. The following lists and describes the object types related to modules in this directory:

- `hdl_comp` — An RTL Compiler or other tool defined component
- `hdl_impl` — An architecture of a RTL Compiler or other tool defined component
- `hdl_arch` — A user defined module
- `hdl_inst` — An instance of a user defined module, a RTL Compiler or other tool defined component

By default, RTL Compiler does *not* ungroup user defined modules, ChipWare components, DesignWare components and GTECH components.

A user defined module is ungrouped during elaboration if either:

- The `ungroup` attribute is set to `true` on the particular `hdl_arch` module before the `elaborate` command is used

  For example, the following command specifies that all instances of the `foo` module should be flattened during elaboration:

  ```
  rc:/> set_attribute ungroup \
      true /hdl_libraries/default/architectures/foo/
  ```

- The `ungroup` attribute is set to `true` on the particular `hdl_inst` instance before the `elaborate` command is used

  For example, the following command specifies that `inst1` should be inlined during elaboration:

```
rc:/> set_attribute ungroup \
    true /hdl_libraries/default/architectures/foo/instances/inst1
```

A particular tool defined component is ungrouped during elaboration if either:

- The `ungroup` attribute is set to `true` on the particular `hdl_comp` component before using the `elaborate` command.

  For example, the following command ungroups all instances of a tool defined component during elaboration:

  ```
  rc:/> set_attribute ungroup true [find / -hdl_comp $component_name]
  ```

- The `ungroup` attribute is set to `true` on the particular `hdl_impl` architecture before using the `elaborate` command.

  For example, the following command ungroups all instances of a user defined module during elaboration:

  ```
  rc:/> set_attribute ungroup true [find / -hdl_arch $module_name]
  ```

- The `ungroup` attribute is set to `true` on the `hdl_inst` instance before using the `elaborate` command.

  For example, the following command ungroups a particular instance during elaboration:

  ```
  rc:/> set_attribute ungroup true [find / -hdl_inst $instance_name]
  ```

To potentially facilitate more carrysave transformation around arithmetic ChipWare components, ungroup components like `CW_add`, `CW_sub`, `CW_addsub`, `CW_inc`, `CW_dec`, `CW_incdec`, `CW_mult`, `CW_square` and so forth during elaboration. For example, the following command ungroups the `CW_add` component during elaboration:

```
rc:/> set_attribute ungroup true [find / -hdl_comp CW_add]
```

Ungrouping Modules After Elaboration

Note: Ungrouping can only be done on instances.

To ungroup all implicitly created modules, follow these steps:

1. Set the desired module prefix for RTL Compiler created modules by typing:
   ```
   rc:/> set_attribute gen_module_prefix CDN_DP_ /
   ```
2. Read in the Verilog files by typing:
   ```
   rc:/> read_hdl files
   ```
3. Elaborate (build) the design by typing:
   ```
   rc:/> elaborate
   ```
4. Specify your constraints.
5. Synthesize the design by typing:
   ```
   rc:/> synthesize -to_mapped
   ```
6. Ungroup the generated modules before writing out the design by typing:
   ```
   rc:/> set akk)sybdes [fubd /des* -sybdes CDN-DP]
         foreach sub_des $all_subdes {
         set inst [get_attr instances $sub-des]
         edit_netlist ungroup $inst
   }
   ```

Finding Information in the Design Information Hierarchy

There a number of ways to find information in the design data structure, including:

Using the cd Command to Navigate the Design Information Hierarchy

Use the `cd` command to navigate to different levels of the directory. There are no options to `cd`.

Tip

When navigating, you do not need to type the complete directory or object name. You can type less by using the '*' wild card character, such as the following:

```
rc:/> cd des*
```

You can also use the `Tab` key to complete the path for you, as long as the characters you have typed uniquely identify a directory or object. For example, the following command will take you from the root directory to the `subdesigns` directory:

```
rc:/> cd sub <TAB>
rc:/designs/ksteal/subdesigns>
```

Using the ls Command to List Directory Objects and Attributes

Use the `ls` command to list directory objects and view their associated attributes.

See Using the ls Command vs. the get_attribute Command on page 72 to learn the difference in using these two commands.

- To view directory names and any other object in the current directory:

```
rc:/> ls
```

- To list all the contents in the long format:

```
rc:/> ls -long
```

or, use the equivalent shortcut command:

```
rc:/> ls -l
```

- To list the contents of the current directory and the associated attributes:

```
rc:/> ls -attribute
```

or, use the equivalent shortcut command:

```
rc:/> ls -a
```

The following is an example of the information displayed with the `-attribute` option:

```
rc:/> ls -attribute /designs/alu/subdesigns/
/designs/alu/subdesigns:
Total: 2 items
./
addinc65/    (subdesign)
    Attributes:
        instances = /designs/alu/instances_hier/ops1_add_25
        logical_hier = false
        speed_grade = very_fast
        user_name = addinc
        wireload = /libraries/tutorial/wireload_models/AL_MEDIUM
```

Note: Using the `ls -a` command will show only the attributes that have been set. To see a complete list of attributes:

```
ls -a -l
```

or

```
ls -la
```

- To list the contents of the `designs` directory in the long format:

```
rc:/designs> ls -long
```

RTL Compiler displays information similar to the following:

```
rc:/> ls -long /designs/alu/port_busses_in/
/designs/alu/port_busses_in:
Total: 7 items
./
accum        (port_bus)
clock        (port_bus)
data         (port_bus)
ena          (port_bus)
opcode       (port_bus)
reset        (port_bus)
```

➤ To list all computed attributes (computed attributes are potentially very time consuming to process and are therefore not listed by default):

```
rc:/designs> ls -computed
```

RTL Compiler displays information similar to the following:

```
rc:/designs> ls -computed
Total: 2 items
./
MOD69/    (design)
    Attributes:
        area = 0.000 library provided units
        cell_area = 0.000 library provided units
        cell_count = 8 library provided units
        lp_leakage_power = 0.000 nW
        net_area = 0.000 library provided units
        slack = no_value picoseconds
        tns = 0.000
        wireload = /libraries/slow/wireload_models/GNUTELLA18_Conservative
```

Using the find Command to Search for Information

The `find` command in RTL Compiler behaves similarly to the `find` command in UNIX. Use this command to search for information from your current position in the design hierarchy, to find specific objects and attribute values, and to find and list wire-load models.

Use the `find` command to extract information without changing your current position in the design data structure.

➤ To search from the root directory, use a slash (“/”) as the first argument:

```
rc:/> find / ...
```

This search begins from the root directory then descends all the subdirectories.

- To start the search from the current position, use a period (.):

  ```
  rc:/> find . ...
  ```

  This search begins from the current directory and then descends all its subdirectories.

- To find hierarchical objects, you can just specify the top-level object instead of the root or current directory. Doing so can provide faster results because it minimizes the number of hierarchies that RTL Compiler traverses. In the following example, if we wanted to only find the output pins for `inst1`, the first specification is more efficient than the second. The second example not only traverses more hierarchies, it also returns `inst2` instances.

  ```
  rc:/> find inst1 -pin out*
  {/designs/woodward/instances_hier/inst1/pins_out/out1[3]}


  rc:/>find / -pint out*
  {/designs/woodward/instances_hier/inst1/pins_out/out1[3]}
  {/designs/MOD69/instances_hier/inst2/pins_out/out1[3]}
  ```

Finding Top-Level Designs, Sub-Designs, and Libraries

- The `find` command can also search for the top-level design names with the `-design` option:

  ```
  rc:/> find / -design *

  /designs/SEQ_MULT
  ```

- To see all the sub-designs below the top-level design (`SEQ_MULT` in this example), type the following command:

  ```
  rc:/> find / -subdesign *
  ```

  In this example `SEQ_MULT` has four subdesigns:

  ```
  /designs/SEQ_MULT/subdesigns/cal
  /designs/SEQ_MULT/subdesigns/chk_reg
  /designs/SEQ_MULT/subdesigns/FSM
  /designs/SEQ_MULT/subdesigns/reg_sft
  ```

- To find the GTECH libraries, type the following command:

  ```
  rc:/> find / -hdl_lib GTECH
  ```

Finding Specific Objects and Attribute Values

- Find particular objects using the `find` command with the appropriate object type. For example, the following example searches for the ChipWare libraries:

  ```
  rc:/> find / -hdl_lib CW
  ```

➤ To find the `CW_add` ChipWare component, type the following command:

```
rc:/> find / -hdl_comp CW_add
```

The following example searches for all the available architectures for the `CW_add` ChipWare component:

```
rc:/> find / -hdl_impl CW_add/*
```

or:

```
find [find /-hdl_comp CW_add] -hdl_impl *
```

➤ Find information, such as object types, attribute values, and location using the `ls -long -attribute` command. For example, using this command on the `CW_add` component returns the following information:

```
rc:/hdl_libraries/CW/components> ls -long -atrribute /hdl_libraries/CW/ \
    components/CW_add
/hdl_libraries/CW/components/CW_add:
Total: 2 items
./    (hdl_comp)
    All attributes:
        avoid = false
        location = /home/toynbee/tools.sun4v/lib/chipware/syn/CW
        ungroup = false
        comp_architectures/
```

Finding and Listing Wire-Load Models

Use the `find` command to locate and list the specifications of the library wire-load models.

➤ To find all the `wireload_models`, type the following command:

```
rc:/> find / -wireload *
```

The command displays information similar to the following:

```
/libraries/slow/wireload_models/ForQA /libraries/slow/wireload_models/
CSM18_Conservative /libraries/slow/wireload_models/CSM18_Aggressive
```

Note: If there are multiple libraries with similar wire-load models or cell names, specify the library name that they belong to before specifying any action on those objects. For example, to list the wire-load models in only the `slow` library, type the following command:

```
rc:/> find /libraries/slow -wireload *
```

Using the get_attribute Command to Display an Attribute Value

Use the get_attribute command to display the current value of any attribute that is associated with a design object. You must specify which object to search for when using the get_attribute command.

- The following command retrieves the setting of the instances attribute from the subdesign FSH:

```
rc:/> get_attribute instances [find / -subdesign FSH]

/designs/SEQ_MULT/instances_hier/I1
```

- The following command finds the value for the attribute instances on the counter subdesign:

```
rc:/designs/design1/subdesigns> get_attribute instances counter

/designs/design1/instances_hier/I1
```

- When multiple design files are loaded, it may be difficult to correlate a module to the file in which it was instantiated. The following example illustrates how to find the Verilog file for a particular dasein submodule, using the get_attribute command.

```
rc:/> get_attribute arch_filename /designs/top/subdesign/dasein
```

The above command would return something like the following output, showing that the dasein submodule was instantiated in the file top.v:

```
../modules/intg_glue/rtl/top.v
```

Using the ls Command vs. the get_attribute Command

The following examples show the difference between using the ls command and the get_attribute command to return the wire-load model.

- The following example uses the ls -attribute command to return the wire-load model:

```
rc:/designs> ls -attribute
Total: 2 items
./
async_set_reset_flop_n/    (design)
    Attributes:
        dft_mix_clock_edges_in_scan_chains = false
        wireload = /libraries/slow/wireload_models/sartre18_Conservative
```

- The following example uses the get_attribute wireload command:

```
rc:/designs> get_attribute wireload /designs/async_set_reset_flop_n/
```

and returns the following wire-load model:

```
/libraries/slow/wireload_models/sartre18_Conservative
```

The `ls -attribute` command lists all user modified attributes and their values. The `get_attribute` command lists only the value of the specified attribute. The `get_attribute` command is especially useful in scripts where its returned values can be used as arguments to other commands.

- The following example involves returning information about computed attributes. Computed attributes are potentially very time consuming to process and are therefore not listed by default.

```
rc:/designs> ls -computed
Total: 2 items
./
MOD69/    (design)
    Attributes:
        area = 0.000 library provided units
        cell_area = 0.000 library provided units
        cell_count = 8 library provided units
        lp_leakage_power = 0.000 nW
        net_area = 0.000 library provided units
        slack = no_value picoseconds
        tns = 0.000
        wireload = /libraries/slow/wireload_models/GNUTELLA18_Conservative
```

While the `ls -computed` command lists all computed attributes, the `get_attribute` command will return information on a specific computed attribute.

```
rc:/designs> get_attribute area /designs/stormy/

106.444
```

Navigating a Sample Design

Figure 3-8 shows a sequential multiplier, `SEQ_MULT`. Figure 3-9 shows the design information hierarchy for the `SEQ_MULT` design. All of the following navigation examples and descriptions refer to this design.

See Describing the Design Information Hierarchy on page 45 for detailed descriptions.

Figure 3-8 SEQ_MULT Design

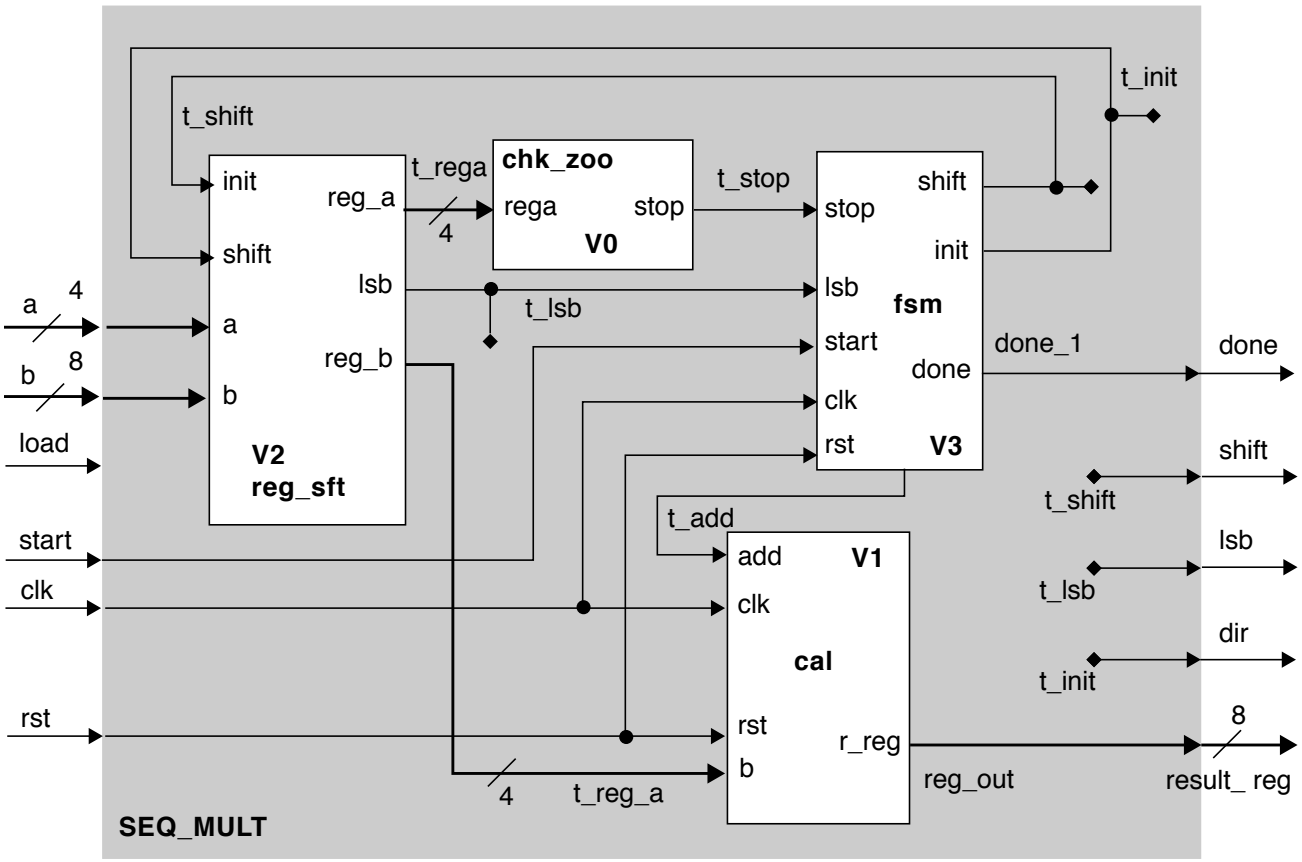
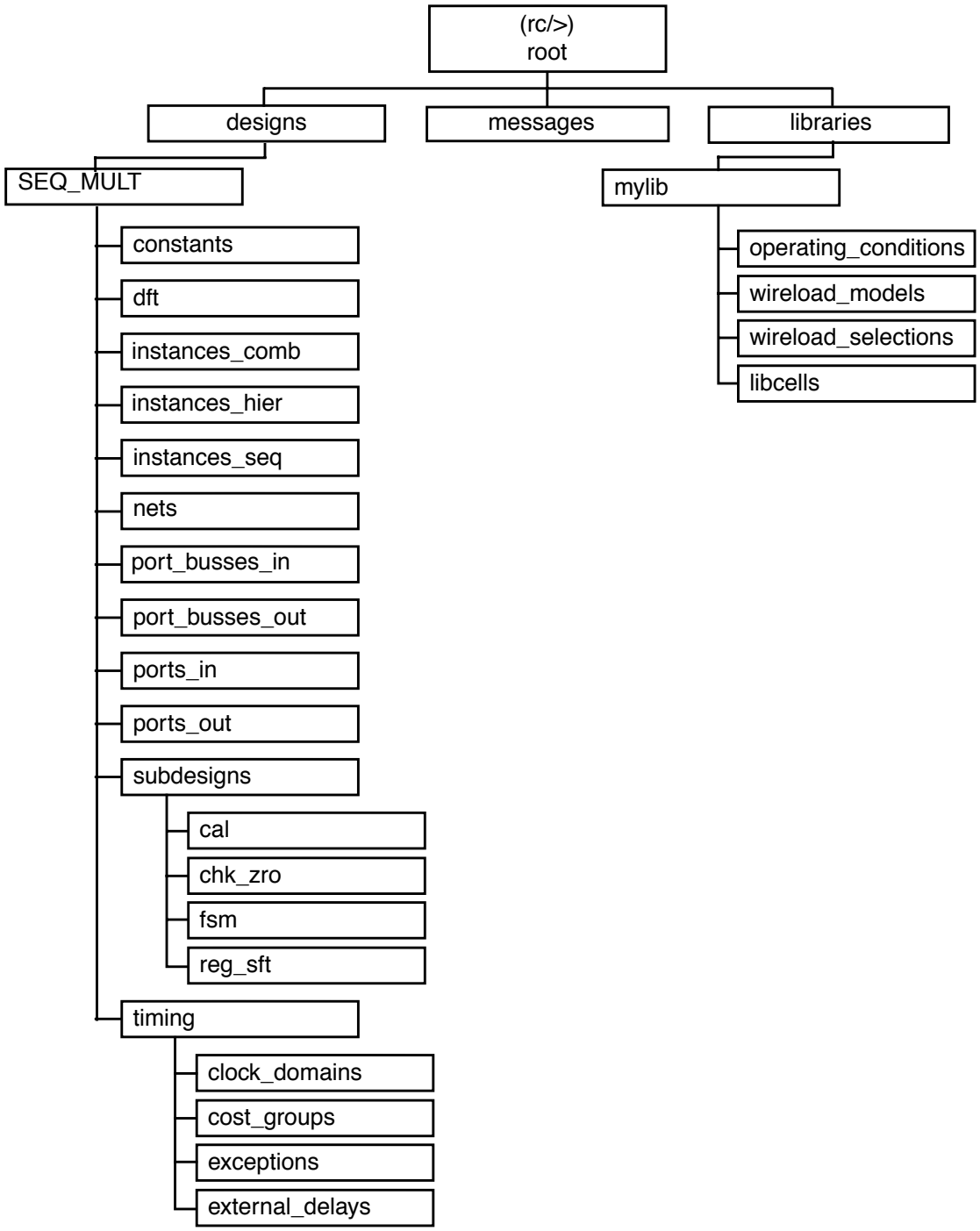


Figure 3-9 Top-Level View for SEQ_MULT Design



Subdesigns

The following commands find the `subdesigns` in the `SEQ_MULT` data structure:

```
rc:/> cd subdes*
rc:/designs/SEQ_MULT/subdesigns> ls -l
```

and returns the following:

```
Total: 3 items
./
cal/        (subdesign)
chk_zro/    (subdesign)
fsm         (subdesign)
reg_sft/    (subdesign)
```

See Subdesigns on page 77for a detailed description of this directory.

Input and Output Ports

To see the top level input and output ports, go to the `ports_in` or `ports_out` directories (Figure 3-9 on page 76).

- The following command finds the input and output ports with the `find` command:

  ```
  rc:/> find [find / -design SEQ_MULT] -port *
  ```

  The command displays information similar to the following:

  ```
  /designs/SEQ_MULT/ports_in/rst /designs/SEQ_MULT/ports_in/clk {/designs/
  SEQ_MULT/ports_in/a[3]} {/designs/SEQ_MULT/ports_in/a[2]} {/designs/SEQ_MULT/
  ports_in/a[1]} {/designs/SEQ_MULT/ports_in/a[0]} {/designs/SEQ_MULT/ports_in/
  b[7]} {/designs/SEQ_MULT/ports_in/b[6]} {/designs/SEQ_MULT/ports_in/b[5]} {/
  designs/SEQ_MULT/ports_in/b[4]} {/designs/SEQ_MULT/ports_in/b[3]} {/designs/
  SEQ_MULT/ports_in/b[2]} {/designs/SEQ_MULT/ports_in/b[1]} {/designs/SEQ_MULT/
  ports_in/b[0]} /designs/SEQ_MULT/ports_in/start /designs/SEQ_MULT/ports_in/
  load {/designs/SEQ_MULT/ports_out/result_reg[7]} {/designs/SEQ_MULT/
  ports_out/result_reg[6]} {/designs/SEQ_MULT/ports_out/result_reg[5]} {/
  designs/SEQ_MULT/ports_out/result_reg[4]} {/designs/SEQ_MULT/ports_out/
  result_reg[3]} {/designs/SEQ_MULT/ports_out/result_reg[2]} {/designs/
  SEQ_MULT/ports_out/result_reg[1]} {/designs/SEQ_MULT/ports_out/result_reg[0]}
  /designs/SEQ_MULT/ports_out/done /designs/SEQ_MULT/ports_out/shift /designs/
  SEQ_MULT/ports_out/lsb /designs/SEQ_MULT/ports_out/dir
  ```

See port in the Working in the Designs Hierarchy on page 47 for a detailed description of the `ports_in` and `ports_out` directories.

Hierarchical Instances

Hierarchical instances in the design are listed in the following directory:

```
/designs/SEQ_MULT/instances_hier/
```

The `/designs/SEQ_MULT/instances_hier` directory contains all the hierarchical instances in the `SEQ_MULT` top level design (see Figure 3-10).

Sequential Instances

Any sequential instances in the top-level design are listed in the `/designs/SEQ_MULT/instances_seq` directory shown in Figure 3-9. The `SEQ_MULT` design does not have any sequential instances at this level. However, it does have some at a lower level in the hierarchy, as shown in Figure 3-10.

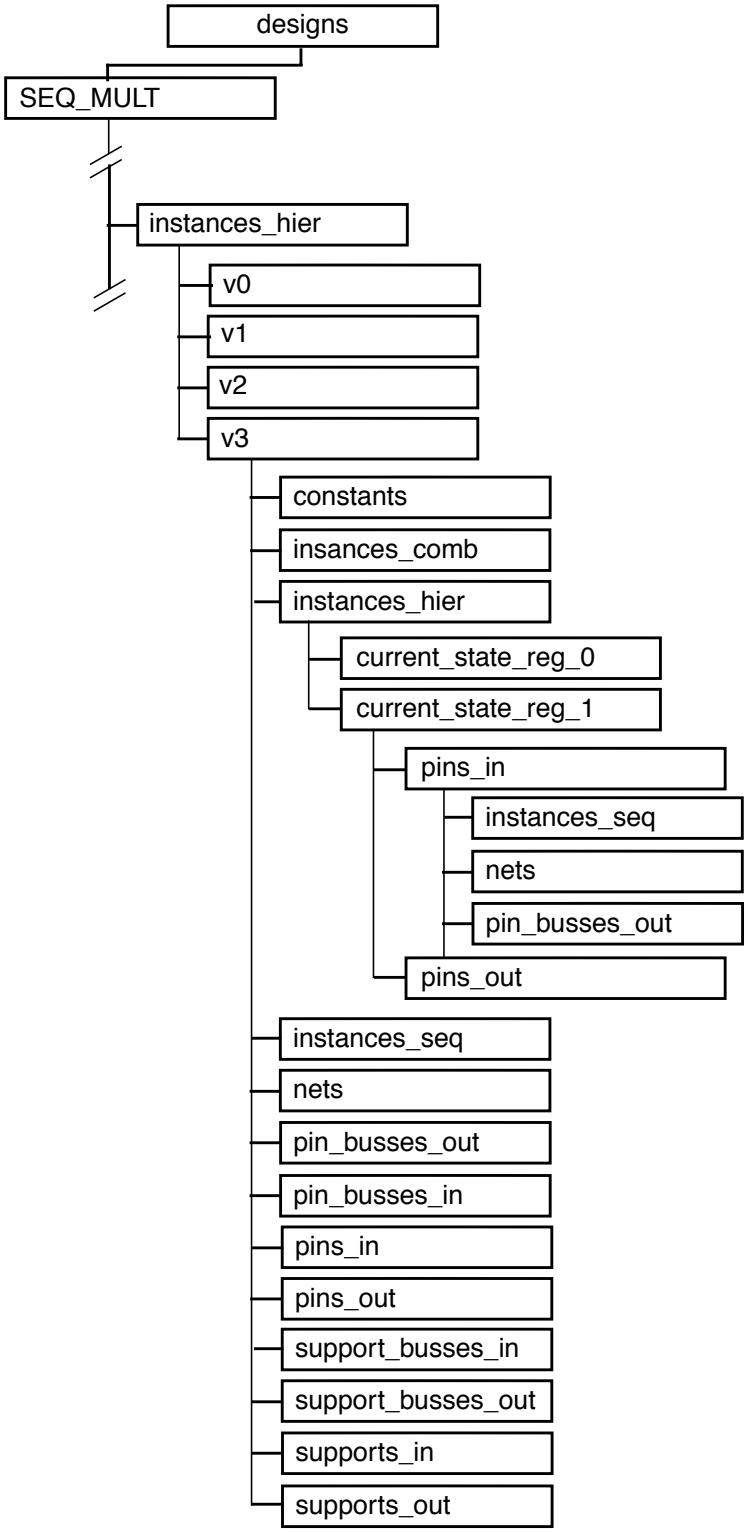
See `instances` for a detailed description of this directory.

Lower-level Hierarchies

Figure 3-10 on page 79 shows some of the lower level directories in the `SEQ_MULT` design.

The lower level directory structures are very similar to the `/designs/SEQ_MULT` contents. The design data structure is based upon the levels of design hierarchy and how the data is structured. Design information levels are created depending upon the design hierarchy.

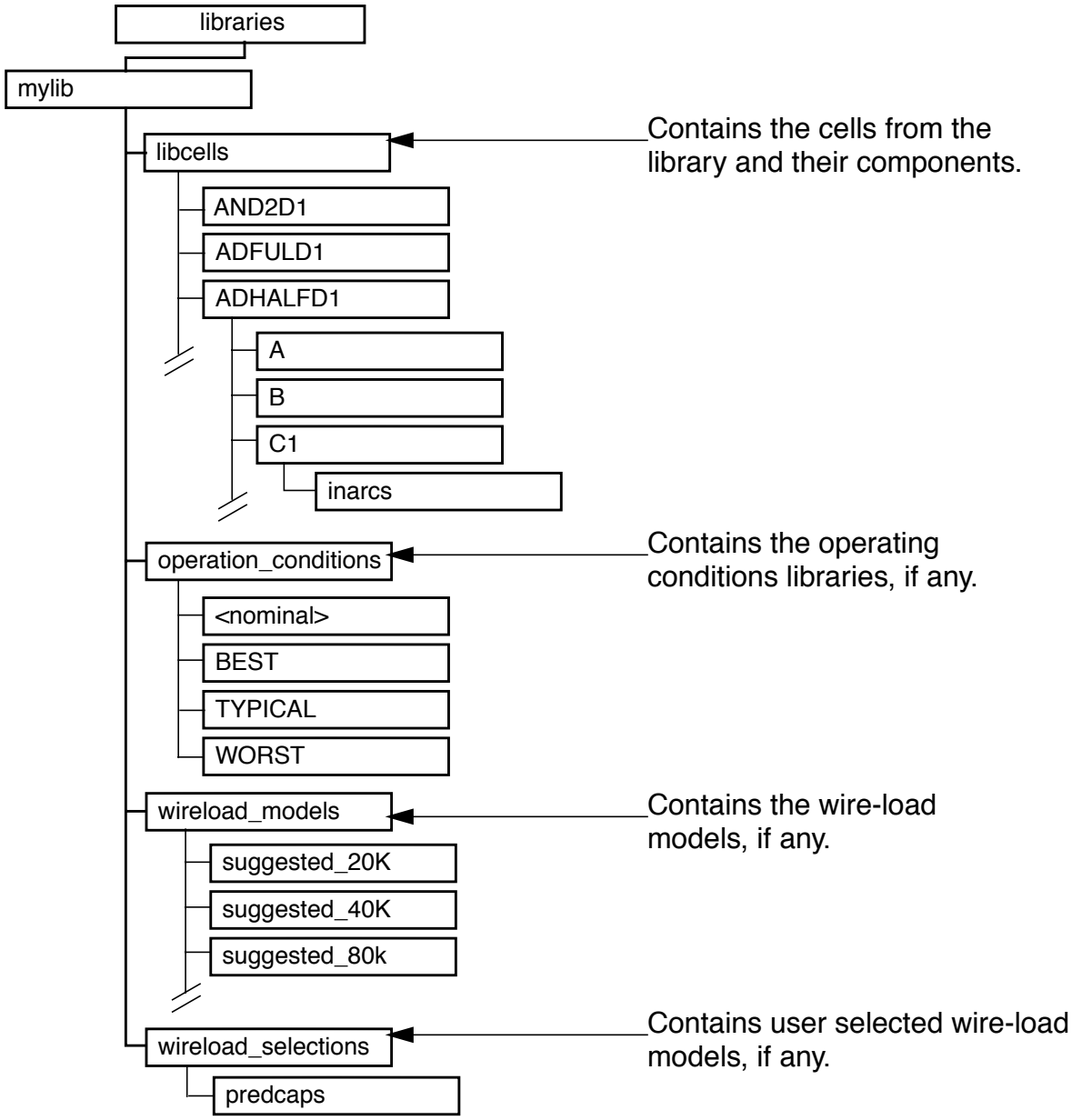
Figure 3-10 Low-Level for SEQ_MULT Design



Library Information

Figure 3-11 shows the `libraries` directory and its contents. The contents will vary with the design, but the following directories are always created for each library.

Figure 3-11 Libraries Directory Structure



See Working in the Library Directory on page 56 for a detailed description of this directory.

Tips and Shortcuts

The following are some helpful tips and shortcuts:

Accessing UNIX Environment Variables from RTL Compiler

You can access your UNIX variables while you are in an RTL Compiler session by using the following variable within RTL Compiler:

```
$env()
```

If you have a UNIX variable to indicate the `library` directory under the current directory, do the following steps:

1. In UNIX, store the path to the `library` directory to a variable. In this case, we use `LIB_PATH`:

   ```
   unix> setenv LIB_PATH ./library
   ```

2. In RTL Compiler, use the `$env` variable with the `lib_search_path` attribute:

   ```
   rc:/> set_attribute lib_search_path $env(LIB_PATH)
   ```

Working with Tcl in RTL Compiler

Using Tcl Commands to Manipulate Objects

Tcl lets you refer to objects using the following two methods: by using a handle to keep the pointer to the particular object, and by using the string name of that object.

Using a handle to keep the pointer to the object results in faster and more efficient manipulations. RTL Compiler takes advantage of this Tcl feature while manipulating objects in its database. Understanding how to use a handle only becomes important if you are writing Tcl scripts to interface with RTL Compiler.

In general, using the `find` command returns the string name of an object, whereas using Tcl list commands, such as `llindex` and `foreach` returns the handle.

For example, assume you have the following hierarchical instance in the database:

```
/designs/TOP/instances_hierarchical/some_instance
```

- To change the `some_instance` name to `some_instance_1`, use the following `set` Tcl command with the `find` command as follows:

  ```
  rc:/> set inst [find / -instance some_instance]

  /designs/TOP/instances_hier/some_instance
  ```

  then use the mv command to rename the instance in the design hierarchy as follows:

  ```
  rc:/> mv $inst [basename $inst]_1

  /designs/TOP/instances_hier/some_instance_1
  ```

  The `find` command returns the string in `$inst`. Therefore, using the `mv` command moves the object with the name stored in `$inst` to the new name. However, the `$inst` still contains the original name, which is listed when using the Tcl `puts` command as follows:

  ```
  rc:/> puts $inst

  /designs/TOP/instances_hierarchical/some_instance
  ```

- To store the updated name in `$inst`, use the Tcl `set` command with the `mv` command as follows:

  ```
  rc:/> set inst [mv $inst [basename $inst]_1]

  /designs/TOP/instances_hierarchical/some_instance_1
  ```

  Using the Tcl `puts` command shows the updated name in the design hierarchy as follows:

  ```
  rc:/> puts $inst

  /designs/TOP/instances_hierarchical/some_instance_1
  ```

- To access the “handle” for the object, use the following sequence:

  ```
  rc:/> set inst [lindex [find /des* -instance some_instance] 0]
  ```

Any further manipulation of the object referred by `$inst` will also change `$inst`. For example:

```
rc:>mv $inst [basename $inst]_1
```

```
/designs/TOP/instnaces_hier/some_instance_1
rc:/> puts $inst
rc:/> /designs/TOP/instances_hier/some_instance_1
```

You can also use a different method to update the content of the Tcl variable with the object being manipulated as follows:

```
rc:/> set inst [mv $inst [basename $inst]_1]
/designs/TOP/instances_hier/some_instance_1
rc:/> puts $inst
/designs/TOP/instances_hier/some_instance_1
```

The following examples explain how the pointer concept works differently from normal string manipulation.

Consider a design that has the following three instances:

```
/designs/TOP/instances_hier/some_instance_1
/designs/TOP/instances_hier/some_instance_2
/designs/TOP/instances_hier/some_instance_3
```

- To change the instance names to `some_instance_1_x`, `some_instance_2_x`, and `some_instance_3_x`, use the following steps:

```
rc:/> set list [find / -instance some_instance_*]
{/designs/TOP/instances_hier/some_instance_1
/designs/TOP/instances_hier/some_instance_2
/designs/TOP/instances_hier/some_instance_3}

rc:/> foreach elem $list {
=> mv $elem [basename $elem]_x
=> puts $elem
=>}
/designs/TOP/instances_hier/some_instance_1_x
/designs/TOP/instances_hier/some_instance_2_x
/designs/TOP/instances_hier/some_instance_3_x
```

Looking at the values of `$elem` that RTL Compiler returns, unlike the first method, the value of `$elem` changes to reflect the updated value of the particular instance name. This happens because the `foreach` command passes the handle to the objects in the `$list`. Therefore, `$elem` is the handle to an instance, not a string. Using the `mv` command modifies the value pointed to by the handle as well.

Likewise, when using the following command syntax:

```
rc:/> set elem [lindex [find / -instance some_instance_*] 0]
```

`$elem` still contains the handle to the instance as follows:

```
/designs/TOP/instances_hierarchical/some_instance_1
```

Be careful when using the `rm` ***command with the handle approach. As shown in the following example, when you remove the instance, the handle does not contain any values.***

```
rc:/> foreach elem $list {
=> rm $elem
=>puts $elem
=> }
objectdeleted
object_deleted
object_deleted
```

To reflect this, RTL Compiler stores the `object_deleted` string in the handle, which is similar to NULL stored in a pointer.

To refer only to objects as strings, or to avoid objects changing as a result of being moved (renamed) or deleted, use the `string_representation` command.

Comparing and Matching Strings in Tcl

There are separate Tcl commands to compare strings and match string patterns. The `string compare` command compares each character in the first string argument to each character in the second. The following example will return a “1” to indicate a difference in the first and second arguments:

```
string compare howisyourevening howisyournight
```

The `string match` Tcl command treats the first argument as a pattern, which can contain wildcards, while treating the second argument as a string. That is, `string match` queries if the specified *string* matches the specified *pattern*. The following example will return “`1`”:

```
string match howisyour* howisyourevening
```

Unless you want to perform pattern matching, do not use `string match`: one of the strings you want to match might contain a * character, which would give a false positive match.

Similarly, the == operator should only be used for numeric comparisons. For example, the following example is considered equivalent in Tcl:

```
rc:/> if {“3.0” == “3”} {puts equal}

equal
```

Instead of using == to compare strings, use the `eq` (equal) operator. For example:

```
rc:/> if {“howisyourevening” eq “howisyourevening”} {puts equal}

equal
```

The following example will not be equal when using the `eq` operator:

```
rc:/>if {“3.0” eq “3”} {puts equal}
```

The Backslash in Tcl

In Tcl, if the backslash (“\”) is used at the end of the line, the contents of the immediately proceeding line are inlined to the line ending in the backslash. For example:

```
rc:/> puts “This will be all\
==>on one line.”
This will all be on one line
```

This is Tcl’s idiosyncrasy, not RTL Compiler’s.

Using Command Line Keyboard Shortcuts

RTL Compiler supports keyboard shortcuts to access common command-line commands. The following table lists the supported keyboard shortcuts. The shortcuts are invoked by simultaneously pressing the control key and another corresponding key.

Table 3-1 Command-line Keyboard Shortcuts

`Control-a`	Go to the beginning of the line
`Control-b`	Move the cursor left
`Control-c`	Kills the current RTL Compiler process and exits RTL Compiler
`Control-d`	Delete the character under the cursor
`Control-e`	Go to the end of the line
`Control-f`	Move the cursor right
`Control-k`	Delete all text to the end of line
`Control-n`	Go to the next command in the history
`Control-p`	Go to the previous command in the history

Table 3-1 Command-line Keyboard Shortcuts, *continued*

`Control-z`	Instruct RTL Compiler to go to sleep. Control returns to the operating system.
up arrow	Previous command in history
down arrow	Next command in history

Using Command Abbreviations

To reduce the amount of typing, you can use abbreviations for commands as long as they do not present any ambiguity. For example:

Complete Command	Abbreviated Command
`multi_cycle –lenient`	`mu –le`

This abbreviation is possible because there is only one command that starts with "`mu`" which is `multi_cycle`, and there is only one reporting option that starts with "`le`" which is `lenient`.

In cases where there is ambiguity because a number of commands share the same character sequence, you only need to supply sufficient characters to resolve the ambiguity. For example, the commands `path_adjust` and `path_delay` both start with "path_". If you wanted to print out the help for these commands, you would need to abbreviate them as follows:

Complete Command	Abbreviated Command
`path_adjust -h`	`path_a -h`
`path_delay -h`	`path_d -h`

Using Command Completion with the Tab Key

You can use the Tab key to complete a command name after typing one or more letters. If there are several commands that start with that sequence of letters, pressing the Tab key fills in the command letters to the first letter that differs between commands. For instance, if you type the letters "pat" and then press the Tab key, the tool spells out "path_": If you type an ambiguous set of letters such as "re", the tool displays the commands that start with those letters:

```
rc:/> re
ambiguous "re": read read_hdl read_sdc reconnect_scan redirect regexp regress
regsub rename rename_cgs report return
```

Using Wildcards

RTL Compiler supports the * and ? wildcard characters:

- To specify a unique name in the design.

  For example, the following two commands are equivalent:

  ```
  rc:/> cd /designs/example1/constants/
  rc:/> cd /d*/example1/co*/
  ```

- To specify multiple design elements.

  For example, the following lists the contents of all directories that end with `out`:

  ```
  rc:/> ls *out
  ```

- To find a design with four characters:

  ```
  rc:/> find . -design ????
  ```

- To find a design with three characters that ends in an “i”:

  ```
  rc:/> find . -design ??i
  ```

The * and ? wildcard characters can also be used together.

Using Smart Searches

Smart searches allow you to find specific items of interest (instances, directories, and so on) without giving the entire hierarchical path name. There are two kinds of smart searches: instance-specific find and path search.

- Instance-specific find

In an instance specific find, instances are accessed without specifying container directories. For example, the following two commands refer to the same instance:

```
rc:/> cd des*/TOP/*/i0/*/i2/*/addinc_add_39_20_2/*/g160
rc:/> cd TOP/i0/i2/addinc_add_39_20_2/g160
```

The instance specific find feature is especially helpful when used with commands such as `report timing` and `get_attribute`. For example, the following two commands are equivalent:

```
rc:/> report timing -through des*/TOP/*/i0/*/i2/*/addinc_add_39_20_2/*/g160/*/Y
```

```
rc:/> report timing -through TOP/i0/i2/addinc_add_39_20_2/g160/Y
```

- Path Search

In a path search, objects are accessed by searching the custom-defined RTL Compiler design hierarchy paths. These paths are initially defined in the `.synth_init` file (see Introduction on page 19 for more information on the `.synth_init` file) but you may edit them at any time.

The default definition is as follows:

```
set_attribute -quiet path {
    .
    /
    /designs/*
    /designs/*/timing/clock_domains/*
    /libraries/*
 }
```

If you type the following command:

```
rc:/> ls alu*
```

RTL Compiler returns all matching items on the listed paths, for instance:

```
/designs/alu:
./                instances_hier/    port_busses_out/   timing/
constants/        instances_seq/     ports_in/
dft/              nets/              ports_out/
instances_comb/   port_busses_in/    subdesigns/
```

Saving the Design Information Hierarchy

There may be occasions in which you want to save the hierarchy, for example for backup purposes or to document the design. The following example shows how to save the hierarchy using Tcl and RTL Compiler commands:

Example 3-4 Saving the Design Information Hierarchy

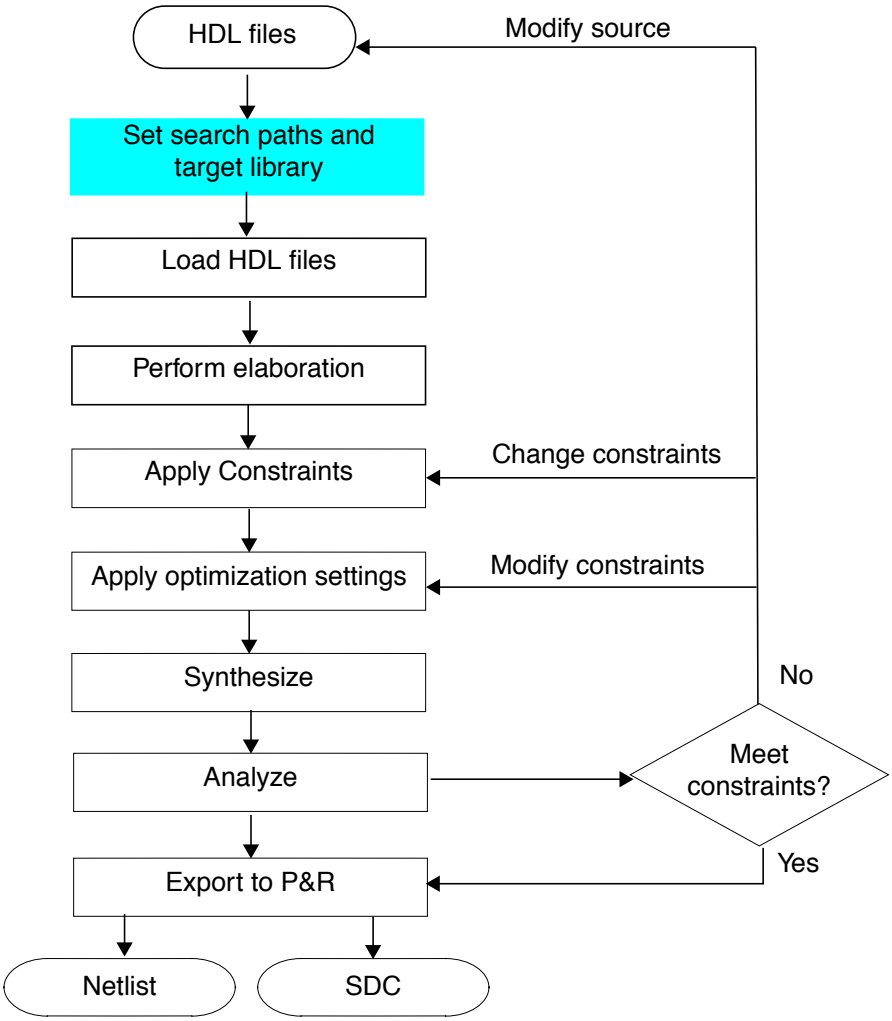
```
proc vdir_save {args} {
  set pov [parse_options [calling_proc] fil $args \
    "-detail bos include detailed info" detail \
    "drs root vdir from which to start saving data" vdir]

  switch -- $pov {
```

```
    -2 {return}
    0 {error "Failed on [calling_proc]"}
  }

  foreach x [lsort -dictionary [find $vdir * *]] {
    # simple data
    set data $x
    # detail data
      if {$detail} {
        redirect -variable data "ls -a $x"
      }
    puts $fil $data
  }

  if {![string equal $fil "stdout"]} {
    close $fil
  }
}
```

4

Using the Technology Library

Overview



Search paths are directory path names that RTL Compiler either explicitly or implicitly searches. This chapter explains how to set search paths and use the technology library.

Tasks

Specifying Explicit Search Paths

You can specify the search paths for libraries, scripts, and HDL files. The default search path is the directory in which RTL Compiler is invoked.

The host directory that contains the libraries, scripts, and HDL files are searched according to the values you specify for the following three attributes:

- `lib_search_path`

  The directories in the path are searched for technology libraries when you issue a `set_attribute library` command.

- `script_search_path`

  The directories in the path are searched for script files when you issue the `include` command.

- `hdl_search_path`

  The directories in the path are searched for HDL files when you issue a `read_hdl` command.

To set the search paths, type the following `set_attribute` commands:

```
rc:/> set_attribute lib_search_path path /
rc:/> set_attribute script_search_path path /
rc:/> set_attribute hdl_search_path path /
```

where *`path`* is the full path of your target library, script, or HDL file locations.

The slash (“/”) in these commands refers to the root-level RTL Compiler object that contains all global RTL Compiler settings.

If you want to include more than one entry for `path`, put all of them inside curly brackets {}. For example, the following command tells RTL Compiler to search for library files both in the current directory (.) and in the path `/home/customers/libs`:

```
rc:/> set_attribute lib_search_path {. /home/customers/libs}
```

To see all of the current settings, type:

```
ls -long -attribute /
```

The slash ("/") specifies the root-level.

Specifying Implicit Search Paths

Use the `path` attribute to specify the paths for implicit searches. Implicit searches occur with certain commands that require RTL Compiler to search the Design Information Hierarchy. Such searches, or finds, are not specified explicitly by the user, but rather is implied in the command.

In the following example, RTL Compiler recursively searches the specified paths and sets a false path between all clock objects named `clk1` and `clk2`.

```
rc:/> set_attribute path ". / /libraries/* /designs/*"
rc:/> dc::set_false_path -from clk1 -to clk2
```

RTL Compiler interprets the names `clk1` and `clk2` to be clock names because the inherent object search order of the SDC command `set_false_path` is clocks, ports, instances, pins. If there were no clocks named `clk1` or `clk2`, RTL Compiler would have interpreted the names to have been port names. If the `path` attribute is not specified, the default implicit search paths are:

```
. / /libraries/* /designs/* /designs/*/timing/clock_domains/*
```

Setting the Target Technology Library

After you set the library search path with the `lib_search_path` attribute, you need to specify the target technology library for synthesis using the `library` attribute.

➤ To specify a single library:

```
rc:\> set_attribute library lib_name.lbr /
```

RTL Compiler will use the library named `lib_name.lbr` for synthesis. RTL Compiler can also accommodate the `.lib` (Liberty) library format. In either case, ensure that you specify the library at the root-level ("/").

Note: If the library is not in a previously specified search path, specify the full path, as follows:

```
rc:\> set_attribute library /usr/local/files/lib_name.lbr
```

➤ To specify a single library compressed with gzip:

```
rc:/> set_attribute library lib_name.lbr.gz /
```

➤ To append libraries:

```
rc:\> set_attribute library {{lib1.lib lib2.lib}}
```

After `lib1.lib` is loaded, `lib2.lib` is appended to `lib1.lib`. This appended library retains the `lib1.lib` name.

Specifying Multiple Libraries

If your design requires multiple libraries, you must load them simultaneously. RTL Compiler uses the operating and nominal conditions, thresholds, and units from the first library specified. If you specify libraries sequentially, RTL Compiler uses only the last one loaded.

In the following example RTL Compiler uses only `lib_name2.lbr` as the target library:

```
rc:/> set_attribute library lib_name.lbr /
rc:/> set_attribute library lib_name2.lbr /
```

To specify multiple libraries using the `library` variable:

1. Define the `library` variable to include both libraries:

   ```
   rc:/> set library {lib_name1.lbr lib_name2.lbr}
   ```

   When listing files, use the Tcl list syntax: `{entry entry ...}`.

2. Set the `library` attribute to `$library`:

   ```
   rc:/> set_attribute library $library /
   ```

To specify multiple libraries by specifying all of the library names:

➤ Type both libraries with the `set_attribute` command, as shown:

```
rc:/> set_attribute library { lib_name.lbr lib_name2.lbr } /
```

To specify multiple libraries while appending some libraries to others:

➤ Separate appended libraries with braces:

```
rc:/> set_attribute library {{lib1.lib lib2.lib} lib3.lib}
```

After `lib1.lib` is loaded, `lib2.lib` is appended to `lib1.lib`. This appended library retains the `lib1.lib` name. Finally, `lib3.lib` is loaded.

Preventing the Use of Specific Library Cells

You can specify individual library cells that you want to be excluded during synthesis with the `avoid` attribute:

```
set_attribute avoid true | false cell_name(s)
```

- The following example prevents the use of cells whose names begin with `snl_mux21_prx` and all cells whose names end with `nsdel`:

  ```
  rc:/> set_attribute avoid true { nlc18_custom/snl_mux21_prx* }
  rc:/> set_attribute avoid true { nlc18/*nsdel }
  ```

- The following example prevents the use of the arithmetic shift right ChipWare component (`CW_ashiftr`):

  ```
  rc:/> set_attribute avoid true /hdl_libraries/CW/cw_ashiftr
  ```

Forcing the Use of Specific Library Cells

You can instruct RTL Compiler to use a specific library cell even if the library's vendor has explicitly marked the cell as "don't use" or "don't touch". The following sequential steps illustrate how to force this behavior:

1. Set the `preserve` attribute to `false` on the particular library cell:

   ```
   rc:/> set_attribute preserve false libcell_name
   ```

2. Next, set the `avoid` attribute to `false` on the same cell:

   ```
   rc:/> set_attribute avoid false libcell_name
   ```

Working with Liberty Format Technology Libraries

Source code for technology libraries is written in the `.lib` (Liberty) format. RTL Compiler has a proprietary binary format, `.lbr`, for representing libraries. However, RTL Compiler can directly accommodate the `.lib` format or convert a library from the `.lib` format to the `.lbr` format.

Querying Liberty Attributes

The `liberty_attributes` string is a concatenation of all attribute names and values that were specified in the `.lib` file for a particular object. Use the Tcl utility, `get_liberty_attribute`, to query liberty attributes. The `liberty_attributes` string is read-only, and it appears on the following object types:

- `library`

- libcell
- libpin
- libarc
- wireload
- operating_condition

The following examples demonstrate the uses of the liberty_attributes string:

```
rc:/> get_liberty_attribute “current_unit” [find / -library *]
1ma
rc:/> get_liberty_attribute “area” [find / -libcell nr23d4]
4
rc:/> get_liberty_attribute “cell_footprint” [find / -libcell nr23d4]
aoi_3_5
rc:/> get_liberty_attribute “function” [find / -libpin nr23d4/zn]
(a1'+a2'+a3')
rc:/> get_liberty_attribute “timing_type” [find / -libarc invtd1/zn/en_d50]
three_state_disable
```

Using Custom Pad Cells

RTL Compiler does not insert buffers between pad pins and top level ports, even if design rule violations or setup violations exist. That is, by default, the nets connecting such objects are treated implicitly as dont_touch nets (*not* as ideal nets).

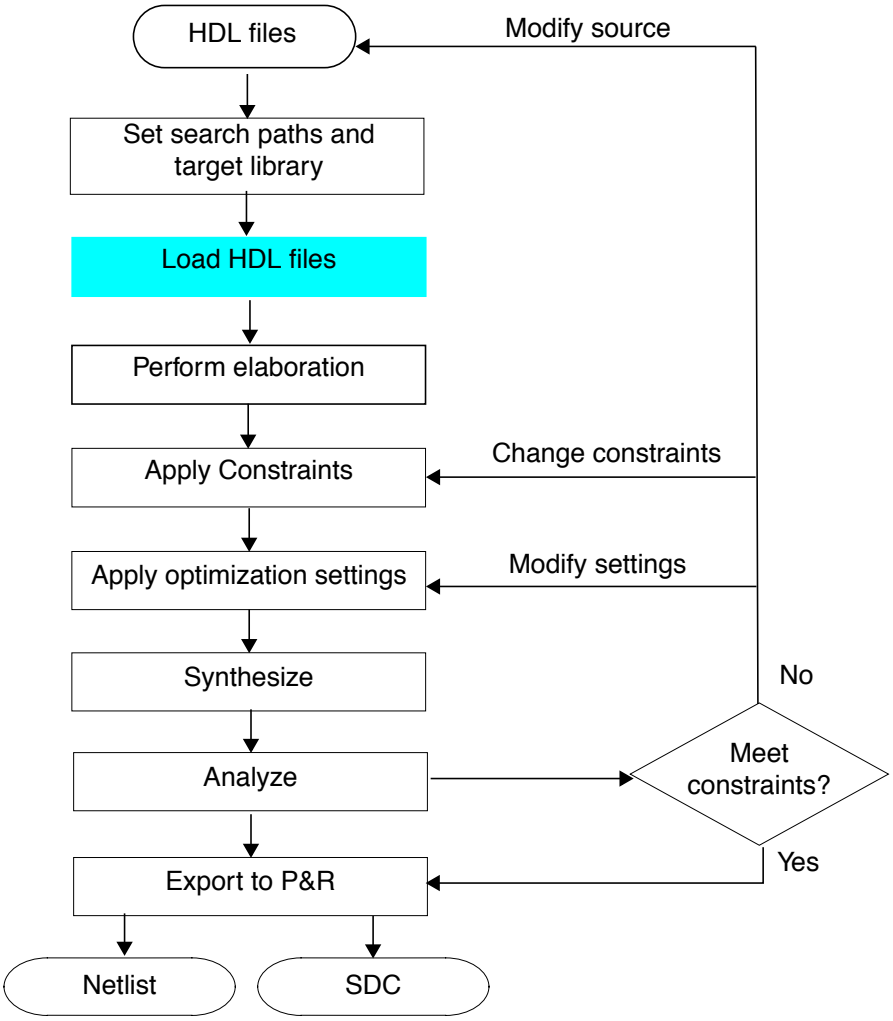
RTL Compiler identifies pad cells through the Liberty attributes is_pad (for libpins) and pad_cell (for libcells). Therefore, if custom pad cells are created and instantiated in the design prior to synthesis, be sure to include the is_pad construct in the libpin description and the pad_cell construct on the libcell description.

5

Loading Files

Overview

This chapter describes how to load HDL files into RTL Compiler.



Tasks

Updating Scripts Through Patching

There is a patch mechanism in RTL Compiler. This mechanism allows “patching” a release by providing a Tcl fix that is automatically sourced during initialization, thus saving you the trouble of having to modify your scripts.

Patches are tied to a version or version range, and they are only applied to the versions they were meant to be used on.

There are two ways to activate a Tcl patch:

1. Copy the patch to the following directory:

   `$CDN_SYNTH_ROOT/lib/cdn/patches`

   You may have to create the directory.

2. .Copy the patch to any directory and point the environment `CDN_SYNTH_PATCH_DIR` variable to that directory.

When a patch is successfully loaded, the RTL Compiler banner will show the patch ID as part of the version. For example, if patches 1 and 3 are applied to version `4.20`, the banner would show the version as being `4.20.p.1.3`.

The `program_version` attribute does not change. The order of the patch ids are in the order in which they are loaded.

Running Scripts

RTL Compiler is a Tcl-based tool and therefore you can create scripts to execute a series of commands instead of typing each command individually. The entire interface is accessible through Tcl and true Tcl syntax and semantics are supported. You can create the script(s) in a text editor and then run them in one of two ways:

- From the UNIX command line, use the `-f` option with the `rc` command to start RTL Compiler and run your scripts immediately:

  ```
  unix> rc -f script_file1 -f script_file2 ...
  ```

  Note: If you have multiple scripts, use the `-f` switch as many times as needed. The scripts are executed in the order they are entered.

- You can simultaneously invoke RTL Compiler as a background process and execute a script by typing the following command from the UNIX command line:

  ```
  unix> rc < script_file_name &
  ```

- If RTL Compiler is already running, use the `include` or `source` command followed by the names of the scripts:

  ```
  rc:/> include script_file1 script_file2 ...
  ```

  or:

  ```
  rc:/> source script_file1 script_file2 ...
  ```

For a sample script file, see "Simple Synthesis Template" on page 271.

Reading HDL Files

Loading HDL Files

HDL files contain design information, such as structural code or RTL implementations. Use the `read_hdl` command to read HDL files into RTL Compiler. When you issue a `read_hdl` command, RTL Compiler reads the files and performs syntax checks.

➤ Read one or more Verilog files in the order given into memory using the following command:

```
rc:/> read_hdl [-v1995 | -v2001 | -sv | -vhdl]
[-library library_name][-structural] [-top top_module_name]
[-define macro=value] file_list
```

Default: `-v1995`

If the design is described by multiple HDL files, you can read them in using the following methods:

- Create a Tcl list that carries the filenames of all the HDL files and use the `read_hdl` command once to read the files simultaneously. For example:

  ```
  read_hdl {top.v block1.v block2.v}
  ```

  or

  ```
  set file_list {top.v block1.v block2.v}
  read_hdl $file_list
  ```

  Note: Elements in a Tcl list are separated by blank spaces, not commas.

  The *`file_list`* is a Tcl list of file names. Use the *`file_list`* argument to specify the name of the HDL file to read. If you are reading several files, then specify them in a string. The host directory where the HDL files are searched for is specified using the `hdl_search_path` root attribute.

See Specifying HDL Search Paths on page 107 for more information.

  The following command reads two VHDL files into a library you defined:

  ```
  read_hdl -vhdl -library my_lib {example1.vhd example2.vhd}
  ```

- Use the `read_hdl` command multiple times to read the files sequentially. For example:

  ```
  read_hdl top.v
  read_hdl {block1.v block2.v}
  ```

  or

  ```
  read_hdl top.v
  read_hdl block1.v
  read_hdl block2.v
  ```

  If multiple files of a design are located at different locations in the UNIX file system, use the `hdl_search_path` attribute to make the TCL scripting more concise. See Specifying HDL Search Paths on page 107 for an example.

- Use the `-v1995` option to specify the Verilog IEEE Std 1364-1995 compliance (default). However, when specifying the -v1995 option, the `read_hdl` command honors the `signed` keyword that was added to the Verilog syntax by IEEE Std 1364-2001. This lets you declare a signal as `signed` to infer signed operators.

- Use the `-v2001` option to specify Verilog IEEE Std1364-2001 compliance. However, if the only v2001 construct you have in the RTL code is the `signed` keyword, you can use the -v1995 option, which supports this keyword.

- Use the `-vhdl` option to specify the VHDL mode, and to read VHDL files where the format is specified by the `hdl_vhdl_read_version` attribute, whose default value is

1993. Read in VHDL designs that are modeled using either the 1987 or the `1993` version, but do not read in a design that has a mixture of these two versions. In other words, use the same version of VHDL when reading in VHDL files.

- Use the `-sv` option to specify the SystemVerilog 3.1 mode.
- Use the `-library` option only with the `-vhdl` option to specify the VHDL library.
- Use the `-netlist` option to read and elaborate a Verilog 1995 structural netlist. Use this option with the `-top` option, which reads and elaborates a specified top level structural Verilog module.Separate multiple file names by blank spaces.

Follow these guidelines when reading HDL files:

- Read files containing macro definitions before the macros are used.
- Using the `-v1995`, `-v2001`, -sv, and `-vhdl` options with the `read_hdl` command will override the setting of the `hdl_language` attribute.
- Follow the `read_hdl` command with the `elaborate` command before using constraint or optimization commands.
- Read in a single library compressed with gzip. For example:

  ```
  read_hdl sample.vhdl.gz.
  ```

  RTL Compiler detects the `.gz` file suffix and automatically unzips the input file.

Your files may have extra, hidden characters (for example, line terminators) if they are transferred from Windows/Dos to UNIX. Be sure to eliminate them because RTL Compiler will issue an error when it encounters these characters.

Specifying the HDL Language Mode

➤ Specify the default language version to read HDL designs using the following attribute:

```
set_attribute hdl_language {v1995 | v2001 | sv | vhdl}
```

Default: `v1995`

This attribute ensures that only HDL files that conform to the appropriate version are parsed successfully.

Note: Using the `-v1995`, `-v2001`, and `-vhdl` options with the `read_hdl` command will override the setting of the `hdl_language` attribute.

By default, RTL Compiler reads Verilog, not VHDL. When reading in Verilog, by default RTL Compiler reads Verilog-1995, not Verilog-2001. When reading VHDL, by default RTL Compiler reads VHDL-1993, not VHDL-1987.

Table 5-1 lists the language modes and the various ways you can use the commands and attributes to set these modes.

Table 5-1 Specifying the Language Mode

Language Mode	Command

Verilog-1995	`read_hdl -v1995 design.v`
	or
	`set_attr hdl_language v1995`
	`read_hdl -v1995 design.v`

Verilog-2001	`read_hdl -v2001 design.v`
	or
	`set_attr hdl_language v2001`
	`read_hdl design.v`

SystemVerilog	`read_hdl -sv design.v`
	or
	`set_attr hdl_language sv`
	`read_hdl design.v`

VHDL-1987	`set_attr hdl_vhdl_read_version 1987`
	`read_hdl -vhdl design.vhd`
	or
	`set_attr hdl_vhdl_read_version 1987`
	`set_attr hdl_language vhdl`
	`read_hdl design.vhd`

Language Mode	Command

VHDL-1993	`set_attr hdl_vhdl_read_version 1993`
	`read_hdl -vhdl design.vhd`
	or
	`set_attr hdl_vhdl_read_version 1993`
	`set_attr hdl_language vhdl`
	`read_hdl design.vhd`

Specifying HDL Search Paths

The HDL files may not be located at the current working directory. Use the `hdl_search_path` attribute to tell RTL Compiler where to look for HDL files. This attribute carries a list of UNIX directories. Whenever a file specified with the `read_hdl` command or an `` `include `` file specified in the Verilog code is needed, RTL Compiler goes to these directories to look for it.

- Specify a list of UNIX directories where RTL Compiler should search for files specified with the `read_hdl` command. For example, the following commands specifies the search path and reads in the `top.v` and `sub.v` files from the appropriate location:

  ```
  set_attr hdl_search_path {../location_of_top ../location_of_sub}
  read_hdl top.v sub.v
  ```

  Default: `set_attr hdl_search_path . /`

  If this attribute carries multiple UNIX directories, the way RTL Compiler searches for HDL files is similar to the search path mechanism in UNIX. Searching for a file follows the order of the directories located in the `hdl_search_path` attribute. The search stops as soon a file is found without trying to explore whether there is another file of the same name located at some other directory specified by the `hdl_search_path` attribute. In other words, if multiple candidates exist, the one found first is chosen.

  For example, assume the design consists of the following three files:

  ```
  ./top.v
  /home/export/my_username/my_project/latest_ver/block1/block1.v
  /home/export/my_username/my_project/latest_ver/block2/block2.v
  ```

  and `top.v` needs the following `` `include `` file:

  ```
  `include "def.h"
  ```

  that is located at the following location:

  ```
  /home/export/my_username/my_project/latest_ver/header/def.h
  ```

  Use the following commands to manage the TCL scripting:

  ```
  set rtl_dir /home/export/my_username/my_project/latest_ver
  set_attr hdl_search_path {. $rtl_dir/header $rtl_dir/block1 $rtl_dir/block2} /
  set file_list {top.v block1.v block2.v}
  read_hdl $file_list
  ```

- If a Verilog subprogram is annotated by a `map_to_module` pragma, which maps it to a module defined in VHDL or a cell defined in a library, the name-based mapping is case-sensitive, and can be affected by the value of the `hdl_vhdl_case` attribute setting.

- If a VHDL subprogram is annotated by a `map_to_module` pragma, which maps it to a module defined in Verilog or a cell that is defined in a library, the name-based mapping is case-insensitive.

Reading a Partially Structural Design

A partially structural design is a mixed RTL design that includes modules that are described by a structural netlist along with modules that are not described by a structural netlist.

➤ Use the `read_hdl - netlist` command to read structural Verilog files when the file does not include behavioral (VHDL or Verilog) modules.

A structural Verilog file contains only structural Verilog-1995 constructs, such as module and gate instances, concurrent assignment statements, references to nets, bit-selects, part-selects, concatenations, and the unary ~ operator. Using the `read_hdl -netlist` command uses less memory and runtime to load a structural file than the `read_hdl` command.

Using the `read_hdl -netlist` command does not automatically elaborate the design. You must use the `elaborate` command.

Note: If the RTL design does not include any modules described by a structural netlist, then use the `read_hdl` command without the `-netlist` option.

Reading and Elaborating a Verilog 1995 Structural Design

In a Verilog 1995 structural netlist, *all* of the modules are represented by a structural netlist.

➤ Use the `read_netlist` command to read and elaborate a Verilog 1995 structural netlist.

A structural Verilog netlist consists of:

- Instantiations of technology elements, Verilog built-in primitives, or user defined modules
- Concurrent assignment statements
- Simple expressions, such as references to nets, bit selects, part selects of nets, concatenations of nets, and the ~ (unary) operator

The `read_netlist` command reads and elaborates the design and creates a generic netlist that is ready to be synthesized. You do *not* need to use the `elaborate` command.

If multiple top-level modules are found in the loaded netlist, RTL Compiler randomly select one of them and deletes the remaining top-level modules.

➤ To specify a top-level module, use the `read_hdl -top` *`module_name`* command.

Reading Verilog Files

Defining Verilog Macros

There are two ways to define a Verilog macro:

- Define it using the `read_hdl` command
- Define it in the Verilog code

Defining a Verilog Macro Using the read_hdl -define Command

➤ Define a Verilog macro using the `-define` option with the `read_hdl` command as follows:

```
read_hdl -define macro verilog_filenames
```

This is equivalent to having a `'define` macro in the Verilog file.

➤ Define the value of a Verilog macro using the `-define "macro = value"` with the `read_hdl` command as follows:

```
read_hdl -define "macro = value" verilog_filenames
```

This is equivalent to having a `'define` macro in the Verilog file.

When the `read_hdl` command uses the `-define` option, it prepends the equivalent `'define` statement to the Verilog file it is loading. For example, you can use one of the following commands:

```
read_hdl -define WA=4 -define WB=6 test.v
read_hdl -define "WA = 4" -define "WB = 6" test.v
```

to read the Verilog file shown in Example 5-1:

Example 5-1 Defining a Verilog Macro Using the read_hdl -define Command

```
‘define MAX(a, b)  ((a) > (b) ? (a) : (b))
module test (y, a, b);
    input [‘WA-1:0] a;
    input [‘WB-1:0] b;
    output [‘MAX(‘WA,‘WB)-1:0] y;
    assign y = a + b;
endmodule
```

This is equivalent to using the `read_hdl test.v` command to read the Verilog file shown in Example 5-2.

Example 5-2 Verilog File with a `define Macro

```
‘define WA 4
‘define WB 6
‘define MAX(a, b)  ((a) > (b) ? (a) : (b))
module test (y, a, b);
    input [‘WA-1:0] a;
    input [‘WB-1:0] b;
    output [‘MAX(‘WA,‘WB)-1:0] y;
    assign y = a + b;
endmodule
```

Important

The order in which you define a Verilog macro is important. Using the `-define` option cannot change a Verilog macro that is defined in the Verilog file. The definition in the HDL code will override the definition using the `read_hdl` command at the command line. For example, using the following command:

```
read_hdl -define WIDTH=6 -define WIDTH=8 test.v
```

to read the Verilog file shown in Example 5-3:

Example 5-3 Using the -define Option Cannot Change a Macro Defined in Verilog Code

```
‘define WIDTH 4
module test (y, a, b);
    input [‘WIDTH-1:0] a, b;
    output [‘WIDTH-1:0] y;
    assign y = a + b;
endmodule
```

This is equivalent to using the `read_hdl test.v` command to read the Verilog file shown in Example 5-4:

Example 5-4 Macro Definition in Verilog Code Overrides read_hdl -define Command

```
‘define WIDTH 6
‘define WIDTH 8
‘define WIDTH 4
module test (y, a, b);
    input [‘WIDTH-1:0] a, b;
    output [‘WIDTH-1:0] y;
    assign y = a + b;
endmodule
```

In this case, the `-define` option is overridden and therefore, ineffective. If a macro is intended to be optionally overridden by the `-define` option using the `read_hdl` command, the Verilog code needs to check the macro’s existence before defining it. For example, you can recode the above example using the following modeling style:

Example 5-5 Overriding a Macro Definition in the Verilog Code

```
‘ifdef WIDTH // do nothing
‘else
‘define WIDTH 4
‘endif
module test (y, a, b);
    input [‘WIDTH-1:0] a, b;
    output [‘WIDTH-1:0] y;
    assign y = a + b;
endmodule
```

Modeling a Macro Using Verilog-2001

Alternatively, using Verilog-2001 you can use the Verilog modeling style shown in Example 5-6:

Example 5-6 Modeling a Macro Definition Using Verilog-2001

```
‘ifndef WIDTH
‘define WIDTH 4
‘endif
module test (y, a, b);
    input [‘WIDTH-1:0] a, b;
    output [‘WIDTH-1:0] y;
    assign y = a + b;
endmodule
```

Reading a Design with Verilog Macros for Multiple HDL Files

If a design is described by multiple HDL files and Verilog macros are used in the design description, the order of reading these HDL files is important.

When the `read_hdl` command is given more than one filename, specify the filenames in a TCL list. The `read_hdl` command loads the files in the specified order in the TCL list.

If multiple `read_hdl` commands are used to load the HDL files, a `‘define` statement is effective until the last file is read, regardless of whether a Verilog macro is defined in an included header file or in the Verilog file itself. The `‘define` statement does not cross over to the next `read_hdl` command.

Therefore, the rules are as follows:

- Read files containing macro definitions before the macros are used.
- Read files containing a macro definition and files using the macro definition in the same `read_hdl` command.

For example, the following files are used to show how ordering affects the functionality of a synthesized netlist:

- A one-line `test.h` file has the `‘define FUNC 2` statement.

- The following `test0.v` file:

```
‘include "test.h"
module tst (y, a, b, c);
    input [3:0] a, b, c;
    output [3:0] y;
    wire [3:0] p;
    blk1 u1 (p, a, b);
    blk2 u2 (y, p, c);
endmodule
```

- The following `test1.v` file:

```
‘ifndef FUNC
    ‘define FUNC 1
‘endif
module blk1 (y, a, b);
    input [3:0] a, b;
    output [3:0] y;
    reg [3:0] y;
    always @ (a or b)
        case (‘FUNC)
           1:  y <= a & b;
           2:  y <= a | b;
           3:  y <= a ^ b;
        endcase
endmodule
```

- The following `test2.v` file:

```
‘ifndef FUNC
    ‘define FUNC 1
‘endif
module blk2 (y, a, b);
    input [3:0] a, b;
    output [3:0] y;
    reg [3:0] y;
    always @ (a or b)
        case (‘FUNC)
           1:  y <= a & b;
           2:  y <= a | b;
           3:  y <= a ^ b;
        endcase
endmodule
```

If going through the following sequence of TCL commands:

```
set_attr library tutorial.lib
set_attr hdl_search_path . /
read_hdl test0.v test1.v
read_hdl test2.v
elaborate
write_hdl -g
```

The `test1.v` file is affected by the macro definition in the `test.h` file, but the `test2.v` file is not, then the following is the generated netlist:

```
module blk1_w_4 (y, a, b); // FUNC defined in test.h
    input [3:0] a, b;
    output [3:0] y;
    wire [3:0] a, b;
    3:0 [3:0] y;
    or g1 (y[0], a[0], b[0]);
    or g2 (y[1], a[1], b[1]);
    or g3 (y[2], a[2], b[2]);
    or g4 (y[3], a[3], b[3]);
endmodule

module blk2_w_4 (y, a, b); // FUNC defined by itself
    input [3:0] a, b;
    output [3:0] y;
    wire [3:0] a, b;
    wire [3:0] y;
    and g1 (y[0], a[0], b[0]);
    and g2 (y[1], a[1], b[1]);
    and g3 (y[2], a[2], b[2]);
    and g4 (y[3], a[3], b[3]);
endmodule

module tst (y, a, b, c);
    input [3:0] a, b, c;
    output [3:00] y;
    wire [3:0] p;
    blk1_w_4 u1(.y (p), .a (a), .b (b));
    blk2_w_4 u2(.y (y), .a (p), .b (c));
endmodule
```

If going through the following sequence of TCL commands:

```
set_attr library tutorial.lib
set_attr hdl_search_path . /
read_hdl test1.v test0.v test2.v
elaborate
write_hdl -g
```

The `test1.v` file is not affected by the macro definition in `test.h`, but the `test2.v` file is. The following is the generated netlist:

```
module blk1_w_4(y, a, b); // FUNC defined by itself
    input [3:0] a, b;
    output [3:0] y;
    wire [3:0] a, b;
    wire [3:0] y;
    and g1 (y[0], a[0], b[0]);
    and g2 (y[1], a[1], b[1]);
    and g3 (y[2], a[2], b[2]);
    and g4 (y[3], a[3], b[3]);
endmodule

module blk2_w_4(y, a, b); // FUNC defined in test.h
    input [3:0] a, b;
    output [3:0] y;
    wire [3:0] a, b;
    wire [3:0] y;
    or g1 (y[0], a[0], b[0]);
    or g2 (y[1], a[1], b[1]);
    or g3 (y[2], a[2], b[2]);
    or g4 (y[3], a[3], b[3]);
endmodule

module tst(y, a, b, c);
    input [3:0] a, b, c;
    output [3:0] y;
    wire [3:0] p;
    blk1_w_4 u1(.y (p), .a (a), .b (b));
    blk2_w_4 u2(.y (y), .a (p), .b (c));
endmodule
```

Reading VHDL Files

Specifying the VHDL Environment

➤ Change the environment setting using the `hdl_vhdl_environment` attribute:

```
set_attribute hdl_vhdl_environment {common | synergy}
```

Default: `common`.

Important

Do not change the `hdl_vhdl_environment` attribute after using the `read_hdl` command or previously analyzed units will be invalidated.

Follow these guidelines when using a predefined VHDL environment:

- Packages and entities in VHDL are stored in libraries. A package contains a collection of commonly used declarations and subprograms. A package can be compiled and used by more than one design or entity.
- RTL Compiler provides a set of pre-defined packages for VHDL designs that use standard arithmetic packages defined by IEEE, Cadence, or Synopsys. The RTL Compiler-provided version of these pre-defined packages are tagged with special directives that let RTL Compiler implement the arithmetic operators efficiently. Each VHDL environment is associated with a unique set of pre-defined packages.
- In each RTL Compiler session, based on the setting of the VHDL environment (`common` or `synergy`) and the VHDL version (`1987` or `1993`), RTL Compiler pre-loads a set of pre-defined packages from the following directory:

  ```
  $CDN_SYNTH_ROOT/lib/vhdl/
  ```

- Refer to Table 5-2 for a description of the predefined VHDL environments and to Table 5-3 and Table 5-4 for descriptions of all the predefined libraries for each of the VHDL environments.

See Using Arithmetic Packages From Other Vendors on page 120 for more information.

Table 5-2 Predefined VHDL Environments

`synergy`	Uses the arithmetic packages supported by the CADENCE Synergy synthesis tool.
`common`	Uses the arithmetic packages supported by the IEEE standards and the arithmetic packages supported by Synopsys' VHDL Compiler. (Default)

Table 5-3 Predefined VHDL Libraries Synergy Environment

Library	Packages
CADENCE	`attributes`
STD	`standard`
	`textio`
SYNERGY	`constraints`
	`signed_arith`
	`std_logic_misc`
IEEE	`std_logic_1164`
	`std_logic_arith`
	`std_logic_textio`

Table 5-4 Predefined VHDL Libraries Common Environment

Library	Packages
CADENCE	`attributes`
STD	`standard`
	`textio`
SYNOPSYS	`attributes`
	`bv_arithmetic`

Table 5-4 Predefined VHDL Libraries Common Environment, *continued*

IEEE	numeric_bit
	numeric_std
	std_logic_1164
	std_logic_arith
	std_logic_misc
	std_logic_signed
	std_logic_textio
	std_logic_unsigned
	vital_primitives
	vital_timing

Verifying VHDL Code Compliance with the LRM

➤ Enforce a strict interpretation of the *VHDL Language Reference Manual* (LRM) to guarantee portability to other VHDL tools using the following attribute:

```
set_attribute hdl_vhdl_lrm_compliance true
```

Default: `false`

Specifying Illegal Characters in VHDL

If you want to include characters in a name that are illegal in VHDL, add a \ character before and after the name, and add space after the name.

Showing the VHDL Logical Libraries

➤ Show the VHDL logical libraries using the `ls /hdl_libraries/*` command. For example:

```
rc:/> ls /hdl_libraries
```

For detailed information, see Chapter 4, “The RTL Compiler Design Information Hierarchy”

Using Arithmetic Packages From Other Vendors

See Specifying the VHDL Environment on page 117 for a description of the pre-defined packages for VHDL designs that use standard arithmetic packages defined by IEEE, Cadence, or Synopsys.

You can override any pre-loaded package or add you own package to a pre-defined library if your design must use arithmetic packages from a third-party tool-vendor or IP provider.

To use arithmetic packages from other vendors, follow these steps:

1. Set up your VHDL environment and VHDL version using the following attributes:

   ```
   set_attribute hdl_vhdl_environment {common | synergy}
   set_attribute hdl_vhdl_read_version { 1993 | 1987 }
   ```

   RTL Compiler automatically loads the pre-defined packages in pre-defined libraries.

2. Analyze third-party packages to override pre-defined packages, if necessary. For example, suppose you have your own package whose name matches one of the `IEEE` packages, and the package name is `std_logic_arith`. Suppose the VHDL source code of your own package is in a file named `my_std_logic_arith.vhdl`. You can override this package in the IEEE library using the following command:

   ```
   read_hdl -vhdl -lib ieee my_std_logic_arith.vhdl
   ```

   Later, if a VHDL design file contains a reference to this package as follows:

   ```
   library ieee;
   use ieee.std_logic_arith.all;
   ```

   RTL Compiler uses the user-defined `ieee.std_logic_arith` package, and never sees the pre-defined `ieee.std_logic_arith` package any more.

3. You can analyze additional third-party packages into a pre-defined library. For example, you have a package whose name does not match one of the pre-defined packages, but you want to add it to the pre-defined `ieee` library. Suppose the package name is `my_extra_pkg` and the VHDL source code of this additional package is in a file named `my_extra_pkg.vhdl`. Add the package into the pre-defined `ieee` library using he following command:

   ```
   read_hdl -vhdl -lib ieee my_extra_pkg.vhdl
   ```

   Later, your VHDL design file can use this package by:

   ```
   library ieee;
   use ieee.my_extra_pkg.all;
   ```

4. Read the VHDL files of your design.

Note: If an entity refers to a package, read in the package before reading in the entity.

Modifying the Case of VHDL Names

➤ Specify the case of VHDL names stored in the tool using the following attribute:

```
set_attribute hdl_vhdl_case { lower | upper | original }
```

For example:

```
set_attribute hdl_vhdl_case lower
```

The case of VHDL names is only relevant for references by foreign modules. Examples of foreign references are Verilog modules and library cells.

Follow these guidelines when modifying the case of VHDL names:

- `lower`–Converts all names to lower-case (`Xpg` is stored as `xpg`).
- `upper`–Converts all names to upper-case (`Xpg` is stored as `XPG`).
- `original`–Preserves the case used in the declaration of the object (`Xpg` is stored as `Xpg`).

Reading Designs with Mixed Verilog and VHDL Files

Reading in Verilog Modules and VHDL Entities With Same Names

RTL Compiler only supports one module or entity with a given name and any definition, either a module or entity, overwrites any previous definition. RTL Compiler generates the following Information message whenever the definition of a module or entity is overwritten by a new module or entity with the same name:

```
Info    :Replacing previously read module [HPT-76]
        :Replacing VHDL module 'test_sub' with Verilog module in file test_sub.v
at line 1
        :A module is replaced when a module of the same name and same library is
read again. The HDL module pool cannot have two modules with the same name in the
same library. Since VHDL is case-insensitive, if either of the two modules with
the same names (but in different case) is a VHDL entity, the more recently read
of the modules prevails, for instance:
        VHDL 'foo' replaces VHDL 'FOO'
        VHDL 'foo' replaces Verilog 'FOO'
        Verilog 'foo' replaces VHDL 'FOO'
```

Using Case Sensitivity in Verilog/VHDL Mixed-Language Designs

RTL Compiler supports a mixed-language design description, which means part of the design is written in VHDL and the rest of the design is written in Verilog. VHDL is case-insensitive and Verilog is case-sensitive.

Care must be taken when the HDL code refers to an object defined in another language.

Follow these guidelines when reading mixed-language designs:

- If the entire design is described in Verilog, the mapping of module names, pin names, and parameter names are all case-sensitive.
- If the entire design is described in VHDL, the mapping of module names, pin names, and parameter names are all case-insensitive.
- If the names of all the objects defined in the Verilog code or in the technology libraries only use lower-case letters, set the `hdl_vhdl_case` attribute to `lower` to prevent any case sensitivity issues.
- If the names of all the objects defined in Verilog code or in the technology libraries only use upper-case letters, set the `hdl_vhdl_case` attribute to `upper` to prevent any case sensitivity issues.
- If the Verilog code instantiates a VHDL entity, the mapping of entity and port names is case-sensitive and can be affected by the value of the `hdl_vhdl_case` attribute setting.
- If the VHDL code instantiates a Verilog module, the mapping of module and port names is case-insensitive.

If the `hdl_vhdl_case` attribute is set to `original`, make sure the Verilog code refers to foreign objects using exactly the same case they are defined in the VHDL code or in the technology library.

Keeping Track of Loaded Files

- Use the hdl_filelist attribute to keep track of the HDL files that have been read into RTL Compiler. Each time you use the `read_hdl` command to read in an HDL file, the library, filename, and language format are appended to this attribute in a TCL list.

  If you use the `hdl_filelist` attribute is a root attribute if you use it before elaboration. After elaboration this attribute is attached to the design. For example:

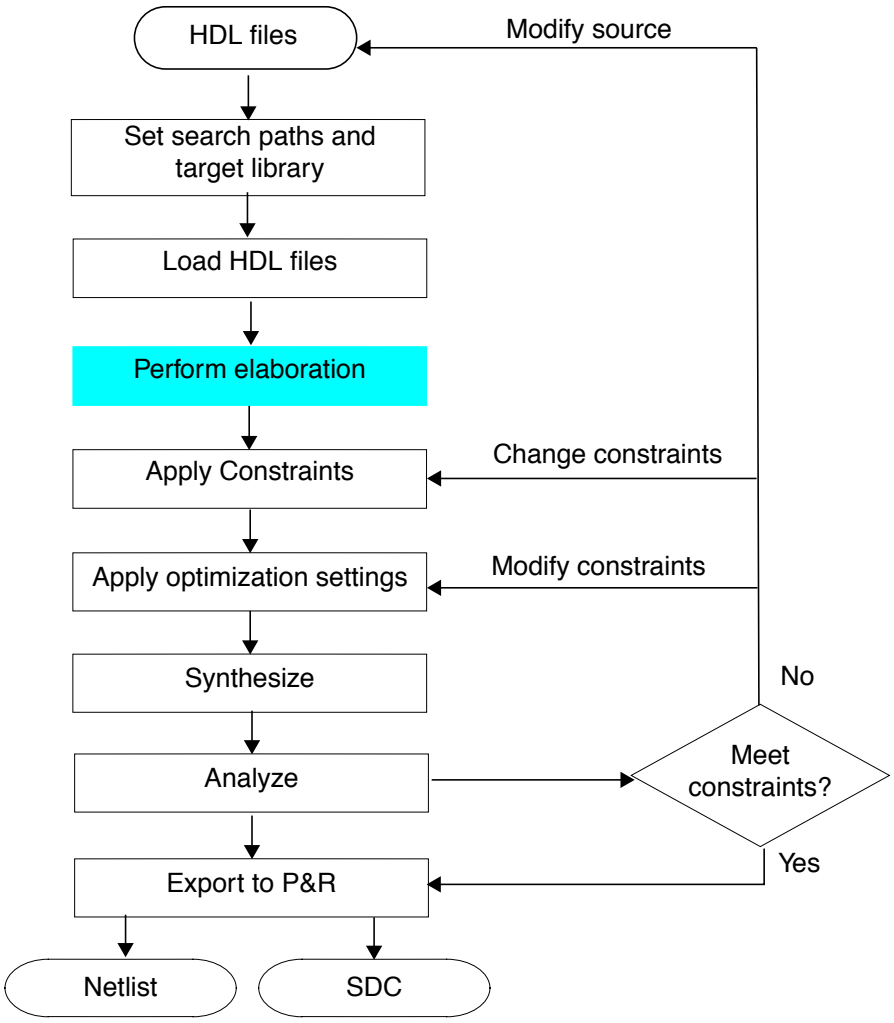
```
rc> read_hdl -v2001 top.v
rc> get_attr hdl_filelist
{default -v2001 {top.v}} {mylib -vhdl {sub.vhdl}}
rc> read_hdl -vhdl -lib mylib sub.vhdl
rc> get_attr hdl_filelist
{default -v2001 {top.v}} {mylib -vhdl {sub.vhdl}}
rc> elaborate
rc> get_attr hdl_filelist /designs/top
{default -v2001 {top.v}} {mylib -vhdl {sub.vhdl}}
```

6

Elaborating the Design

Overview

Elaboration involves various design checking and optimizations and is a necessary step to proceed with synthesis. This chapter describes elaboration in detail.



Tasks

Performing Elaboration

The `elaborate` command automatically elaborates the top-level design and all of its references. During elaboration, RTL Compiler performs the following tasks:

- Builds data structures
- Infers registers in the design
- Performs higher-level HDL optimization, such as dead code removal
- Checks semantics

Note: If there are any gate-level netlists read in with the RTL files, RTL Compiler automatically links the cells to their references in the technology library during elaboration. You do not have to issue an additional command for linking.

At the end of elaboration, RTL Compiler displays any unresolved references (immediately after the key words `Done elaborating`):

```
Done elaborating '<top_level_module_name>'.
Cannot resolve reference to <ref01>
Cannot resolve reference to <ref02>
Cannot resolve reference to <ref03>
...
```

After elaboration, RTL Compiler has an internally created data structure for the whole design so you can apply constraints and perform other operations.

Performing Elaboration with no Parameters

1. Load all Verilog files with the `read_hdl` command.

   For information on the `read_hdl` command, see Loading HDL Files on page 102.

2. Type the following command to start elaboration with no parameters:

   ```
   rc:/> elaborate toplevel_module
   ```

Performing Elaboration with Parameters

You can overwrite existing design parameters during elaboration. For example, the following module has the `width` parameter set to `8`:

```
module alu(aluout, zero, opcode, data, accum, clock, ena, reset);
  parameter width=8;
  input clock, ina, reset;
  input [width-1.0] data, accum;
  input [2:0] opcode;
  output [width-1.0] aluout;
  output zero;
  ...
```

You can change it to `16` by issuing the following command:

```
rc:/> elaborate alu -parameters 16
```

The `alu_out` will be built as a 16-bit port.

Important

If there are multiple parameters in your Verilog code, you must specify the value of each one in the order that they appear in the code. *Do not skip any parameters or you risk setting one to the wrong value.*

The following example sets the value of the first parameter to `16`, the second to `8`, and the third to `32`.

```
rc:/> elaborate design1 -parameters {16 8 32}
```

Specifying HDL Library Search Paths

➤ Specify a list of UNIX directories where RTL Compiler should search for files for unresolved modules or entities when using the elaborate command. For example, the following commands specifies the search path and reads in the top.v file, which has an instance of module sub, but the top.v file does not contain a description of module sub:

```
set_attr hdl_search_path {../location_of_top}
read_hdl top.v
set_attribute library tutorial.lib
elaborate -libpath ../mylibs -libpath /home/verilog/libs -libext ".h"
-libext ".v"
```

The latter command is equivalent to the following:

```
elaborate -libpath { ../mylibs /home/verilog/libs } -libext { ".h" ".v" }
```

The elaborate command looks for the top.v file in the directories specified through the hdl_search_path attribute. After top.v is parsed, the elaborate command looks for undefined modules, such as sub, in the directories specified through the -libpath option. First, the tool looks for a file that corresponds to the name of the module appended by the first specified file extension (sub.h). Next, it looks for a file that corresponds to the name of the module appended by the next specified file extension (sub.v), and so on.

Elaborating a Specified Module or Entity

➤ Generate a generic netlist for a specific Verilog module and all its sub-modules, or a VHDL entity and all its components using the elaborate command as follows:

```
elaborate des_top
```

Naming Individual Bits of Array and Record Ports and Registers

Use the following attributes to control the instance names of sequential elements (flip-flops and latches) that represent individual bits of an array or a VHDL record. They also control bit-blasted port names of an input/output port that is an array or a VHDL record.

- hdl_array_naming_style
- hdl_record_naming_style
- hdl_reg_naming_style

See Examples on page 132 for information on how to use these attributes.

➤ Use the `hdl_array_naming_style` attribute to control how an index of an array is appended to an instance or port name.

Default: `%s\[%d\]`

The `hdl_array_naming_style` attribute expects a string that contains zero or one `%s` followed by one `%d`. For example, this attribute's value can be something such as: `_%d`, `_%d_`, `\[%d\]`, `%s_%d`, `%s_%d_`, `%s\[%d\]`, and so on.

The following rules assume an array of multiple dimension. A one-dimensional array is a degenerated case of a multi-dimension array.

- If there is no `%s` in this string, RTL Compiler follows these rules when using this attribute:
  - A suffix is generated for each dimension, according to the format specified in this string.
  - The `%d` represents an index of a certain dimension.
  - All pieces of the suffix are concatenated, from the highest dimension to the lowest dimension, to form a combined suffix.
- If there is a `%s` in this string, RTL Compiler follows these rules when using this attribute:
  - The %s represents the *growing* combined suffix, formed in the way described below.
  - The `%d` represents an index of a certain dimension.
  - The combined suffix starts as a null string. From the highest dimension to the lowest dimension, each dimension is incorporated into the growing suffix, according to the format specified in this string.

In either case, the combined suffix is later appended to the base name to form the instance name. The base name refers to the signal name seen in the RTL statement that declares this array.

➤ Use the `hdl_record_naming_style` attribute to control how a field of a VHDL record is appended to an instance or port name.

Default: `%s\[%s\]`

The `hdl_record_naming_style` attribute expects a string that contains one or two `%s`. For example, its value can be something such as: `_%s`, `_%s_`, `\[%s\]`, `%s_%s`, `%s_%s_`, `%s\[%s\]` and so on.

For naming purposes, a record is like an array whose indices are textual, not numerical. The following rules assume a multi-layer record, meaning a record of record of record, and so on. The usual record is a degenerated case of a multi-layer record.

■ If there is only one `%s` specified in the attribute string, RTL Compiler follows these rules when using this attribute:

- ❑ A suffix is generated for each layer of record, according to the format specified in this string.
- ❑ The `%s` represents a field of a certain layer of record.
- ❑ All pieces of suffix are concatenated, from the outmost layer to the innermost layer, to form a combined suffix.

■ If there are two `%s` specified in this attribute string, RTL Compiler follows these rules when using this attribute:

- ❑ The first `%s` represents the *growing* combined suffix, formed in the way described below.
- ❑ The second `%s` represents a field of a certain layer of record.
- ❑ The combined suffix starts as a null string. From the outmost layer to the innermost layer, each layer is incorporated into the growing suffix, according to the format specified in this string.

In either case, the combined suffix is later appended to the base name to form the instance name. The base name refers to the signal name seen in the RTL statement that declares this record.

➤ Use the `hdl_reg_naming_style` attribute to control how to compose the instance or port name based on the:

- ❑ Signal name declared in the RTL code

- Combined suffix from its array/record indices/field, if any.

  Default: `%s_reg%s`

  The `hdl_reg_naming_style` attribute expects a string that contains one or two `%s`. The attribute value is usually set to either `_reg%s` or `%s_reg%s`.

  The following rules treat a record as an array with textual indices. A combined suffix is formed based on formats specified by the `hdl_array_naming_style` attribute and the `hdl_record_naming_style` attribute.

- If there is only one `%s` specified in the attribute string, RTL Compiler follows these rules when using this attribute:
  - The only `%s` represents the combined suffix.
  - The combined suffix is appended to the base name to form the instance/port name, according to the format specified in this string.
  - There is no way to add a prefix to the instance or port name. The prefix is always a null string. An instance or port name is always started with the base name.
- If there are two `%s` in this string, RTL Compiler follows these rules when using this attribute:
  - The first `%s` represents the base name.
  - The second `%s` represents the combined suffix.
  - The instance or port name is assembled from the base name plus the combined suffix, according to the format specified in this string.
  - There can be a prefix before the base name, such as before the first `%s`.

Examples

The following shows how to use these attributes, assuming you have the RTL shown in Example 6-1.

Example 6-1 Verilog RTL

```
module tst (clk, in, out);
    input clk, in;
    output out;
    reg a;
    reg b[3:2];
    reg c[5:4][3:2];
    reg [1:0] d;
    reg [1:0] e[3:2];
    reg [1:0] f[5:4][3:2];
    integer i, j, k;
    always @ (posedge clk)
    begin
                                a <= in;
        for (i=2 ; i<=3 ; i=i+1)
                                b[i] <= in;
        for (i=4 ; i<=5 ; i=i+1)
        for (j=2 ; j<=3 ; j=j+1)
                                c[i][j] <= in;
        for (i=0 ; i<=1 ; i=i+1)
                                d[i] <= in;
        for (i=2 ; i<=3 ; i=i+1)
        for (j=0 ; j<=1 ; j=j+1)
                                e[i][j] <= in;
        for (i=4 ; i<=5 ; i=i+1)
        for (j=2 ; j<=3 ; j=j+1)
        for (k=0 ; k<=1 ; k=k+1)
                                f[i][j][k] <= in;
    end
    assign out = a & b[2] & b[3] & d[0] & d[1] &
        c[5][3] & c[5][2] & c[4][3] & c[4][2] &
        e[3][1] & e[3][0] & e[2][1] & e[2][0] &
        f[5][3][1] & f[5][3][0] & f[5][2][1] & f[5][2][0] &
        f[4][3][1] & f[4][3][0] & f[4][2][1] & f[4][2][0];
endmodule
```

If you set the following attributes:

```
set_attr hdl_array_naming_style "_%d_"
set_attr hdl_reg_naming_style   "_reg%s"
```

or

```
set_attr hdl_array_naming_style "%s_%d_"
set_attr hdl_reg_naming_style   "%s_reg%s"
```

The register instance names in the netlist are as follows:

```
a_reg                       f_reg_4__2__0_
b_reg_2_       d_reg_0_     f_reg_4__2__1_
b_reg_3_       d_reg_1_     f_reg_4__3__0_
c_reg_4__2_    e_reg_2__0_  f_reg_4__3__1_
c_reg_4__3_    e_reg_2__1_  f_reg_5__2__0_
c_reg_5__2_    e_reg_3__0_  f_reg_5__2__1_
c_reg_5__3_    e_reg_3__1_  f_reg_5__3__0_
                            f_reg_5__3__1_
```

If you set the following attributes:

```
set_attr hdl_array_naming_style "_%d"
set_attr hdl_reg_naming_style   "_reg%s
```

or

```
set_attr hdl_array_naming_style "%s_%d"
set_attr hdl_reg_naming_style   "%s_reg%s"
```

The register instance names in the netlist are as follows:

```
a_reg                       f_reg_4_2_0_
b_reg_2_       d_reg_0_     f_reg_4_2_1_
b_reg_3_       d_reg_1_     f_reg_4_3_0_
c_reg_4_2_    e_reg_2_0_    f_reg_4_3_1_
c_reg_4_3_    e_reg_2_1_    f_reg_5_2_0_
c_reg_5_2_    e_reg_3_0_    f_reg_5_2_1_
c_reg_5_3_    e_reg_3_1_    f_reg_5_3_0_
                            f_reg_5_3__
```

If you set the following attributes:

```
set_attr hdl_array_naming_style "\[%d\]"
set_attr hdl_reg_naming_style   "_reg%s"
```

or

```
set_attr hdl_array_naming_style "%s\[%d\]"; # default
set_attr hdl_reg_naming_style   "%s_reg%s";# default
```

The register instance names in the netlist are as follows:

```
a_reg                      \f_reg[4][2][0]
_reg[2]       †_reg[0]     \f_reg[4][2][1]
_reg[3]       †_reg[1]     \f_reg[4][3][0]
\c_reg[4][2] _reg[2][0]    \f_reg[4][3][1]
\c_reg[4][3] _reg[2][1]    \f_reg[5][2][0]
\c_reg[5][2] _reg[3][0]    \f_reg[5][2][1]
\c_reg[5][3] _reg[3][1]    \f_reg[5][3][0]
                           \f_reg[5][3][1]
```

If you set the following attributes:

```
set_attr hdl_array_naming_style "A_%s<%d>"
set_attr hdl_reg_naming_style   "L_%s_ReG%s"
```

The register instance names in the netlist are as follows:

```
L_a_ReG                                 \L_f_ReGA_A_A_<4><2><0>
\L_b_ReGA_<2>       \L_d_ReGA_<0>       \L_f_ReGA_A_A_<4><2><1>
\L_b_ReGA_<3>       \L_d_ReGA_<1>       \L_f_ReGA_A_A_<4><3><0>
\L_c_ReGA_A_<4><2>  \L_e_ReGA_A_<2><0>  \L_f_ReGA_A_A_<4><3><1>
\L_c_ReGA_A_<4><3>  \L_e_ReGA_A_<2><1>  \L_f_ReGA_A_A_<5><2><0>
\L_c_ReGA_A_<5><2>  \L_e_ReGA_A_<3><0>  \L_f_ReGA_A_A_<5><2><1>
\L_c_ReGA_A_<5><3>  \L_e_ReGA_A_<3><1>  \L_f_ReGA_A_A_<5><3><0>
                                        \L_f_ReGA_A_A_<5><3><1>
```

VHDL Example

If you have the following RTL as shown in Example 6-2:

Example 6-2 VHDL RTL

```
package p is
                type type_a is array (1 to 2) of bit;
                type type_r is record j1, j2 : bit;      end record;
                type type_aa is array (3 to 4) of type_a;
                type type_rr is record k3, k4 : type_r;   end record;
                type type_ar is array (3 to 4) of type_r;
                type type_ra is record m3, m4 : type_a;   end record;
                type type_ara is array (5 to 6) of type_ra;
                type type_rar is record n5, n6 : type_ar;  end record;
end;

use work.p.all;
entity tst is
    port (  u : out type_a;    a : in type_a;
            v : out type_r;    b : in type_r;
            w : out type_aa;   c : in type_aa;
            x : out type_rr;   d : in type_rr;
            y : out type_ara;  e : in type_ara;
            z : out type_rar;  f : in type_rar;
            clk : in bit                          );
end;

architecture rtl of tst is
begin
    process (clk)
    begin
        if (clk = '1' and clk'event) then
                u <= a;
                v <= b;
                w <= c;
                x <= d;
                y <= e;
                z <= f;

        end if;

    end process;
end rtl;
```

If you set the following attributes:

```
set_attr hdl_array_naming_style  "_%d_"
set_attr hdl_record_naming_style "_%s_"
set_attr hdl_reg_naming_style    "_reg%s"
```

or

```
set_attr hdl_array_naming_style  "%s_%d_"
set_attr hdl_record_naming_style "%s_%s_"
set_attr hdl_reg_naming_style    "%s_reg%s"
```

The register instance names in the netlist are as follows:

```
u_reg_1_     v_reg_j1_
u_reg_2_     v_reg_j2_
w_reg_3__1_       y_reg_5__m3__1_     z_reg_n5__3__j1_
w_reg_3__2_       y_reg_5__m3__2_     z_reg_n5__3__j2_
w_reg_4__1_       y_reg_5__m4__1_     z_reg_n5__4__j1_
w_reg_4__2_       y_reg_5__m4__2_     z_reg_n5__4__j2_
x_reg_k3__j1_     y_reg_6__m3__1_     z_reg_n6__3__j1_
x_reg_k3__j2_     y_reg_6__m3__2_     z_reg_n6__3__j2_
x_reg_k4__j1_     y_reg_6__m4__1_     z_reg_n6__4__j1_
x_reg_k4__j2_     y_reg_6__m4__2_     z_reg_n6__4__j2_
```

The input/output port names in the netlist are as follows:

```
input clk;           input d_k3__j1_;        input f_n5__3__j1_;
input [1:2] a;       input d_k3__j2_;        input f_n5__3__j2_;
input b_j1_;         input d_k4__j1_;        input f_n5__4__j1_;
input b_j2_;         input d_k4__j2_;        input f_n5__4__j2_;
input [1:2] c_3_;    input [1:2] e_5__m3_;   input f_n6__3__j1_;
input [1:2] c_4_;    input [1:2] e_5__m4_;   input f_n6__3__j2_;
                     input [1:2] e_6__m3_;   input f_n6__4__j1_;
                     input [1:2] e_6__m4_;   input f_n6__4__j2_;
output [1:2] u;      output x_k3__j1_;       output z_n5__3__j1_;
output v_j1_;        output x_k3__j2_;       output z_n5__3__j2_;
output v_j2_;        output x_k4__j1_;       output z_n5__4__j1_;
output [1:2] w_3_;   output x_k4__j2_;       output z_n5__4__j2_;
output [1:2] w_4_;   output [1:2] y_5__m3_;  output z_n6__3__j1_;
                     output [1:2] y_5__m4_;  output z_n6__3__j2_;
                     output [1:2] y_6__m3_;  output z_n6__4__j1_;
                     output [1:2] y_6__m4_;  output z_n6__4__j2_;
```

If you set the following attributes:

```
set_attr hdl_array_naming_style  "_%d"
set_attr hdl_record_naming_style "_%s"
set_attr hdl_reg_naming_style    "_reg%s"
```

or

```
set_attr hdl_array_naming_style  "%s_%d"
set_attr hdl_record_naming_style "%s_%s"
set_attr hdl_reg_naming_style    "%s_reg%s"
```

The register instance names in the netlist are as follows:

```
u_reg_1     v_reg_j1_
u_reg_2     v_reg_j2_
w_reg_3_1   y_reg_5_m3_1 z_reg_n5_3_j1
w_reg_3_2   y_reg_5_m3_2 z_reg_n5_3_j2
w_reg_4_1   y_reg_5_m4_1 z_reg_n5_4_j1
w_reg_4_2   y_reg_5_m4_2 z_reg_n5_4_j2
x_reg_k3_j1 y_reg_6_m3_1 z_reg_n6_3_j1
x_reg_k3_j2 y_reg_6_m3_2 z_reg_n6_3_j2
x_reg_k4_j1 y_reg_6_m4_1 z_reg_n6_4_j1
x_reg_k4_j2 y_reg_6_m4_2 z_reg_n6_4_j2
```

The input/output port names in the netlist are as follows:

```
input clk;          input d_k3_j1;        input f_n5_3_j1;
input [1:2] a;      input d_k3_j2;        input f_n5_3_j2;
input b_j1;         input d_k4_j1;        input f_n5_4_j1;
input b_j2;         input d_k4_j2;        input f_n5_4_j2;
input [1:2] c_3;    input [1:2] e_5_m3;   input f_n6_3_j1;
input [1:2] c_4;    input [1:2] e_5_m4;   input f_n6_3_j2;
                    input [1:2] e_6_m3;   input f_n6_4_j1;
                    input [1:2] e_6_m4;   input f_n6_4_j2;
output [1:2] u;     output x_k3_j1;       output z_n5_3_j1;
output v_j1;        output x_k3_j2;       output z_n5_3_j2;
output v_j2;        output x_k4_j1;       output z_n5_4_j1;
output [1:2] w_3;   output x_k4_j2;       output z_n5_4_j2;
output [1:2] w_4;   output [1:2] y_5_m3;  output z_n6_3_j1;
                    output [1:2] y_5_m4;  output z_n6_3_j2;
                    output [1:2] y_6_m3;  output z_n6_4_j1;
                    output [1:2] y_6_m4;  output z_n6_4_j2;
```

If you set the following attributes:

```
set_attr hdl_array_naming_style  "\[%d\]"
set_attr hdl_record_naming_style "\[%s\]"
set_attr hdl_reg_naming_style    "_reg%s"
```

or

```
set_attr hdl_array_naming_style  "%s\[%d\]";#default
set_attr hdl_record_naming_style "%s\[%s\]";#default
set_attr hdl_reg_naming_style    "%s_reg%s";#default
```

The register instance names in the netlist are as follows:

```
u_reg[1]     \v_reg[j1]
u_reg[2]     \v_reg[j2]
\w_reg[3][1]       \y_reg[5][m3][1]     \z_reg[n5][3][j1]
\w_reg[3][2]       \y_reg[5][m3][2]     \z_reg[n5][3][j2]
\w_reg[4][1]       \y_reg[5][m4][1]     \z_reg[n5][4][j1]
\w_reg[4][2]       \y_reg[5][m4][2]     \z_reg[n5][4][j2]
\x_reg[k3][j1]     \y_reg[6][m3][1]     \z_reg[n6][3][j1]
\x_reg[k3][j2]     \y_reg[6][m3][2]     \z_reg[n6][3][j2]
\x_reg[k4][j1]     \y_reg[6][m4][1]     \z_reg[n6][4][j1]
\x_reg[k4][j2]     \y_reg[6][m4][2]     \z_reg[n6][4][j2]
```

The input/output port names in the netlist are as follows:

```
input clk;           input [k3][j1];          input \f[n5][3][j1];
input [1:2] a;       input [k3][j2];          input \f[n5][3][j2];
input [j1];          input [k4][j1];          input \f[n5][3][j1];
input [j2];          input [k4][j2];          input \f[n5][3][j2];
input [1:2] \c[3];   input [1:2] [5][m3];     input \f[n6][3][j1];
input [1:2] \c[4];   input [1:2] [5][m4];     input \f[n6][3][j2];
                     input [1:2] [6][m3];     input \f[n6][3][j1];
                     input [1:2] [6][m4];     input \f[n6][3][j2];
output [1:2] u;      output \x[k3][j1];       output \z[n5][3][j1];
output \v[j1];       output \x[k3][j2];       output \z[n5][3][j2];
output \v[j2];       output \x[k4][j1];       output \z[n5][3][j1];
output [1:2] \w[3]; output \x[k4][j2];        output \z[n5][3][j2];
output [1:2] \w[4]; output [1:2] \y[5][m3]; output \z[n6][3][j1];
                     output [1:2] \y[5][m4]; output \z[n6][3][j2];
                     output [1:2] \y[6][m3]; output \z[n6][3][j1];
                     output [1:2] \y[6][m4]; output \z[n6][3][j2];
```

If you set the following attributes:

```
set_attr hdl_array_naming_style  "A_%s<%d>"
set_attr hdl_record_naming_style "B_%s<%s>"
set_attr hdl_reg_naming_style    "L_%s_ReG%s"
```

The register instance names in the netlist are as follows:

```
\L_u_ReGA_<1>     \L_v_ReGB_<j1>
\L_u_ReGA_<2>     \L_v_ReGB_<j2>
\L_w_ReGA_A_<3><1>     \L_y_ReGA_B_A_<5><m3><1>   \L_z_ReGB_A_B_<n5><3><j1>
\L_w_ReGA_A_<3><2>     \L_y_ReGA_B_A_<5><m3><2>   \L_z_ReGB_A_B_<n5><3><j2>
\L_w_ReGA_A_<4><1>     \L_y_ReGA_B_A_<5><m4><1>   \L_z_ReGB_A_B_<n5><4><j1>
\L_w_ReGA_A_<4><2>     \L_y_ReGA_B_A_<5><m4><2>   \L_z_ReGB_A_B_<n5><4><j2>
\L_x_ReGB_B_<k3><j1>   \L_y_ReGA_B_A_<6><m3><1>   \L_z_ReGB_A_B_<n6><3><j1>
\L_x_ReGB_B_<k3><j2>   \L_y_ReGA_B_A_<6><m3><2>   \L_z_ReGB_A_B_<n6><3><j2>
\L_x_ReGB_B_<k4><j1>   \L_y_ReGA_B_A_<6><m4><1>   \L_z_ReGB_A_B_<n6><4><j1>
\L_x_ReGB_B_<k4><j2>   \L_y_ReGA_B_A_<6><m4><2>   \L_z_ReGB_A_B_<n6><4><j2>
```

The input/output port names in the netlist are as follows:

```
input clk;              input \B_B_d<k3><j1>;        input  \B_A_B_f<n5><3><j1>;
input [1:2] a;          input \B_B_d<k3><j2>;        input  \B_A_B_f<n5><3><j2>;
input \B_b<j1>;         input \B_B_d<k4><j1>;        input  \B_A_B_f<n5><4><j1>;
input \B_b<j2>;         input \B_B_d<k4><j2>;        input  \B_A_B_f<n5><4><j2>;
input [1:2] \A_c<3>;    input [1:2 \B_A_e<5><m3>;  input  \B_A_B_f<n6><3><j1>;
input [1:2] \A_c<4>;    input [1:2] \B_A_e<5><m4>;   input  \B_A_B_f<n6><3><j2>;
                        input [1:2] \B_A_e<6><m3>;   input  \B_A_B_f<n6><4><j1>;
                        input [1:2] \B_A_e<6><m4>;   input  \B_A_B_f<n6><4><j2>;
output [1:2] u;         output \B_B_x<k3><j1>;       output \B_A_B_z<n5><3><j1>;
output \B_v<j1>;        output \B_B_x<k3><j2>;       output \B_A_B_z<n5><3><j2>;
output \B_v<j2>;        output \B_B_x<k4><j1>;       output \B_A_B_z<n5><4><j1>;
output [1:2] \A_w<3>;   output \B_B_x<k4><j2>;       output \B_A_B_z<n5><4><j2>;
output [1:2] \A_w<4>;   output [1:2] \B_A_y<5><m3>;  output \B_A_B_z<n6><3><j1>;
                        output [1:2] \B_A_y<5><m4>;  output \B_A_B_z<n6><3><j2>;
                        output [1:2] \B_A_y<6><m3>;  output \B_A_B_z<n6><4><j1>;
                        output [1:2] \B_A_y<6><m4>;  output \B_A_B_z<n6><4><j2>;
```

Naming Parameterized Modules

- Specify the format of module names generated for parameterized modules using the hdl_parameter_naming_style attribute. For example:

  ```
  set_attribute hdl_parameter_naming_style "_%s%d"
  ```

The elaborate command automatically elaborates the design by propagating parameter values specified for instantiation, as shown in Example 6-3. In this Verilog example, the elaborate command builds the modules TOP and BOT, derived from the instance u0 in design TOP. The actual 7 and 0 values of the two L and R parameters provided with the u0 instance override the default values in the module definition for BOT. The final name of the subdesign will be BOT_L7_R0.

Example 6-3 Automatic Elaboration

```
module BOT(o);
  parameter L = 1;
  parameter R = 1;
output [L:R] o;

  assign o = 1'b0;
endmodule

module TOP(o);
  output [7:0] o;

  BOT #(7,0) u0(o);
endmodule
```

Example 6-4 is a VHDL design that will be used to show how specify different suffix formats using the hdl_parameter_naming_style attribute.

Example 6-4 Test VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity top is
    port (d_in : in std_logic_vector(63 downto 0);
          d_out : out std_logic_vector(63 downto 0));
end top;

architecture rtl of top is
    component core
        generic (param_1st : integer := 7;
                 param_2nd : integer := 4  );
        port (  d_in : in std_logic_vector(63 downto 0);
                d_out : out std_logic_vector(63 downto 0)
);
    end component;
begin
    u1 : core
        generic map (param_1st => 1, param_2nd => 4)
        port map (d_in => d_in, d_out => d_out);
    ....
end rtl;
```

If you specify the `_%s_%d` suffix format as shown in the VHDL Example 6-5, then the modules names in the netlist will be as shown in Example 6-6.

Example 6-5 set_attribute hdl_parameter_naming_style _%s_%d

```
set_attr hdl_parameter_naming_style "_%s_%d"
set_attr library tutorial.lbr
read_hdl -vhdl test.vhd
elaborate top
write_hdl
```

Example 6-6 Netlist With the hdl_parameter_naming_style_%s_%d Suffix Format

```
module core_param_1st_7_param_2nd_4 (d_in, d_out);
    input [63:0] d_in;
    output [63:0] d_out;
endmodule

module top (d_in, d_out);
    input [63:0] d_in;
    output [63:0] d_out;
    core_param_1st_7_param_2nd_4 u1 (.d_in(d_in),.d_out(d_out));
    ...
endmodule
```

If you specify the `_%s%d` default suffix format as shown in Example 6-7, then the modules names in the netlist will be as shown in Example 6-8.

Example 6-7 set_attribute hdl_parameter_naming_style "_%s%d"

```
set_attr hdl_parameter_naming_style "_%s%d"
set_attr library tutorial.lbr
read_hdl -vhdl test.vhd
elaborate top
write_hdl
```

Example 6-8 Netlist with the Default hdl_parameter_naming_style Suffix Format

```
module core_param_1st7_param_2nd4 (d_in, d_out);
    input [63:0] d_in;
    output [63:0] d_out;
endmodule

module top (d_in, d_out);
    input [63:0] d_in;
    output [63:0] d_out;
    core_param_1st7_param_2nd4 u1 (.d_in(d_in),.d_out(d_out));
    ...
endmodule
```

If you specify the_`%d` suffix format as shown in the VHDL Example 6-9, then the modules names in the netlist will be as shown in Example 6-10.

Example 6-9 set_attribute hdl_parameter_naming_style "_%d"

```
set_attr hdl_parameter_naming_style "_%d"
set_attr library tutorial.lbr
read_hdl -vhdl test.vhd
elaborate top
write_hdl
```

Example 6-10 Netlist With the hdl_parameter_naming_style "_%d" Suffix Format

```
module core_7_4 (d_in, d_out);
    input [63:0] d_in;
    output [63:0] d_out;
endmodule

module top (d_in, d_out);
    input [63:0] d_in;
    output [63:0] d_out;
    core_7_4 u1 (.d_in(d_in), .d_out(d_out));
    ...
endmodule
```

Overriding Top-Level Parameter or Generic Values

While automatic elaboration works for designs that are instantiated in a higher level design, some applications require an override of the default parameter or generic values directly from the `elaborate` command, as in elaborating top-level modules or entities with different parameters or generic values.

Overriding Top-Level Parameter Values by Positional Associations

- Override the default parameter values using the `-parameters` option with the `elaborate` command, as shown in Example 6-11. This option specifies the values to use for the indicated parameters.

Example 6-11 Overriding the Default Top-Level Parameter Values

```
//Synthesizing the design TOP with parameter values L=3 and R=2:
elaborate TOP -parameters {3 2}
//yields the following output:
Setting attribute of root /: 'hdl_parameter_naming_style' = _%s%d
Setting attribute of root /: 'library' = tutorial.lbr
Elaborating top-level block 'TOP_L3_R2' from file 'ex11.v'.
Done elaborating 'TOP_L3_R2'
```

Overriding Top-Level Parameter Values by Named Associations

➤ Override top-level parameter values using the `-parameters` option with the `elaborate` command using named associations as follows:

```
elaborate -parameters { {name1 value1} {name2 value2} ...} [module...]
```

The default top-level module is built. If fewer parameters are specified than exist in the design, then the default values of the missing parameters will be used in building the design. If more parameters are specified than exist in the design, then the extra parameters are ignored.

➤ Synthesize the `ADD` design with the parameter or generic values `L=0` and `R=7` using the following command:

```
elaborate ADD -parameters {{L 0} {R 7}}
```

➤ To synthesize all bit widths for the adder `ADD` from `1` through `16`, use:

```
foreach i {0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15} {
    eval elaborate ADD -parameters "{{L 0} {R [expr $i]}}"
```

Keeping Track of the RTL Source Code

➤ Set the following attribute to `true` to keep track of the RTL source code:

```
set_attribute hdl_track_filename_row_col { true | false }
```

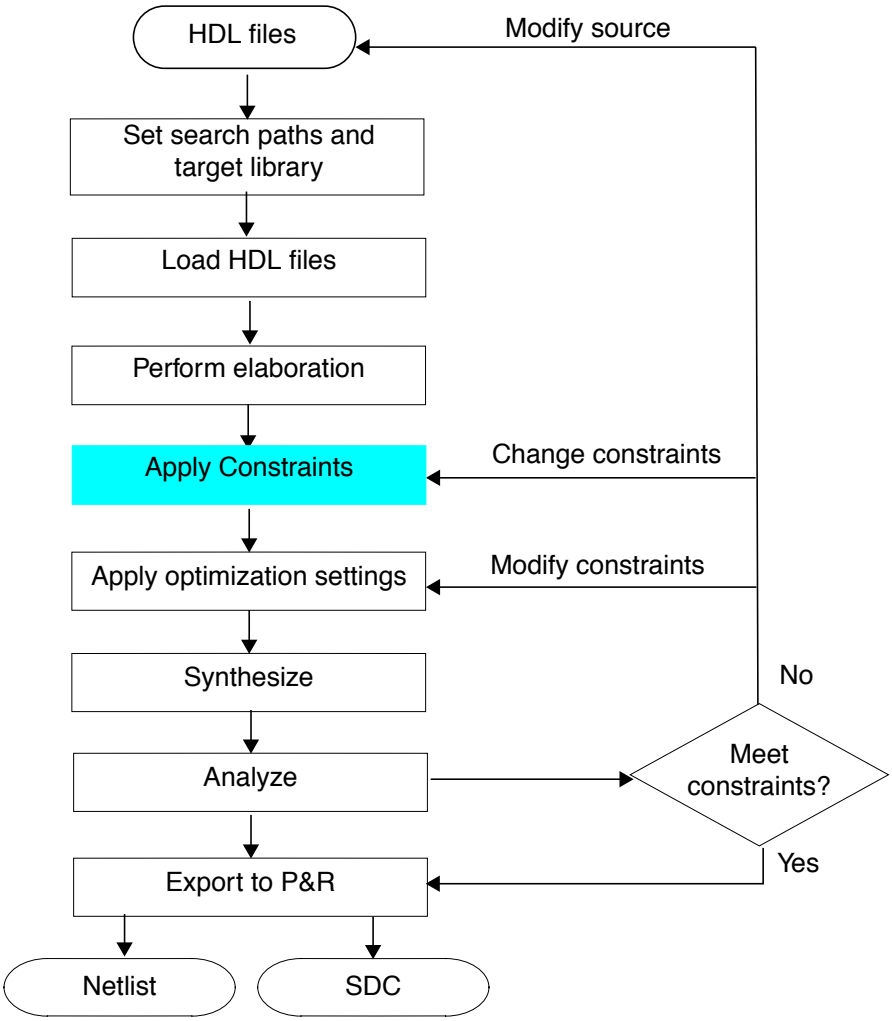
Default: `false`

This attribute enables RTL Compiler to keep track of filenames, line numbers, and column numbers for all instances before optimization. RTL Compiler also uses this information in subsequent error and warning messages.

7

Applying Constraints

Overview



This chapter describes how to apply the basic constraints in RTL Compiler. For a detailed description on how to use constraints in RTL Compiler see *Setting Constraints and Performing Timing Analysis in Encounter RTL Compiler*.

Tasks

Importing and Exporting SDC

RTL Compiler provides the ability to read in and write out SDC constraints.

➤ To import SDC constraints, use the `read_sdc` command:

```
rc:/> read_sdc filename
```

➤ To export SDC constraints use the `write_sdc` command:

```
rc:/> write_sdc > filename
```

Applying Timing Constraints

In RTL Compiler, a clock waveform is a periodic signal with one rising edge and one falling edge per period. Clock waveforms may be applied to design objects such as input ports, clock pins of sequential cells, external clocks (also known as virtual clocks), mapped cells, or hierarchical boundary pins.

- To define clocks use the `define_clock` command.

Note: RTL Compiler uses picoseconds and femtoferads as timing units. It *does not* use nanoseconds and picoferads.

You can group clocks that are synchronous to each other, allowing timing analysis to be performed between these clocks. This group is called a clock domain. If a clock domain is not specified, RTL Compiler will assume all the clocks are in the same domain.

By default, RTL Compiler assigns clocks to `domain_1`, but you can create your own domain name with the `-domain` argument to `define_clock`.

The following example demonstrates how to create two different clocks and assign them to two separate clock domains:

```
rc:/> define_clock -domain domain1 -name clk1 -period 720 [find / -port SYSCLK]
rc:/> define_clock -domain domain2 -name clk2 -period 720 [find / -port CLK]
```

To remove clocks, use the rm command. If you have defined a clock and saved the object variable, for example as `clock1`, you can remove the clock object as shown in the following example:

```
rc:/> rm $clock1
```

The following example shows how to remove the clock if you have not saved the clock object as a variable:

```
rc:/> rm [find / -clock clock_name]
```

When a clock object is removed, external delays that reference it are removed, and timing exceptions referring to the clock are removed if they cannot be satisfied without the clock.

For more detailed information on timing constraints, see *Setting Constraints and Performing Timing Analysis in Encounter RTL Compiler*.

Using Physical Layout Estimators

RTL Compiler uses physical layout estimators (PLEs) to incorporate physical information into synthesis. PLEs use equations instead of wire-load models to drive synthesis. Specifically, PLEs:

- Use bounding boxes and fanout to dynamically derive wire length
- Calculate load and delay using average resistance (in OHMs per micron) and capacitance (in pF per micron) per unit length. The resistance and capacitance are derived from the process technology information.
- Calculate wire area in microns using the average net width from the process technology information

The need for wire-load models is eliminated with PLEs. The wire-load column from the `report area` output will consequently be empty.

To use PLEs in your synthesis flow, first instruct RTL Compiler to use physical information with the `ple` argument of the `interconnect_mode` attribute.

```
rc:/> set_attribute interconnect_mode ple /
```

Next, specify a LEF or a capacitance table file, or both, with the `lef_library` and `cap_table_file` attributes. If you have already specified your library path with the `lib_search_path` attribute, you do not need to give the full UNIX for these files like the following example:

```
rc:/> set_attribute lib_search_path /home/stormy/designs/lib
rc:/> set_attribute lef_library teagan.lef
rc:/> set_attribute cap_table_file stormy.cap
```

Tip

Since most capacitance table files do not include resistance information, specify both the LEF and capacitance table files. However, you can also specify only one if the other is not available.

Applying Design Rule Constraints

When optimizing a design, RTL Compiler tries to satisfy all design rule constraints (DRCs). Examples of DRCs include maximum transition, fanout, and capacitance limits; operating conditions; and wire-load models.These constraints are specified using attributes on a module or port, or from the technology library. However, even without user-specified constraints, rules may still be inferred from the technology library.

To specify a maximum transition limit for all nets in a design or on a port, use the `max_transition` attribute on a top-level block or port:

```
rc:/> set_attribute max_transition value [design|port]
```

To specify a maximum fanout limit for all nets in a design or on a port, use the `max_fanout` attribute on a top-level block or port:

```
rc:/> set_attribute max_fanout value [design|port]
```

To specify a maximum capacitance limit for all nets in a design or on a port, use the `max_capacitance` attribute on a top-level block or port:

```
rc:/> set_attribute max_capacitance value [design|port]
```

To specify a specific wire-load model to be used during synthesis, use the `force_wireload` attribute. The following example specifies the `1x1` wire-load model on a design named `stormy`:

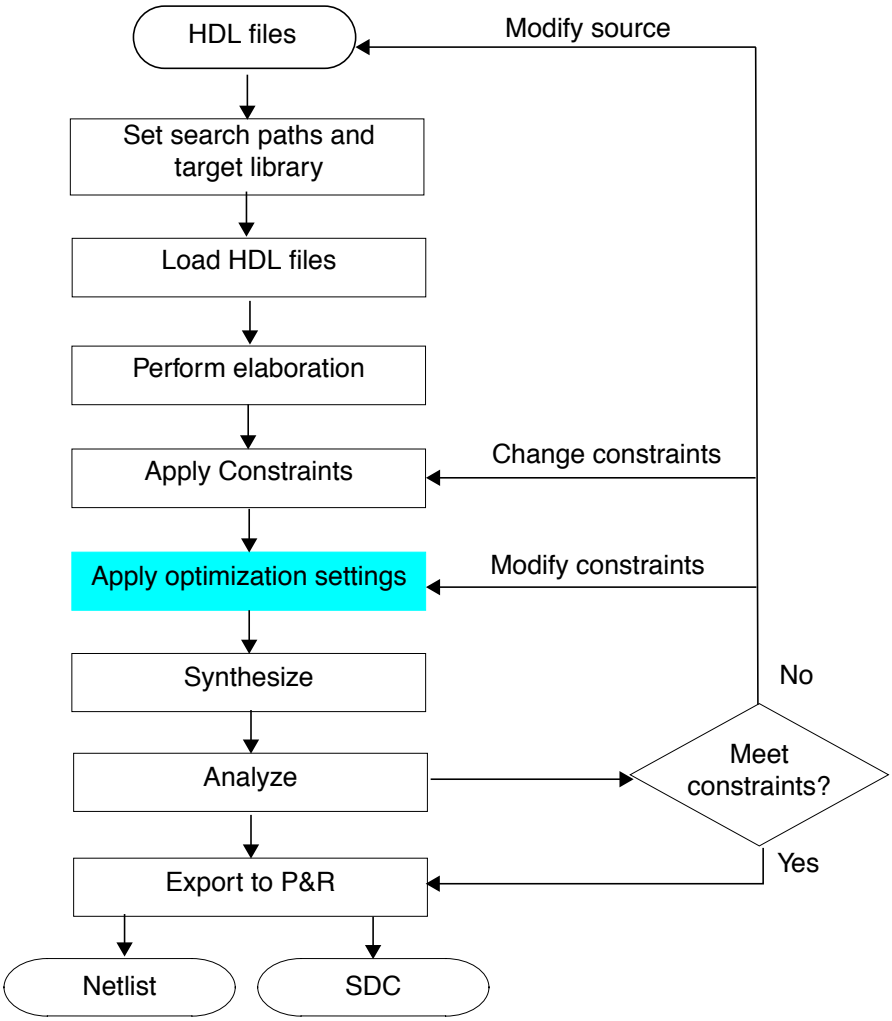
```
rc:/> set_attribute force_wireload 1x1 stormy
```

For a more detailed information on DRCs, see *Setting Constraints and Performing Timing Analysis in Encounter RTL Compiler*.

8

Defining Optimization Settings

Overview



This chapter describes how to apply optimization settings to your design before synthesis.

Tasks

Preserving Instances and Modules

Use the `preserve` attribute to prevent RTL Compiler from performing optimizations on specific objects in the design.

By default, RTL Compiler will perform optimizations that can result in logic changes to any object in the design. You can prevent any logic changes in a block while still allowing mapping optimizations in the surrounding logic, as shown below.

➤ To preserve hierarchical instances, type the following command:

```
rc:/> set_attribute preserve true object
```

where *`object`* is a hierarchical instance name.

➤ To preserve primitive instances, type the following command:

```
rc:/> set_attribute preserve true object
```

where *`object`* is a primitive instance name.

➤ To preserve modules or submodules, type the following command:

```
rc:/> set_attribute preserve true object
```

where *object* is a module or submodule name.

In RTL Compiler, you can preserve and protect your existing mapped logic from being further synthesized or changed. Preserve logic with the `preserve` attribute on the desired module or the instance.

The `preserve` attribute can be used on a mapped sub-design to prevent any logic changes in the block while allowing mapping optimizations to be performed in surrounding logic.

The default value of this attribute is `false`. Therefore, you must set it to `true` to preserve the module. The following example preserves the module `counter`:

```
rc:/> set_attribute preserve true [find / -subdesign counter]
```

The following example preserves a mapped leaf cell instance, `g1`.

```
rc:/> set_attribute preserve true [find / -instance g1]
```

This command will force logic optimization to preserve the current mapping of the `g1` instance.

Modifying Instances

RTL Compiler offers the following additional features to give you more flexibility with preserved instances or modules during synthesis:

- `The size_ok` argument enables RTL Compiler to preserve an instance (`g1` in the example below), while allowing it to be resized.

  ```
  rc:/> set_attribute preserve size_ok [find / -instance g1]
  ```

- The `delete_ok` argument allows RTL Compiler to delete an instance (`g1` in the example below), but not to rename, remap, or resize it.

  ```
  rc:/> set_attribute preserve delete_ok [find / -instance g1]
  ```

- `The size_delete_ok` argument allows RTL Compiler to resize or delete an instance (`g1` in the example below), but not to rename or remap it.

  ```
  rc:/> set_attribute preserve size_delete_ok [find / -instance g1]
  ```

Grouping and Ungrouping Objects

RTL Compiler provides a set of commands that enable you to group or ungroup any existing instances, designs, or subdesigns. Grouping and ungrouping are helpful when you need to change your design hierarchy as part of your synthesis strategy.

- *Grouping* builds a level of hierarchy around a set of instances.
- *Ungrouping* flattens a level of hierarchy.

Grouping

If your design includes several subdesigns, you can group some of the subdesign instances into another single subdesign for placement or optimization purposes using the `edit_netlist group` command.

For example, the following command creates a new subdesign called `CRITICAL_GROUP` that includes instances `I1` and `I2`.

```
rc:/> edit_netlist group -group_name CRITICAL_GROUP [find / -instance I1] \
    [find / -instance I2]
```

The new instance name for this new hierarchy will be `CRITICAL_GROUP`, and it will be placed in the directory path:

```
/designs/top_counter/instances_hier/CRITICAL_GROUP
```

Ungrouping

To flatten a hierarchy in the design, use the `ungroup` command.

```
rc:/> ungroup instance
```

where *`instance`* is the name of the instances to be ungrouped.

If you need to ungroup the design hierarchy `CRITICAL_GROUP` (which contains instances `I1` and `I2`), use the `ungroup` command along with the instance name as shown below:

```
rc:/> ungroup [find / -instance CRITICAL_GROUP]
```

Note: RTL Compiler will respect all preserved instances in the hierarchy. For more information on preserving instances, see "Preserving Instances and Modules" on page 155.

Setting Boundary Optimization

RTL Compiler performs boundary optimization for all hierarchical instances in the design during synthesis. Examples of boundary optimizations include:

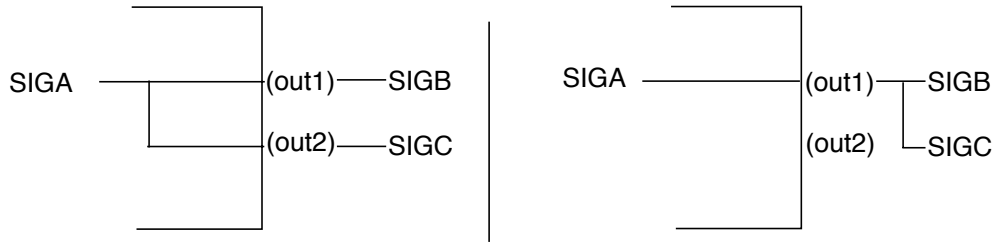
- Constant propagation across hierarchies

  Both through input ports and output ports
- Rewiring of equivalent signals across hierarchy

If two outputs of a module are identical, RTL Compiler may disconnect one of them and use the other output to drive the fanout logic for both.

RTL Compiler may also rewire opposite signals which are functionally equivalent, but of opposite polarity, as illustrated in Figure 8-1 on page 158. The figure on the left shows `SIGA` fanning to two output ports, `out1` and `out2`. These are connected to `SIGB` and `SIGC`, respectively. The figure on the right shows that RTL Compiler can change that signal so that `SIGB` and `SIGC` are both connected to `out1`.

Figure 8-1 Rewiring Equivalent Signals



You can control boundary optimization during synthesis using the following two attributes:

- `boundary_opto`
- `delete_unloaded_seqs`

If you cannot perform top-down formal verification on the design, then turn off boundary optimization for sub-blocks that will be individually verified.

➤ To disable boundary optimization on the subdesign, type the following command:

```
rc:/> set_attribute boundary_opto false [find /des* -subdesign name]
```

By default RTL Compiler will remove flip-flops and logic if they are not transitively fanning out to output ports. To prevent this, use the `delete_unloaded_seqs` attribute as shown below:

```
rc:/> set_attribute delete_unloaded_seqs false [subdesigns or /]
```

Mapping to Complex Sequential Cells

The sequential mapping feature of RTL Compiler takes advantage of complex flip-flops in the library to improve the cell count of your design, and sometimes the area or timing (depending on the design).

RTL Compiler performs sequential mapping when the flops are inferred in RTL. For instantiated flops, other than sizing, RTL Compiler performs no other optimization.

Asynchronous flip-flop inputs are automatically inferred from the sensitivity list and the conditional statements within the `always` block.

➤ To keep the synchronous feedback logic immediately in front of the sequential elements, type the following command:

```
rc:/> set_attribute rtl_DFF_synchro_load_enable 1
```

Setting this attribute may have a negative impact on the area and timing. RTL Compiler currently does not support multi-bit register bank inference from the library.

Deleting Unused Sequential Instances

RTL Compiler optimizes sequential instances that transitively do not fanout to primary output. This information is generated in the log file. This is especially relevant if you see unmapped points in formal verification.

```
Deleting 2 sequential instances. They do not transitively drive any primary
outputs:
    ifu/xifuBtac/xicyBtac/icyBrTypeHold1F_reg[1] (floating-loop root),  ifu/
xifuBtac/xicyBtac/icyBrTypeHold1T_reg[1]
```

➤ To prevent the deletion of unloaded sequential instances, set the `delete_unloaded_seqs` attribute to `false`. The default value of this attribute is `true`.

```
rc:/> set_attribute delete_unloaded_seqs false /
```

➤ To prevent constant `0` propagation through flip-flops, set the `optimize_constant_0_flops` attribute to `false`. The default value of this attribute is `true`.

```
rc:/> set_attribute optimize_constant_0_flops false /
```

➤ To prevent constant `1` propagation through flip-flops, set the `optimize_constant_1_flops` attribute to `false`. The default value of this attribute is `false`.

```
rc:/> set_attribute optimize_constant_1_flops false /
```

➤ To prevent constant propagation through latches set the `optimize_constant_latches` to `false`. The default value of this attribute is `true`.

```
rc:/> set_attribute optimize_constant_latches false /
```

Optimizing Total Negative Slack

By default, RTL Compiler optimizes Worst Negative Slack (WNS) to achieve the timing requirements. During this process, it tries to fix the timing on the most critical path. It also checks the timing on all the other paths. However, RTL Compiler will not work on the other paths if it cannot improve timing on the WNS.

➤ To make RTL Compiler work on all the paths to reduce the total negative slack (TNS), instead of just WNS, type the following command:

```
rc:/> set_attribute endpoint_slack_opto true /
```

Ensure that you specify the attribute on the root-level ("/"). This attribute instructs RTL Compiler to work on all the paths that violate the timing and try to reduce their slack as much as possible.

This may cause the run time and area to increase, depending on the design complexity and the number of violating paths.

Creating Hard Regions

Use the `hard_region` attribute to specify hierarchical instances that are recognized as hard regions in your floorplan during logic synthesis.

Place and route tools operate better if your design has no buffers between regions at the top level. To accommodate this, specify hard regions before mapping.

To create hard regions, follow these steps:

1. Specify the hard region, for example `pbu_ctl`, by typing:

   ```
   set_attribute hard_region 1 [find / -instance pbu_ctl]
   ```

2. Eliminate buffers and inverter trees between hard regions using the variable `map_rm_hr_driven_buffers`, even if design timing gets worse, by typing:

   ```
   set map_rm_hr_driven_buffers 1
   ```

   Primary inputs and outputs are also treated as hard regions for this purpose.

3. Run the `synthesize -to_mapped` command.

The regular boundary optimization related controls are also applicable to hard regions.

Deleting Buffers and Inverters Driven by Hard Regions

To prepare your design for place and route tools, you need to remove the buffer and inverter trees between hard regions. You can specify that any buffers or inverters driven by a hard region be deleted by setting the `map_rm_hr_driven_buffers` variable to `1`.

➤ To remove buffers and inverters, type the following command:

```
rc:/> set map_rm_hr_driven_buffers 1
```

This instructs RTL Compiler to eliminate the buffers and inverters between hard regions, even if doing so degrades design timing. Primary inputs and outputs are treated as hard regions for this purpose.

Where possible, inverters will be paired up and removed, or RTL Compiler will try to push them back into the driving hard region. Otherwise, the inverter is left alone because orphan buffers, buffers that do not belong to any region, can be placed anywhere during place and route. The backend flows can address this kind of buffering. The regular boundary optimization controls are applicable to hard regions.

Note: Timing may become worse due to this buffer removal. This clean-up phase occurs before writing out the netlist for place and route.

Preventing Boundary Optimization Through Hard Regions

If you want to prevent boundary optimization through identified hard regions, type the following command before performing synthesis or technology mapping:

```
rc:/> set boundary_opt_down_hard_regions 0
```

Making DRC the Highest Priority

By default, RTL Compiler tries to fix all DRC errors, but not at the expense of timing. If DRCs are not being fixed, it could be because of infeasible slew issues on input ports or infeasible loads on output ports. You can force RTL Compiler to fix DRCs, even at the expense of timing, with the `drc_first` attribute.

➤ To ensure DRCs get solved, even at the expense of timing, type the following command:

```
rc:/> set_attribute drc_first true
```

By default, this attribute is `false`, which means that DRCs will not be fixed if it introduces timing violations.

9

Super-threading

Overview

RTL Compiler's super-threading capability allows you to reduce synthesis turn-around time by distributing the processing work across multiple CPUs. The CPUs may be on the same machine on which RTL Compiler was launched, or it may be on different machines across a network.

Unlike previous approaches to parallelize logic synthesis, RTL Compiler's capability is unique: the synthesis results produced through parallel processing are identical to those produced in a non-parallel run. Therefore, only the turn-around time is affected.

Licensing Requirements

When the remote servers are launched, the main process will try to acquire an `RTL_Compiler_Ultra_II` license. If this license is not available, optimization will continue without super-threading.

- The first remote server requires no extra licenses.
- Each remote server after the first will require its own `RTL_Compiler_Ultra` license. If this license is not available, optimization will continue without that server.
- If you want to reserve licenses in advance, use the `license` command. That is, use the `license` command to check out licenses before super-threading optimization starts.

  See the `license` command in the *Command Reference for Encounter RTL Compiler* for more information about its usage.

Note: Instead of an `RTL_Compiler_Ultra` license you can also use one of the following licenses: `FE_GPS`, `SOC_Encounter_GPS`, or `RTL_Compiler_Verification`.

Tasks

Super-threading is available in the global map portion of the `synthesize -to_mapped` command. Enable super-threaded optimization by setting the root attribute `super_thread_servers`:

```
rc:/> set_attribute super_thread_servers {machine_names}
```

This attribute should be set to a Tcl list representing the set of machines on which RTL Compiler can launch processes to super-thread.

Setting Super-threading Optimization

Setting the `super_thread_servers` attribute does not cause the server processes to be launched immediately. Instead, the processes are launched just before the super-threaded optimization starts and are automatically shut down when the optimization completes.

RTL Compiler launches the server processes using the UNIX command `rsh`. Before setting the `super_thread_servers` attribute, ensure you can execute the following command without getting any errors or being prompted for a password:

```
unix> rsh machine_name -c echo hello world
```

If you are prompted for a password, you may need to set up a `~/.rhosts` file. See the UNIX manpage for `rsh` for more information.

The following examples illustrate how to set super-threading optimization in various scenarios:

- The following example illustrates RTL Compiler launching a single process for super-threaded optimization on the machine in which RTL Compiler is currently running. In super-threaded optimization mode, the original process merely serves as a work-dispatcher. This will only slow synthesis compared to if no super-threading were specified due to inter-process communication overhead.

  ```
  rc:/> set_attribute super_thread_servers {localhost} /
  ```

- The following example illustrates RTL Compiler launching three processes on the current machine for super-threading. If the current machine has at least three CPUs, a turn-around time reduction of up to 3x can be achieved.

  ```
  rc:/> set_attribute super_thread_servers {localhost localhost localhost} /
  ```

- RTL Compiler will support super-thread servers of different platform types. The following example illustrates RTL Compiler launching two server processes on the machine called `linux33` and two processes on `sun42`.

  ```
  rc:/> set_attribute super_thread_servers {linux33 linux33 sun42 sun42} /
  ```

- In the following example, three processes are specified on `linux21`:

  ```
  rc:/> set_attribute super_thread_servers {linux21 linux21 linux21}
  ```

  However, there are only two CPUs on `linux21`. In this case, three super-thread server processes will be launched but the operating system will divide processing time among the two existing CPUs so that each process only gets 2/3 of a CPU.

  The turn-around time will be negligible compared to if only two super-thread servers has been specified.

- To turn off super-threading specify an empty list:

  ```
  rc:/> set_attribute super_thread_servers {}
  ```

- RTL Compiler supports super-threading on LSF. The following command launches two server processes on `linux33` and two on LSF:

  ```
  rc:/> set_attribute super_thread_servers {linux33 linux33 lsf lsf} /
  ```

  You can also launch processes just on LSF. The following example launches three processes on LSF:

  ```
  rc:/> set_attribute super_thread_servers {lsf lsf lsf} /
  ```

  If you want to pass options to `bsub`, use the `bsub_options` attribute in conjunction with `super_thread_servers`:

  ```
  rc:/> set_attribute bsub_options some_options /
  rc:/> set_attribute super_thread_servers {lsf lsf} /
  ```

  If you invoked RTL Compiler with the `-lsf_cpus` and `-lsf_queue` options, they will be overridden by the parameters given to the `super_thread_servers` and `bsub_options` attributes. In the following example two processes will be sent to LSF and the `stormy queue` used:

  ```
  unix> rc -lsf_cups 4 -lsf_queue teagan_queue
  ...
  rc:/> set_attribute super_thread_servers {lsf lsf} /
  rc:/> set_attribute bsub_options stormy_queue /
  ```

If a specified super-thread server does not reply in two minutes (120 seconds), the process will be sent back to the dispatcher to process. Only the process on the non-responsive server will be affected: all other processes will be honored.

- In the following example, processes are specified on `linux33`, `linux41`, and `linux51`. However, `linux33` is too busy and cannot respond within the two minute time frame. As a result, the processes on `linux41` and `linux51` will continue but the process specified on `linux33` will be sent back to the machine that dispatched the processes (localhost, for example).

  ```
  rc:/> set_attribute super_thread_servers {linux33 linux41 linux51}
  ```

- Use the following example to measure the "wall clock" time of a process. The wall clock time is the elapsed time as opposed to the CPU time, which is returned by the `runtime` attribute. In this example, the wall clock time of the `synthesize -to_mapped` command is measured:

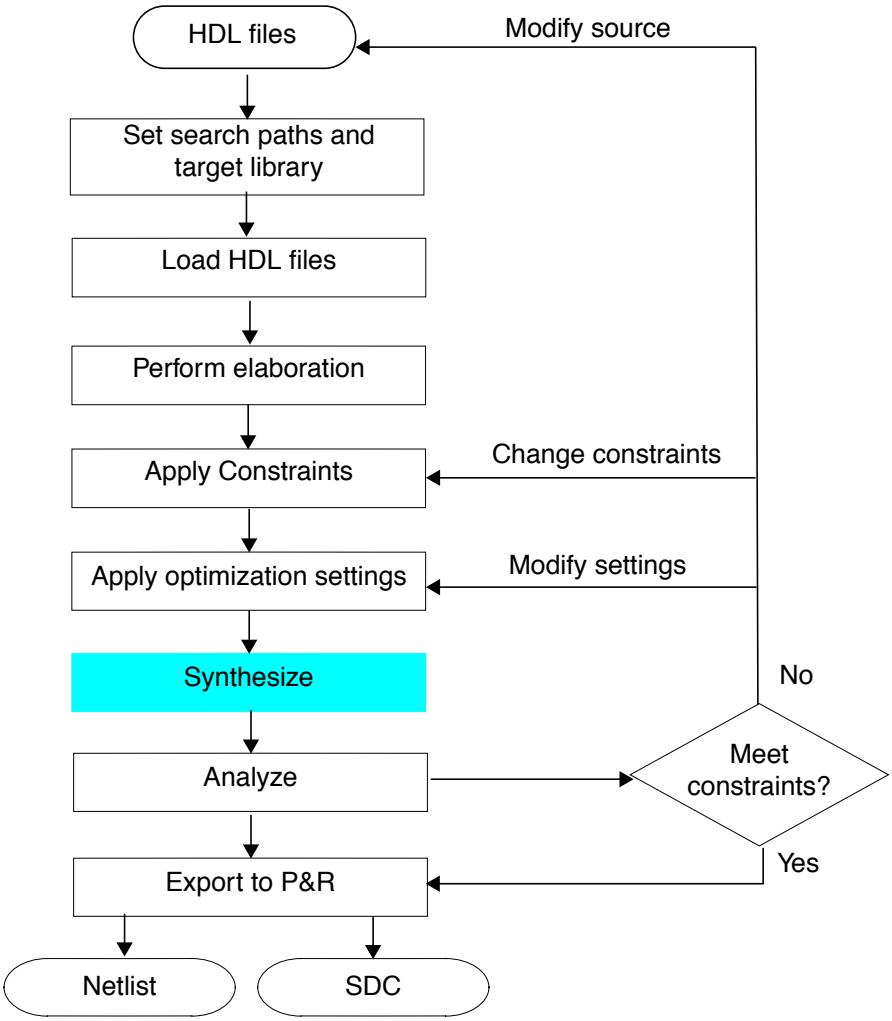
```
rc:/> set_attribute super_thread_servers {dani1 dani2}
...
rc:/> set start_ms [clock clicks -milliseconds]>
rc:/> synthesize -to_mapped>
rc:/> set elapsed_seconds [expr {([clock clicks -milliseconds] \
    $start_ms) / 1000.0}]


33.611
```

  We see that the command took 33.611 seconds to complete.

10

Performing Synthesis

Overview



Synthesis is the process of transforming your HDL design into a gate-level netlist, given all the specified constraints and optimization settings.

In RTL Compiler, synthesis involves the following four processes:

- RTL Optimization
- Global Focus Mapping
- Remapping
- Incremental optimization

RTL Optimization

During RTL optimization, RTL Compiler performs optimizations like datapath synthesis, resource sharing, speculation, mux optimization, and carrysave arithmetic (CSA) optimizations. After this step, RTL Compiler performs logic optimizations like structuring and redundancy removal.

For more information on datapath synthesis, see *Datapath Synthesis in Encounter RTL Compiler*.

Global Focus Mapping

RTL Compiler performs global focus mapping at the end of the RTL technology-independent optimizations (during the `synthesize –to_mapped` command).

This step includes restructuring and mapping the design concurrently, including optimizations like splitting, pin swapping, buffering, pattern matching, and isolation.

Remapping

After Global Focus Mapping, RTL Compiler performs synthesis remapping. During this phase, RTL Compiler only performs global sizing of cells. There are actually multiple remapping phases: some are targeted at area optimization while others at timing optimization.

Incremental Optimization (IOPT)

The final optimization RTL Compiler performs is incremental optimization. Optimizations performed during IOPT improve timing and area and fix DRC violations.

By default, RTL Compiler does not fix DRC violations if doing so causes timing violations. This priority can be over ridden by setting the `drc_first` attribute to `true`. By default, timing has the highest priority.

IOPT also includes Critical Region Resynthesis (CRR) which iterates over a small window on the critical path to improve slack. You can control CRR through the `effort` level argument in the `synthesize` command. It is asserted by specifying the `high` effort level.

Tasks

Synthesizing your Design

After you set the constraints and optimizations for your design, you can proceed with synthesis using the `synthesize` command. Synthesis is performed in two steps:

1. Synthesizing the design to generic logic (RTL optimizations are performed in this step).
2. Mapping to the technology library and performing incremental optimization.

These two sequential steps are represented by the `synthesize` command options `-to_generic` and `-to_mapped` (see Table 10-1):

- The `synthesize -to_generic` command performs RTL optimization on your design.

  Note: When the `synthesize` command is issued on an RTL design without any arguments, the `-to_generic` option is implied.

- The `synthesize -to_mapped` command maps the specified design(s) to the cells described in the supplied technology library and performs logic optimization.

  The goal of optimization is to provide the smallest possible implementation of the design that satisfies the timing requirements. The three main steps performed by `synthesize -to_mapped` option are:

- Technology-independent Boolean optimization
- Technology mapping
- Technology-dependent gate optimization

The `synthesize -to_mapped` command queries the library for detailed timing information. After you use the `synthesize -to_mapped` command to generate an optimized netlist, you can analyze the netlist using the `report` command and output it to a file using the `write` command. For more information on the `write` command, see "Writing Out the Design Netlist" on page 225.

Table 10-1 shows a matrix of actions performed by the `synthesize` command depending on the state of the design and the option specified.

Table 10-1 Actions Performed by the synthesize Command

Specified Option	Current Design State		
	RTL	**Generic**	**Mapped**
No Option Specified	■ RTL Optimization	■ Mapping ■ Incremental Optimizations	■ Unmapping ■ Remapping ■ Incremental Optimizations
`-to_generic`	■ RTL Optimization	■ Nothing	■ Unmapping
`-to_mapped`	■ RTL Optimization ■ Mapping ■ Incremental Optimizations	■ Mapping ■ Incremental Optimizations	■ Unmapping ■ Remapping ■ Incremental Optimizations

Synthesizing Submodules

In RTL Compiler you can have multiple designs, each with its own design hierarchy. The `synthesize` command allows you to synthesize any of these top-level designs separately.

Whenever you need to synthesize any submodule in your design hierarchy, use the derive_environment command to promote this subdesign to a top-level design. The steps below illustrate how to synthesize a submodule:

1. Elaborate the top-level design, in which the submodule is contained, with the elaborate command:

   ```
   rc:/> elaborate module_top
   ```

2. Apply constraints.

3. Synthesize the design to gates using the low effort level (to more accurately extract the constraints):

   ```
   rc:/> synthesize -to_mapped -effort low
   ```

4. Promote the submodule into a top-level module using the derive_environment command:

   ```
   rc:/> derive_environment -name <new_top> -instance <new_top_instance_name>
   ```

   The new_top module will have its own environment, since its constraints were derived from the top-level design. The new_top module will now be seen as another top-level module in the Design Information Hierarchy.

   ```
   rc:/> ls /designs


   ./    module_top/    new_top/
   ```

5. Write out the new_top design constraints using the write_script command:

   ```
   rc:/> write_script new_top > new_top.con
   ```

6. For the best optimization results, remove the derived new_top module, re-read in the HDL file, elaborate the new_top module, and then synthesize (in this case only the submodule will be synthesized):

   ```
   rc:/> rm new_top
   rc:/> read_hdl <read_RTL_files>
   rc:/> elaborate <new_top>
   rc:/> include new_top.con
   rc:/> synthesize
   ```

Note: Alternatively, you can re-synthesize new_top immediately after writing out the constraints without re-reading the HDL file. However, doing so might not provide the best optimization results:

```
rc:/> synthesize /designs/new_top
```

Synthesizing Unresolved References

In RTL Compiler, unresolved references are instances that do not have any library or module definitions. It is important to distinguish unresolved references from timing models: Timing models, also known as black boxes, are library elements that have timing information, but no functional descriptions.

The ports of unresolved references are considered to be directionless. Unresolved references tend to cause numerous multidrivers. RTL Compiler will maintain any logic leading into or out of the I/Os of unresolved references and treat them as unconstrained.

Re-synthesizing with a New Library (Technology Translation)

Technology translation and optimization is the process of using a new technology library to synthesize an already technology mapped netlist. The netlist is first read-in, and then "unmapped" to generic logic gates. The generic netlist would then be synthesized with the new library. The following example illustrates this process:

1. Read-in the mapped netlist using the -structural option of the `read_hdl` command:

   ```
   rc:/> read_hdl -structural mapped_netlist.v
   ```

2. Use Tcl in conjunction with the `preserve` and `avoid` attributes to allow the flip-flops in the design to be optimized and mapped according to the new library:

   ```
   rc:/> foreach cell [find /lib* -libcell *] {
   ==> set_attribute preserve false $cell
   ==> set_attribute avoid false $cell
   ==> }
   ```

3. Unmap the netlist to generic gates using the `-to_generic` option of the `synthesize` command:

   ```
   rc:/> synthesize -to_generic
   ```

4. Write-out the generic netlist:

   ```
   rc:/> write_hdl -generic > generic_netlist.v
   ```

5. Remove the design from the design information hierarchy:

   ```
   rc:/> rm /designs/*
   ```

6. Set the new technology library:

   ```
   rc:/> set_attribute library new_library.lib
   ```

7. Re-read and elaborate the generic netlist

   ```
   rc:/> read_hdl generic_netlist.v
   rc:/> elaborate
   ```

8. Apply constraints and Synthesize to technology mapped gates using the `-to_mapped` option of the `synthesize` command:

   ```
   rc:/> synthesize -to_mapped
   ```

After the final step, proceed with your RTL Compiler session.

Setting Effort Levels

You can specify an effort level by using the `-effort {low | medium | high}` option with the `synthesize` command. The possible values for the `-effort` option are as follows:

- `low`

  The design is mapped to gates, but RTL Compiler does very little RTL optimization, incremental clean up, or redundancy identification and removal. The low setting is generally not recommended.

- `medium` (default setting)

  RTL Compiler performs better timing-driven structuring, incremental synthesis, and redundancy identification and removal on the design.

- `high`

RTL Compiler does the timing-driven structuring on larger sections of logic and spends more time and makes more attempts on incremental clean up. This effort level involves very aggressive redundancy identification and removal.

Generic Gates in a Generic Netlist

RTL Compiler can write out a generic netlist, read it back in, and restore circuitry written into the netlist. In this process, the generic netlist may have some *generic gates* that are defined and understood by RTL Compiler.

There are four kinds of generic gates:

- Generic Flop

  `CDN_flop`

- Generic Latch

  `CDN_latch`

- Generic Mux

  `CDN_mux2`

  `CDN_mux3`

  `CDN_mux4`

  `CDN_mux5`

  ...

- Generic Dont-Care

  `CDN_dc`

When seeing a generic gate in the design description, RTL Compiler has built-in knowledge about its input and output interface, its function, and its implementation.

Generic Flop

A `CDN_flop` is a generic edge-triggered flip-flop. The following shows the `CDN_flop` function and I/O interface:

Generic Flop CDN_flop

```
 module CDN_flop (clk, d, sena, aclr, apre, srl, srd, q);
    input clk, d, sena, aclr, apre, srl, srd;
    output q;
    reg  qi;
    assign #1 q = qi;
    always @(posedge clk or posedge apre or posedge aclr)
        if (aclr)
                        qi = 0;
        else if (apre)
                        qi = 1;
        else if (srl)
                        qi = srd;
        else
        begin
            if (sena)
                        qi = d;
        end
    initial
                        qi = 1'b0;
endmodule
```

Generic Latch

A `CDN_latch` is a generic level-triggered latch. The following example shows the `CDN_latch` function and I/O interface:

Generic Latch CDN_latch

```
module CDN_latch (ena, d, aclr, apre, q);
    input ena, d, aclr, apre;
    output q;
    reg  qi
    assign #1 q = qi;
    always @(d or ena or apre or aclr)
        if (aclr)
                              qi = 0;
        else if (apre)
                              qi = 1;
        else
        begin
             if (ena)
                               qi = d;
        end
    initial
                              qi = l'b0;
module
```

Generic Mux

The `CDN_mux*` gates are generic multiplexers. For example:

- `CDN_mux2` is a 2-to-1 mux
- `CDN_mux3` is a 3-to-1 mux
- `CDN_mux4` is a 4-to-1 mux
- `CDN_mux5` is a 5-to-1 mux

The following example shows the `CDN_mux2` function and I/O interface:

Generic Mux CDN_mux2

```
module CDN_mux2 (sel0, data0, sel1, data1, z);
    input sel0, data0, sel1, data1;
    output z;
    wire data0, data1, sel0, sel1;
    reg z;
    always @(sel0 or data0 or sel1 or data1)
        case ({sel0, sel1})
                2'b10:   z = data0;
                2'b01:   z = data1;
        default: z = 1'bx;
        endcase
endmodule
```

The following example shows the `CDN_mux3` function and I/O interface:

Generic Mux CDN_mux3

```
module CDN_mux3 (sel0, data0, sel1, data1, se12, data2, z);
    input sel0, data0, sel1, data1, se12, data2;
    output z;
    wire data0, data1, data2, sel0, sel1, sel2;
    reg z;
    always @(sel0 or data0 or sel1 or data1 or sel2 or data2)
        case ({sel0, sel1, sel2})
                3'b100:   z = data0;
                3'b010:   z = data1;
                3'b001:   z = data2;
                default:  z = 1'bx;
        endcase
endmodule
```

The following example shows the `CDN_mux5` function and I/O interface:

Generic Mux CDN_mux5

```
module CDN_mux5 (sel0, data0, sel1, data1,
        sel2, data2, sel3, data3, sel4, data4, z);
    input sel0, data0, sel1, data1,
        sel2, data2, sel3, data3, sel4, data4;
    output z;
    wire data0, data1, data2, data3, data4;
    wire sel0, sel1, sel2, sel3, sel4;
    reg z;
    always @(sel0 or data0 or sel1 or data1 or sel2 or
        data2 or sel3 or data3 or sel4 or data4)
    case ({sel0, sel1, sel2, sel3, sel4})
            5'b10000: z = data0;
            5'b01000: z = data1;
            5'b00100: z = data2;
            5'b00010: z = data3;
            5'b00001: z = data4;
            default:  z = 1'bx;
    endcase
endmodule
```

Generic Dont-Care

A `CDN_dc` is a dont-care gate. The following example shows the `CDN_dc` function and I/O interface:

Generic Dont-Care Gate CDN_dc

```
module CDN_dc (cf, dcf, z);
    input cf, dcf;
    output z;
    wire z;
    assign z = dcf ? 1'bx : cf;
endmodule
```

There are two input pins and one output pin.The `z` output pin is the data output. The `cf` input pin is the data input that provides the care function. The `dcf` input pin is an active-high dont-care control that provides the dont-care function. The output data is a dont-care, for example `1'bx`, if the dont-care control is active and if the `dcf` input is 1. The `CDN_dc` gate is a feed-

through from the `cf` input to the `z` output, if the dont-care control pin is inactive, such as if the `dcf` input is 0.

Writing the Generic Netlist

SYNTHESIS Macro

The `write_hdl -g` command describes these generic gates, but encloses each one with a pair of `ifdef-endif` Verilog compiler directives. For example:

```
`ifdef SYNTHESIS
`else
  module CDN_latch (ena, d, aclr, apre, q);
    ....
  endmodule
`endif
```

The if-branch is empty. To make it Verilog-1995 compatible, the tool does not use the `` `ifndef `` directive.

Using the `write_hdl -g` command may produce a netlist that has a mixture of Verilog primitives and RTL Compiler generic gates.

Example Generic Netlists

The following examples show how the generic gates are used in the generic netlist.

The following is the synthesis flow used in these examples:

```
set_attribute library tutorial.lib
set_attribute hdl_ff_keep_feedback false
read_hdl test.v
elaborate
write_hdl -g
```

Setting the `hdl_ff_keep_feedback` attribute to `false` tells RTL Compiler to use the sena logic inside of the generic flop to implement the load enable logic. If you do not set this attribute, RTL Compiler uses the glue logic outside of the generic flop to implement the load enable logic.

CDN_flop

With the following the RTL code shown in Example 10-1, RTL Compiler produces a netlist, such as shown in Example 10-2.

Example 10-1 RTL Code Inferring Flop With sync_set_reset

```
module test (q, d, clk, rstn, enb); // flop with sync set and reset
    input clk, rstn, enb, d;  output q;  reg q;
    // cadence sync_set_reset "rstn"
    always @(posedge clk)
    begin
        if (!rstn)      q = 1'b0;
        else if (enb)   q = d;
    end
endmodule
```

Example 10-2 Generic Netlist From Example 10-1

```
module test (q, d, clk, rstn, enb);
    input d, clk, rstn, enb;
    output q;
    wire d, clk, rstn, enb, q, rst;
    not g1 (rst, rstn);
    CDN_flop q_reg (.clk(clk), .d(d), .sena(enb),.aclr(1'b0), .apre(1'b0),
                    .srl(rst), .srd(1'b0), .q(q));
endmodule

`ifdef SYNTHESIS
`else
module CDN_flop (clk, d, sena, aclr, apre, srl, srd, q);
...
endmodule
`endif
```

Using the `sync_set_reset` pragma tells RTL Compiler to use the `srl` and `srd` logic inside of the generic flop to implement the sync set and reset logic. If you do not set this pragma, RTL Compiler uses the glue logic outside of the generic flop to implement the sync set and reset logic.

With the RTL code, as shown in Example 10-3, RTL Compiler produces a netlist, such as shown in Example 10-4.

Example 10-3 RTL Code Inferring Flop With async_set_reset

```
module test (q, d, clk, rstn, enb); // flop with async_set_reset
    input clk, rstn, enb, d;  output q;  reg q;
    always @(posedge clk or negedge rstn)
    begin
        if (!rstn)      q = 1'b0;
        else if (enb)   q = d;
    end
endmodule
```

Example 10-4 Generic Netlist From Example 10-3

```
module test (q, d, clk, rstn, enb);
    input d, clk, rstn, enb;
    output q;
    wire d, clk, rstn, enb, q, rst;
    not g1 (rst, rstn);
    CDN_flop q_reg (.clk(clk), .d(d), .sena(enb), .aclr(rst), .apre(1'b0),
                    .srl(1'b0), .srd(1'b0), .q(q));
endmodule

'ifdef SYNTHESIS
'else
module CDN_flop (clk, d, sena, aclr, apre, srl, srd, q);
...
endmodule
'endif
```

CDN_latch

With the following RTL code, as shown in Example 10-5, RTL Compiler produces a netlist, such as the one shown in Example 10-6.

Example 10-5 RTL Code Inferring Latch

```
module test (q, d, g, rstn); // latch
    input g, rstn, d;  output q;  reg q;
    // cadence async_set_reset "rstn"
    always @(d or g or rstn)
    begin
        if (!rstn)      q = 1'b0;
        else            if (g) q = d;
    end
endmodule
```

Example 10-6 Generic Netlist From Example 10-5

```
module test (q, d, g, rstn);
    input d, g, rstn;
    output q;
    wire d, g, rstn, q, rst;
    not g1 (rst, rstn);
    CDN_latch q_reg (.d(d), .ena(g),.aclr(rst), .apre(1'b0), .q(q));
endmodule

`ifdef SYNTHESIS
`else
module CDN_latch (ena, d, aclr, apre, q);
...
endmodule
`endif
```

Using the `async_set_reset` pragma tells RTL Compiler to use the `apre` and `acrl` logic inside of the generic latch to implement the async set and reset logic. If you do not set this pragma, RTL Compiler uses glue logic outside of the generic latch to implement the async set and reset logic.

CDN_mux

With the following RTL code, as shown in Example 10-7, RTL Compiler produces a netlist, such as shown in Example 10-8.

Example 10-7 RTL Code Inferring 2-to-1 Mux

```
module test (y, a, b, s); // 2-to-1 mux
    input s;
    input [2:0] a, b;
    output [2:0] y;
    assign y = s ? b : a;
endmodule
```

Example 10-8 Generic Netlist From Example 10-7

```
module mux (ctl, in_0, in_1, z);
    input [1:0] ctl;
    input [2:0] in_0, in_1;
    output [2:0] z;
    wire [1:0] ctl;
    wire [2:0] in_0, in_1;
    wire [2:0] z;
    CDN_mux2 g1 (.sel0(ctl[1]), .data0(in_0[2]),.sel1(ctl[0]), .data1(in_1[2]),
                 .z(z[2]));
    CDN_mux2 g4 (.sel0(ctl[1]), .data0(in_0[1]),.sel1(ctl[0]), .data1(in_1[1]),
                 .z(z[1]));
    CDN_mux2 g5 (.sel0(ctl[1]), .data0(in_0[0]),.sel1(ctl[0]), .data1(in_1[0]),
                 .z(z[0]));
endmodule

module test (y, a, b, s);
    input s;
    input [2:0] a, b;
    output [2:0] y;
    wire [2:0] a, b;
    wire [2:0] y;
    wire s, s_inv;
    not g2 (s_inv, s);
    mux m1 (.ctl({s_inv, s}), .in_0(a), .in_1(b), .z(y));
endmodule

`ifdef SYNTHESIS
`else
module CDN_mux2 (sel0, data0, sel1, data1, z);
...
endmodule
`endif
```

With the following RTL code, as shown in Example 10-9, RTL Compiler produces a netlist, such as shown in Example 10-10.

Example 10-9 RTL Code Inferring 5-to-1 Mux

```
module test (y, a, b, c, d, e, s); // 5-to-1 mux
    input [2:0] s;
    input [2:0] a, b, c, d, e;
    output [2:0] y;
    reg [2:0] y;
    always @(a or b or c or d or e or s)
        case (s) // cadence full_case parallel_case
                3'b000:  y = a;
                3'b001:  y = b;
                3'b010:  y = c;
                3'b011:  y = d;
                3'b100:  y = e;
                default: y = 5'bx;
    endcase
endmodule
```

Example 10-10 Generic Netlist From Example 10-9

```
module mux (ctl, in_0, in_1, in_2, in_3, in_4, z);
    input [4:0] ctl;
    input [2:0] in_0, in_1, in_2, in_3, in_4;
    output [2:0] z;
    wire [4:0] ctl;
    wire [2:0] in_0, in_1, in_2, in_3, in_4;
    wire [2:0] z;
CDN_mux5 g1 (.sel0(ctl[4]), .data0(in_0[2]),.sel1(ctl[3]), .data1(in_1[2]),
             .sel2(ctl[2]), .data2(in_2[2]),.sel3(ctl[1]), .data3(in_3[2]),
             .data3(in_3[2]), .sel4(ctl[0]), .data4(in_4[2]), .z(z[2]));
CDN_mux5 g4 (.sel0(ctl[4]), .data0(in_0[1]),.sel1(ctl[3]), .data1(in_1[1]),
             .sel2(ctl[2]), .data2(in_2[1]), .sel3(ctl[1]), .data3(in_3[1]),
             .sel4(ctl[0]), .data4(in_4[1]), .z(z[1]));
CDN_mux5 g5 (.sel0(ctl[4]), .data0(in_0[0]),.sel1(ctl[3]), .data1(in_1[0]),
             .sel2(ctl[2]), .data2(in_2[0]),.sel3(ctl[1]), .data3(in_3[0]),
             .sel4(ctl[0]), .data4(in_4[0]), .z(z[0]));
endmodule

module test (y, a, b, c, d, e, s);
input [2:0] a, b, c, d, e;
input [2:0] s;
output [2:0] y;
wire [2:0] a, b, c, d, e;
wire [2:0] s, s_inv;
wire [2:0] y;
wire m000, m001, m010, m011, m100;
wire s000, s001, s010, s011, s100;
not v2 (s_inv[2], s[2]);
not v1 (s_inv[1], s[1]);
not v0 (s_inv[0], s[0];
mux m1 (.ctl({s000, s001, s010, s011, s100}), .in_0(a),.in_1(b), .in_2(c),
        .in_3(d), .in_4(e), .z(y));
nand m0 (m000, s_inv[2], s_inv[1], s_inv[0]);
nand m1 (m001, s_inv[2], s_inv[1], s[0]    );
nand m2 (m010, s_inv[2], s[1],     s_inv[0]);
nand m3 (m011, s_inv[2], s[1],     s[0]    );
nand m4 (m100, s[2],     s_inv[1], s_inv[0]);
```

Example 10-10 *continued*

```
not i0 (s000, m000);
not i1 (s001, m001);
not i2 (s010, m010);
not i3 (s011, m011);
not i4 (s100, m100);
endmodule

`ifdef SYNTHESIS
`else
module CDN_mux5 (sel0, data0, sel1, data1, ... );
...
endmodule
`endif
```

Reading the Netlist

This section applies to both the `read_hdl` command and the `read_hdl -structural` command.

As described in Chapter 6.2 of IEEE Std 1364.1-2002, RTL Compiler, by default, has a macro named `SYNTHESIS` defined. Therefore, RTL Compiler does not see the description of the generic gates and RTL Compiler does not re-synthesize generic gates found in the design description, if any.

However, if the input HDL code defines any of these module/entity names - `CDN_flop`, `CDN_latch`, `CDN_mux*`, or `CDN_dc` - your definition takes precedence. With any of these special names:

- If your definition cannot be found in the input HDL code, RTL Compiler uses the built-in generic definition.
- If your definition is found in the input HDL code, RTL Compiler does not try to identify whether it is the same as (or equivalent to) what the `write_hdl -g` command writes out; RTL Compiler synthesizes your description.

In a bottom-up structural flow, the following scenario can happen. At an early stage of netlist loading, RTL Compiler cannot resolve a `CDN_flop`, `CDN_latch`, `CDN_mux*`, `CDN_dc` instantiation. Therefore, RTL Compiler uses the built-in generic definition for it. At a later stage of netlist loading, RTL Compiler finds the description in another netlist file and uses it for the previous instance that has been *linked* to the built-in generic definition. In other words, the previous decision to use the built-in generic definition for that instance of `CDN_flop`,

`CDN_latch`, `CDN_mux*`, `CDN_dc` is overridden. Your definition takes precedence, even if it comes from a different netlist file.

11

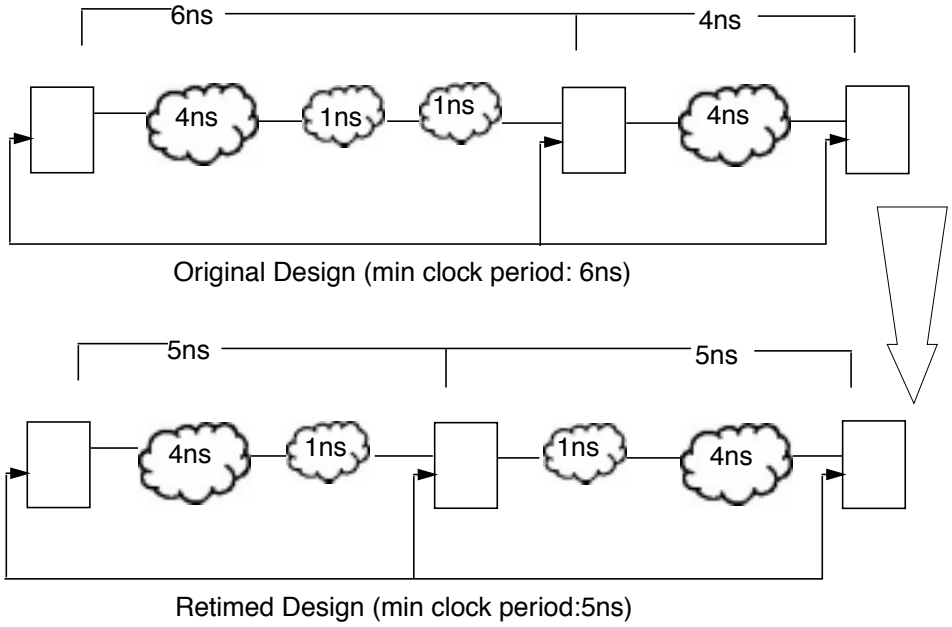
Retiming the Design

Overview

Retiming is a technique for improving the performance of sequential circuits by repositioning registers to reduce the cycle time or the area without changing the input-output latency. This technique is generally used in datapath designs. Pipelining is a subset of retiming where sufficient stages of registers are added to the design. The retiming operation distributes the sequential elements at the appropriate locations to meet performance requirements. Thus, retiming allows you to improve the performance of the design during synthesis without having to redesign the RTL. Retiming does not change or optimize the existing combinational logic.

Figure 11-1 shows how to use retiming to reduce the clock period from 6ns to 5ns.

Figure 11-1 Retiming for Minimum Delay



RTL Compiler supports both automatic and manual retiming.

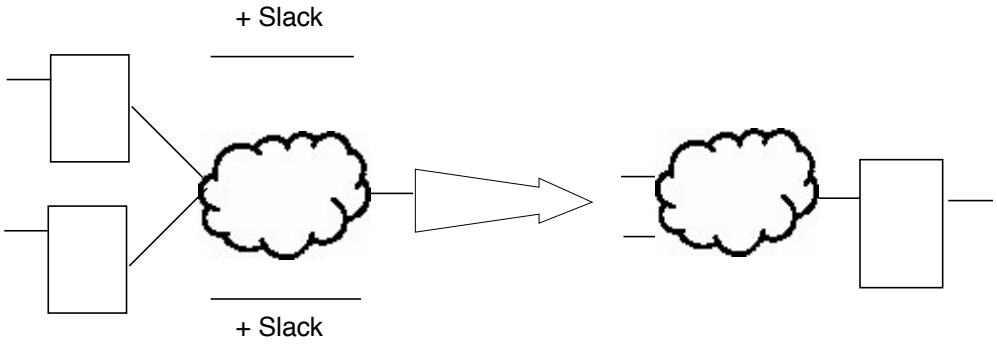
Retiming for Timing

Improving the clock period or timing slack is the most common use of retiming. This can be a simple pipelined design, which contains the combinational logic describing the functionality, followed by the number of pipeline registers that satisfy the latency requirement. It can also be a sequential design that is not meeting the required timing. RTL Compiler distributes the registers within the design to provide the minimum cycle time. The number of registers in the design before retiming may not be the same after retiming because some of the registers may have been combined or replicated.

Retiming for Area

Retiming does not optimize combinational logic and hence the combinational area remains the same. When retiming for area, RTL Compiler moves registers in order to minimize the register count without worsening the critical path in the design. A simple scenario on how registers can be reduced is shown in Figure 11-2.

Figure 11-2 Retiming for Area



Tasks

Retiming Using the Automatic Top-Down Retiming Flow

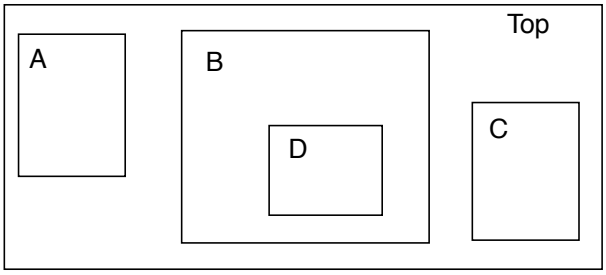
In the top-down (implicit) retiming flow, RTL Compiler retimes those blocks that were marked with the `retime` attribute. In this flow, retiming focuses on minimizing the delay. To retime the design to minimize area, you must use the manual retiming flow. See Manual Retiming (Block Level Retiming) on page 198 for more information about manual retiming.

If the `retime` attribute is set on a top level design, all the subdesigns will also be retimed. Use this flow when retiming is part of a well-planned synthesis strategy and the design has retimeable subdesigns. Set the `retime` attribute to `true` on the desired modules after elaboration and then synthesize the entire design with `synthesize -to_mapped`. No special command is needed: RTL Compiler automatically derives appropriate constraints, synthesizes, and retimes the specified modules.

When synthesizing, you must synthesize to a technology mapped netlist. That is, you must specify the `synthesize -to_mapped` command. Using `synthesize -incremental` or `synthesize -to_generic` for example, will not result in retimed modules despite the `retime` attribute specification.

Figure 11-3 depicts a small, hierarchical design with three levels of hierarchy. The top level module is called `Top`. Submodules `A`, `B`, and `C` represent the next level down while the `D` submodule represents the last level. Thus, subdesign `B` contains subdesign `D` and some glue logic.

Figure 11-3 Graphic Illustration of a Hierarchical Design



Example 11-1 Top-Down Retiming on Submodules

The following example illustrates how to retime only the `A` and `D` modules referred to in Figure 11-3:

1. Read the HDL for the entire design using the following command:

   ```
   rc:/> read_hdl Top.v
   ```

2. Elaborate the top level design using the following command:

   ```
   rc:/> elaborate Top
   ```

3. Set the `retime` retiming attribute on the subdesigns you want to retime. In this example, this would be `A` and `D`:

   ```
   rc:/> set_attribute retime true /designs/Top/subdesigns/A
   rc:/> set_attribute retime true /designs/Top/subdesigns/B/subdesign/D
   ```

   Note: This step enables automatic retiming. The specified modules will now automatically be retimed during synthesis.

4. Apply top-level design constraints in SDC or by using the RTL Compiler native format and optimization settings. In the following step, SDC is used:

   ```
   rc:/> read_sdc top.sdc
   rc:/> include top.scr
   ```

   The `top.scr` file contains the optimization settings. There is no need to specify any special or “massaged” constraints in this top-down automatic flow.

5. Synthesize the design top-down using the following command:

   ```
   rc:/> synthesize –to_mapped
   ```

   During this step the design is optimized, including technology independent RTL optimization, advanced datapath synthesis, global focus mapping, and incremental optimization. During the mapping phase, retiming is performed automatically on the blocks marked with the `retime` attribute and focuses on minimizing the delay.

   Note: For retiming in this automatic top-down flow, you must use the

`synthesize -to_mapped` command to realize retiming optimization in the specified modules.

6. Evaluate the results using the following commands:

```
rc:/> report timing
rc:/> report gates
rc:/> report area
```

If you wanted to retime `Top` and all its subdesigns, not just `A` and `D`, merely set the `retime` attribute on `Top`:

```
rc:/> set_attribute retime true /designs/Top/
```

Manual Retiming (Block Level Retiming)

Use the manual retiming method when you want to retime specific sub-blocks in your design. Manual retiming does not involve a flow, like automatic top-down retiming. Instead, your specific retiming scenarios dictate which and when retiming commands and attributes are used.

Synthesizing for Retiming

Designs intended for retiming should be synthesized with realistic constraints to account for any pipeline stages.

Synthesize for retiming by either using the `path_adjust` command before synthesis or synthesize the design automatically while deriving realistic constraints using the `-prepare` option of the `retime` command:

```
rc:/> retime -prepare
```

As the option named implies, `retime -prepare` “prepares” the design for retiming by deriving the appropriate constraints and synthesizing to a gate-level design that is ready for retiming. When retiming is subsequently performed, the original constraints will be used and not those derived with the `-prepare` option.

Note: If you are retiming to minimize area with the `-min_area` option, do not use the `-prepare` option at all. See Retiming for Minimum Area on page 199 for more information.

Retiming for Minimum Delay

Perform block level retiming on a block by block basis to further optimize the design, thereby minimizing the delay or area. This is performed on a gate-level design that has been synthesized.

Pipelined designs should first be synthesized with their pipeline constraints. Otherwise, synthesis will produce a design with a larger area due to over constraining. This expanded area cannot be minimized even with subsequent synthesis optimizations.

When you are retiming to optimize for timing on only one design or subdesign, you can use the `-prepare` and `-min_delay` options together:

```
rc:/> retime -prepare –min_delay
```

Alternatively, you can issue `retime -prepare` before `retime -min_dealy` sequentially:

```
rc:/> retime -prepare
rc:/> retime -min_delay
```

If you are specifying multiple subdesigns, then issuing the commands separately will first map all subdesigns and then retime them. If you specify the options together on multiple subdesigns, then each subdesign will be mapped and retimed before the next subdesign is processed.

Retiming for Minimum Area

Retiming can recover sequential area from a design with both easy to meet timing goals and a positive slack from the initial synthesis. Retiming a design that does not meet timing goals after the initial synthesis could impact total negative slack: the paths with the better slack can be “slowed down” to the range of worst negative slack.

- Use retiming to try to recover area with the following command:

  ```
  rc:/> retime –min_area
  ```

Do not use the `-prepare` option at all if you are retiming to minimize area.

The following two examples illustrate scripts that perform block level manual retiming. Example 11-2 does not use the `retime -prepare` command while Example 11-3 does. Example 11-3 does not require the removal of any path adjust or multi-cycle constraints.

Example 11-2 Retiming without Using the retime -prepare Command

```
read_hdl <block.v>
elaborate
include design.constraints //clock period should have been massaged to account
                           //for the pipeline stages – path adjust, and so on.
synthesize –to_mapped
rm massaged_clock_constraints //Remove any clock constraints that were massaged
                              //for retiming purposes
rm /designs/block/timing/exceptions/path_adjusts/*
rm /designs/block/timing/exceptions/multi_cycles/*
```

```
report timing
retime –min_delay  | -min_area
report timing
report gates
synthesize –to_mapped –incremental
..
```

Example 11-3 Retiming Using the retime -prepare Command

```
read_hdl <block.v>
elaborate
include design.constraints
retime –prepare
report timing
retime –min_delay
report timing
report gates
synthesize –to_mapped -incremental
..
```

As both of the above examples illustrate, it is best to specify the `-incremental` option with the `synthesize -to_mapped` command because doing so will generally yield realize faster run-times.

Incorporating Design for Test (DFT) and Low Power Features

There are two flows that involve retiming a design with DFT and low power features. One is the recommended flow, while the other is available if the recommended flow cannot be pursued.

The recommended flow involves setting the retiming, DFT, and low power attributes before sythesizing the design to gates. The following example illustrates this flow.

Example 11-4 Recommended Flow for Retiming with DFT and Low Power

```
set_attribute lp_insert_clock_gating true /
set_attribute lp_insert_operand_isolation true /

read_hdl teagan.v
elaborate

set_attribute lp_clock_gating_max_flops 18 /designs/*
set_attribute lp_clock_gating_min_flops 6 /designs/*
```

```
set_attribute lp_clock_gating_test_signal test_signal /designs/top_design
set_attribute max_leakage_power number /designs/top_design
define_dft test_mode test_mode_signal
define_dft shift_enable shift_enable_signal
check_dft_rules

set_attribute retime true [design | subdesign]
synthesize -to_mapped

report timing
report clock_gating
report dft_registers

connect_scan_chains -auto
report dft_chains
synthesize -to_mapped -incremental
```

Note: If you have multiple clock-gating cells for the same load-enable signal (for example, you are limiting the fanout of a clock-gating cell), retiming will put all the flops driven by the same clock-gating cell in a separate, single class. Flops with different classes would not be merged.

- See *Design for Test in Encounter RTL Compiler* for more information on DFT.
- See *Low Power in Encounter RTL Compiler* for more information on low power.

The following example illustrates an alternative flow that involves retiming the design after it has been mapped to gates: the clock-gating logic has been inserted and the scan flops have been mapped. In this flow, the `retime` command is explicitly issued (indicating manual retiming) whereas in the recommended flow only the `retime` attribute was specified (indicating automatic, top-down retiming).

Example 11-5 Alternative Flow for Retiming with DFT and Low Power

```
set_attribute lp_insert_clock_gating true /
set_attribute lp_insert_operand_isolation true /

read_hdl teagan.v
elaborate

set_attribute lp_clock_gating_max_flops 18
set_attribute lp_clock_gating_min_flops 6
set_attribute dft_scan_style {muxed_scan|clocked_lssd_scan} /
```

```
define_dft test_mode test_mode_signal
define_dft shift_enable shift_enable_signal
check_dft_rules

synthesize -to_mapped                    //Synthesizes the netlist that has
                                          //the scan flops and clock gating logic
set_attribute unmap_scan_flops true /

retime -min_delay
report timing

replace_scan
connect_scan_chain -auto_create
report dft_chains
report clock_gating
report dft_setup

synthesize -to_mapped -incremental
report timing
```

- You do not have to issue the `retime -prepare` command in this flow. An exception would be if the design contains pipelining and the original constraints are not well adjusted for retiming. In such a case, issuing `retime -prepare` before `retime -min_delay` could help achieve better area and timing.
- The scan flops must be unmapped after synthesizing to gates. Otherwise, all the scan flops will not to be retimed. Furthermore, since the scan flops must be unmapped before retiming, the scan chains will become unconnected. As the example above illustrates, the scan chains must be restitched with the `connect_scan_chain` command.
- The `replace_scan` command needs to be used in this flow because scan flops are replaced with simple flops during retiming. Consequently, the original flops could be replaced with bigger scan flops. As the example above illustrates, it is recommended that you perform an incremental optimization to resize such flops for timing.

See *Design for Test in Encounter RTL Compiler* for more information on DFT.

See *Low Power in Encounter RTL Compiler* for more information on low power.

Localizing Retiming Optimizations to Particular Subdesigns

Use the `retime_hard_region` attribute to contain the retiming operations to a specific subdesign. By default, RTL Compiler operates on all retimeable logic through all levels of

hierarchy. Therefore, if multiple subdesigns on the same level of hierarchy are being retimed, their interfaces may get modified. Setting the `retime_hard_region` attribute on these subdesigns will localize the retiming operations to the submodule boundaries. However, doing so will have a negative impact on QoS.

The following example prevents all the registers in the `SUB_1` subdesign from being moved across its boundary:

```
rc:/> set_attribute retime_hard_region true [find / -subdesign SUB_1]
    Setting attribute of subdesign SUB_1: 'retime_hard_region' = true
```

Controlling Retiming Optimization

Use the `dont_retime` attribute to control which sequential instances can be moved around and which should not be moved. For example:

```
rc:/designs/retime_eg/instances_seq/U1> set_attribute dont_retime true a_reg1
rc:/designs/retime_eg/instances_seq/U2> set_attribute dont_retime true B_reg
```

Note: Set the `dont_retime` attribute before using the `retime` command.

An object specified with the `dont_retime` attribute is treated as a boundary for moving flops, thus, flops cannot move over it. Although retiming is only available on sequential instances, RTL Compiler does not consider the following objects for retiming:

- Asynchronous registers with *both* set and reset signals (but does consider registers with either a set or reset signal)
- Latches
- Preserved modules
- RAMs
- Three-state buffers
- Unresolved references

All sequential registers that are part of the following timing exceptions are treated as implicit `dont_retime` objects:

- false path
- multicycle path
- path adjust
- path delay

- preserved sequential cells (sequential cells marked with the `preserve` attribute)

Note: During retiming, registers which belong to a path_group will be removed from the `path_group`. After retiming, the original path_group constraints will have to be re-applied if they are needed for static timing analysis or optimization purposes.

Retiming Registers with Asynchronous Set and Reset Signals

Setting the `retime_async_reset` attribute to `true` will retime those registers that have either a set or reset signal. Registers that have both set and reset signals will not be retimed in any case.

Optimize registers with reset signals with the `retime_optimize_reset` attribute. The attribute will replace those registers whose set or reset conditions evaluate to `dont_care` with simple flops without set or reset inputs. This attribute needs to be set in addition to the `retime_async_reset` attribute.

Figures 11-4 through 11-6 below illustrate the `my_flop` register experience retiming as well as retiming with asynchronous reset optimization.

Figure 11-4 Register with Asynchronous Reset

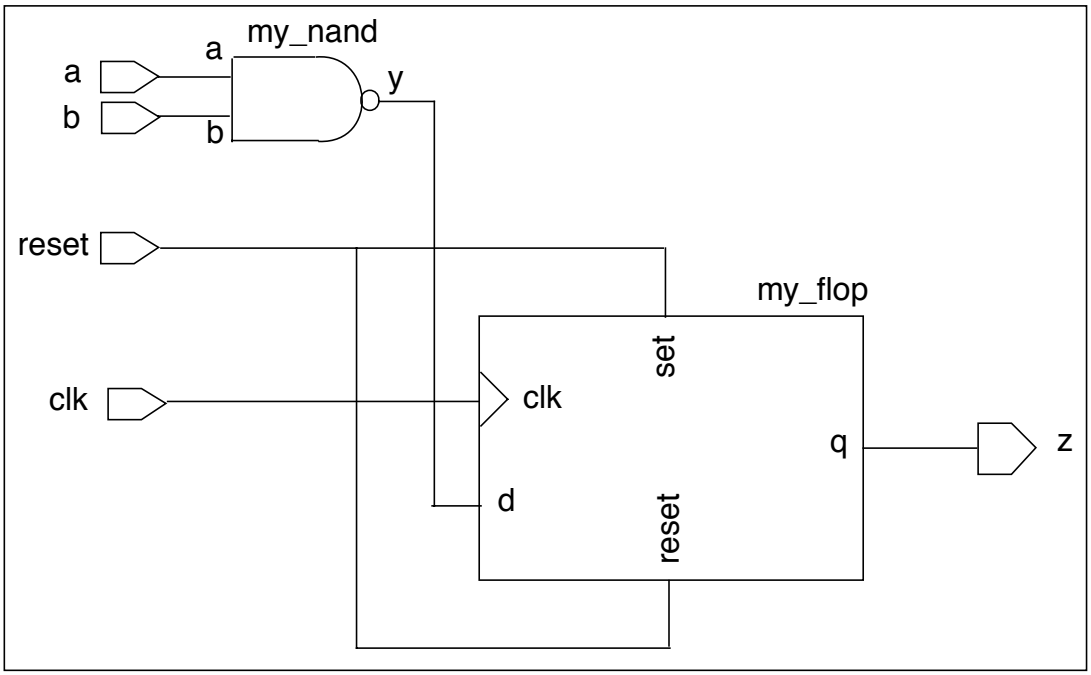


Figure 11-5 Register with Asynchronous Reset after Retiming

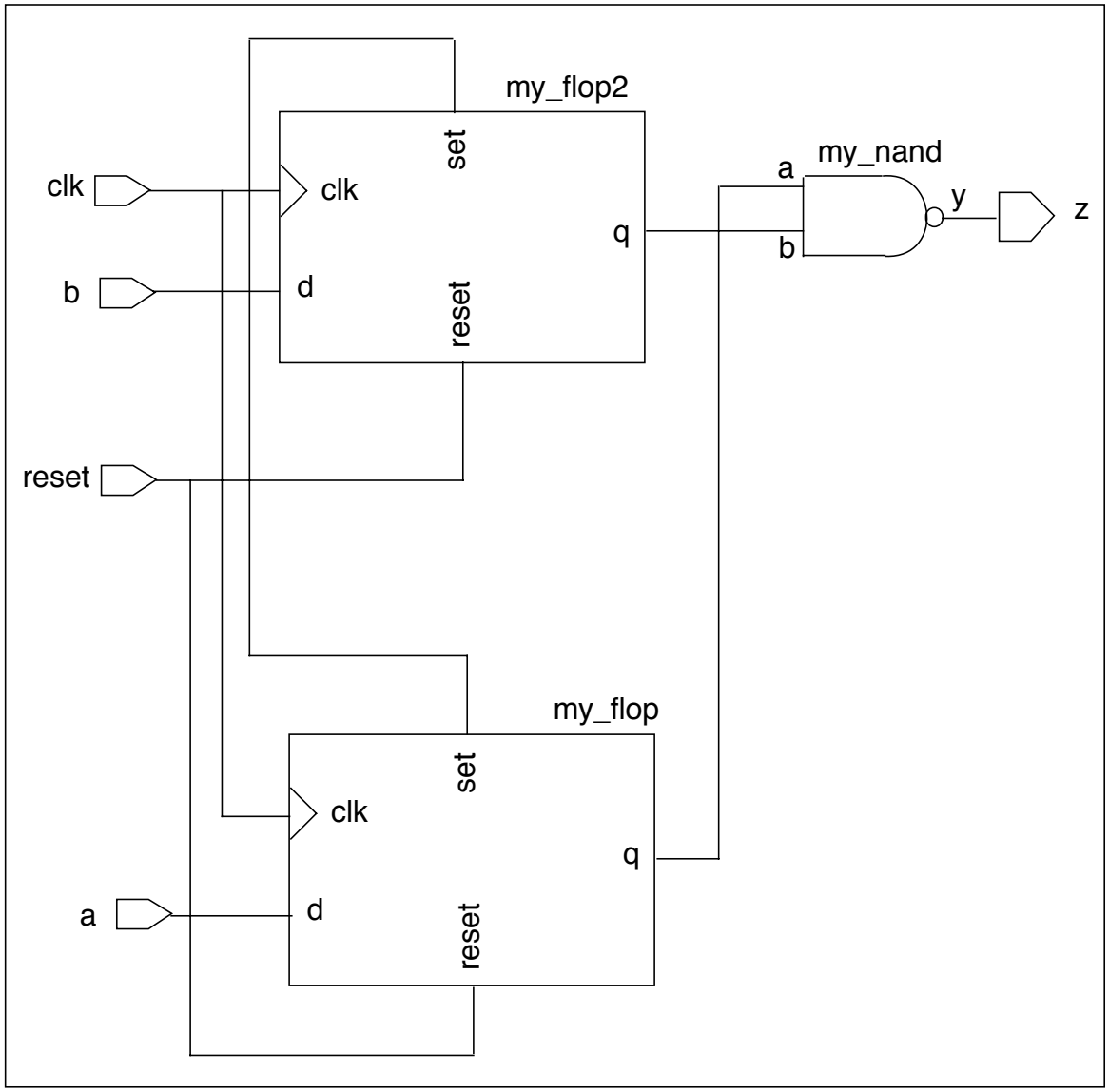
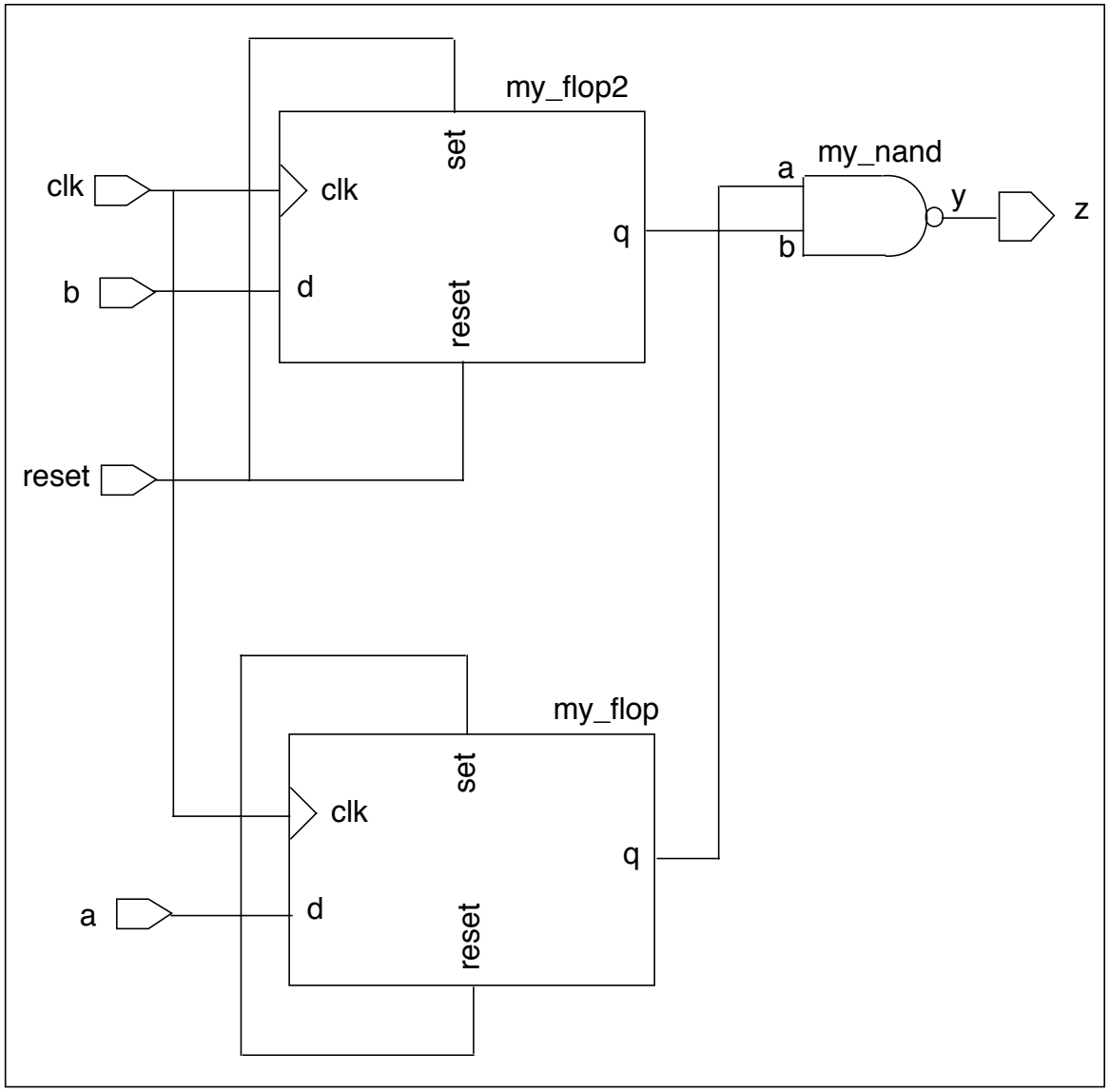


Figure 11-6 Register with Asynchronous Reset after Retiming and Optimization



The `retime_async_reset` and `retime_optimize_reset` attributes are root attributes and they should be set before issuing the `retime` command:

Note: Using the `retime_async_reset` attribute can cause longer run-times.

Example 11-6 Retiming Asynchronous Registers with Set and Reset Signals

```
...
rc:/> set_attribute retime_async_reset true /
rc:/> set_attribute retime_optimize_reset true /
rc:/> retime -prepare
```

```
rc:/> retime -min_delay
...
```

By default, registers with either set or reset or both assume the `dont_retime` attribute and consequently they will not be retimed. If retiming is initially performed without enabling the `retime_async_reset` attribute, such registers cannot be retimed later unless the `dont_retime` is removed. Therefore, enable the `retime_async_reset` attribute before the initial retiming.

Note: Enabling the `retime_async_reset` attribute could impact run-time because the tool needs to ensure that the initial condition of the set and reset is preserved. Registers with both asynchronous set and reset signals will not be retimed in any case.

Identifying Retimed Logic

You can identify which registers were moved due to retiming optimization with the `retime_reg_naming_suffix` attribute. The attribute allows you to specify a particular suffix to the affected registers. By default, the `_reg` suffix is appended. You must specify this attribute before you retime the design.

The following example instructs RTL Compiler to add the `__retimed_reg` suffix to all registers that are moved during retiming optimization:

```
rc:/> set_attribute retime_reg_naming_suffix __retimed_reg
    Setting attribute of root /: 'retime_reg_naming_suffix' = __retimed_reg
```

The affected registers could look like the following example:

```
D_F_LPH0002_H retime_16__retimed_reg(.E (ck), .D (n_118), .L2N (n_159));
D_F_LPH0001_E retime_17__retimed_reg((.E (ck), .D (n_118), .L2 (n_158));
D_F_LPH0002_E retime_8__retimed_reg((.E (ck), .D (n_112), .L2N (n_165));
```

RTL Compiler also allows you to retrieve the original names of the retimed registers through the `trace_retime` and `retime_original_registers` attributes. Mark the registers you want to track with the `trace_retime` attribute and use the `retime_original_registers` attribute to return the original names of those registers that were marked with the `trace_retime` attribute.

The following example specifies that all retimed registers have a `_stormy_reg` suffix. It then marks all registers so that they can be retrieved. After retiming, we see that the original name of the `retime_1_stormy_reg` register is `teagan1_reg[7]`.

Example 11-7 Retrieving the Original Name of a Retimed Register

```
rc:/> set_attribute retime_reg_naming_suffix _stormy_reg
rc:/> set_attribute trace_retime true [find / -instance teagan1_reg[7]]
```

```
...
rc:/> retime -prepare
rc:/> retime -min_delay
rc:/> get_attribute retime_original_registers retime_1_stormy_reg

teagan1_reg[7]
```

Retiming Multiple Clock Designs

RTL Compiler retimes only one clock domain at a time. If your design has multiple clocks, you must:

1. Set the `dont_retime` attribute to `true` on all the sequential instances for all clock domains except for the current one on which you wish to work.
2. Retime the design.
3. Set the `dont_retime` attribute to `true` on the retimed domain and `false` on the new domain to be retimed.

Repeat these steps until all desired clock domains are retimed. The following example illustrates these steps on a design with two clock domains, `clk1` and `clk2`.

Example 11-8 Retiming a Design with Two Clock Domains

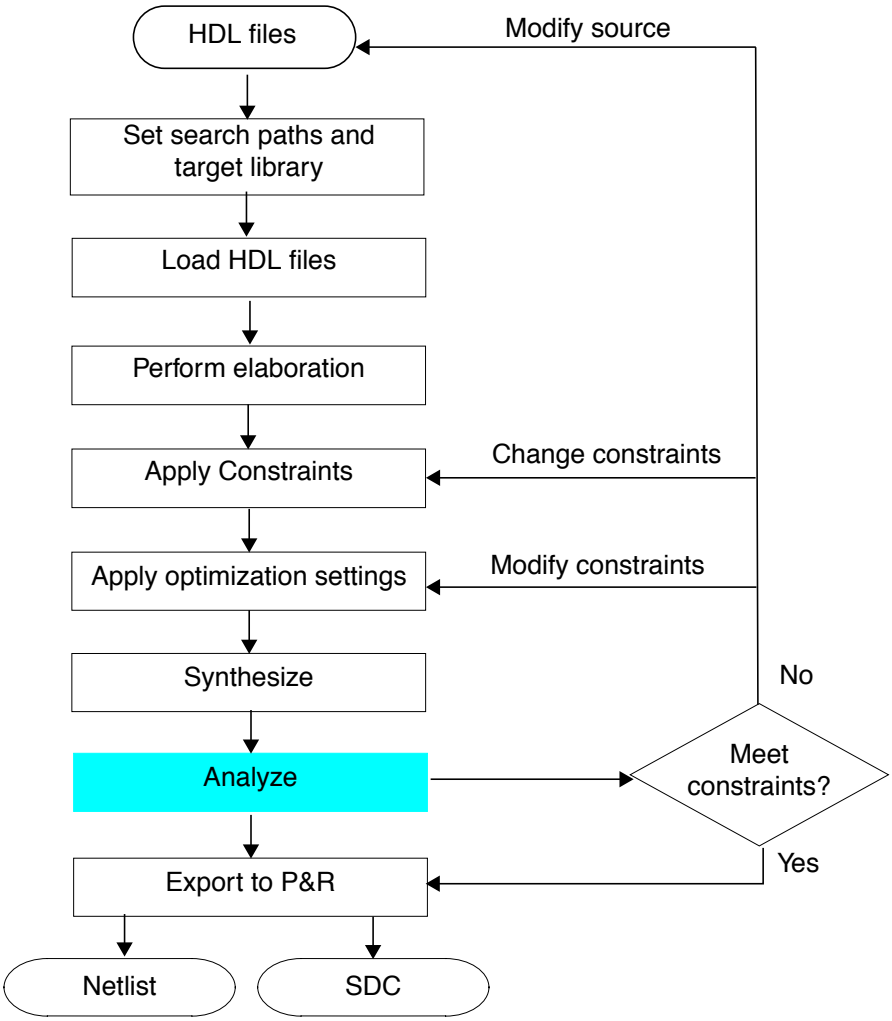
```
rc:/> read_hdl test2clk.v
rc:/> elaborate

specify_multiclock_constraints

rc:/> set_attribute dont_retime true [all::all_seqs -clock clk2]
rc:/> retime -prepare    //Optional
rc:/> retime -min_delay
rc:/> set_attribute dont_retime false [all::all_seqs -clock clk2]
rc:/> set_attribute dont_retime true [all::all_seqs -clock clk1]
rc:/> retime -prepare    //Optional
rc:/> retime -min_delay
```

In the above example, after issuing the first `retime -min_delay`, all the logic clocked by `clk1` will be retimed. The `dont_retime` attribute is set to `true` on the `clk1` domain before issuing the `retime` command again. Otherwise, the `clk1` domain would get retimed again while the `clk2` domain would remain untimed. The second `retime -min_delay` command will now retime the `clk2` domain.

12

Generating Reports

Overview



This chapter discusses how to analyze your synthesis results using the report command and the log file.

Tasks

Generating Timing Reports

Use the `report timing` command to generate reports on the timing of the current design. The default timing report generates the detailed view of the most critical path in the current design.

➤ To generate a timing report, `cd` into the design directory and type the following command:

```
rc:/designs/top> report timing
```

The timing report provides the following information:

- Type of cell (`flop-flop`, `or`, `nor`, and so on)
- The cell's fanout and timing characteristics (load, slew, and total cell delay)
- Arrival time for each point on the most critical path

Use the `-lint` option to generate timing reports at different stages of synthesis. This option provides a list of possible timing problems due to over constraining the design or incomplete timing constraints, such as not defining all multicycle or false paths.

```
rc:/designs/top> report timing -lint
```

Use the `-from` and `-to` options to report the timing value between two points in the design. The timing points in the report is given with the `<<<` indicator.

```
rc:/designs/top> report timing -from [find / -instance cout_reg_3] -to flag5
```

The following timing report is an example output of the above command:

```
...
I1/clock
  cout_reg_3/CK       <<<                              0              0 R
  cout_reg_3/Q              DFFRHQX1      3    24.8    646    +518     518 R
I1/cout[3]
p0160A/B                                                    +0       518
p0160A/Y                    NOR2X1        1    7.4     262    +174     692 F
p0201A/B                                                    +0       692
p0201A/Y                    NAND3BX1      1    8.0     285    +174     866 R
p0257A/B                                                    +0       866
p0257A/Y                    NOR4X1        1    3.6     185    +133     999 F
top_counter/flag5    <<<    out port                      +0       999 F
...
```

Use the `-exceptions` or `-cost_group` options to generate the timing reports for any of the previously set timing exception names or the set of path group names defined by the `define_cost_group` command. These help generate custom timing reports for the paths that you previously assigned to cost groups.

```
rc:/designs/top> report timing -exceptions <exception_name>
```

or

```
rc:/designs/top> report timing -cost_group <cost_group_name>
```

If timing is not met, “Timing Slack” is reported with a minus (-) number and “TIMING VIOLATION” is written out.

RTL Compiler generates the timing accuracy report down to the gate and net level.

The following is an example timing report run with the `-num_paths` option:

```
rc:/> report timing -num_paths 4
============================================================
  Generated by:        Cadence RTL Compiler (RC) 2005.Q3.0
  ...
============================================================

path   1:

    Pin               Type      Fanout  Load  Slew   Delay   Arrival
                                        (fF)  (ps)    (ps)    (ps)
-------------------------------------------------------------------------
...
```

```
-------------------------------------------------------------------------
Timing slack :     543ps
Start-point  : accum[1]
End-point    : aluout_reg_7/D

path   2:

    Pin               Type      Fanout  Load  Slew   Delay   Arrival
                                        (fF)  (ps)    (ps)    (ps)
-------------------------------------------------------------------------
...
-------------------------------------------------------------------------
Timing slack :     547ps
Start-point  : accum[1]
End-point    : aluout_reg_6/D

path   3:

    Pin               Type      Fanout  Load  Slew   Delay   Arrival
                                        (fF)  (ps)    (ps)    (ps)
-------------------------------------------------------------------------
...
-------------------------------------------------------------------------
Timing slack :    1030ps
Start-point  : accum[1]
End-point    : aluout_reg_5/D

path   4:

    Pin               Type      Fanout  Load  Slew   Delay   Arrival
                                        (fF)  (ps)    (ps)    (ps)
-------------------------------------------------------------------------
...
-------------------------------------------------------------------------
Timing slack :    1034ps
Start-point  : accum[1]
End-point    : aluout_reg_4/D
```

Generating Area Reports

The area report gives a summary of the area of each component in the current design. The report gives the number of gates and the area size based on the specified technology library. Levels of hierarchy are indented in the report.

In the outputs generated by `report gates` and `report area` commands, RTL Compiler shows the technology library name, operating conditions, and the wire-load mode used to generate these reports.

➤ To generate an area report, type the following:

```
rc:/> report area
```

RTL Compiler generates a report similar to the example below.

```
=====================================================
  Generated by:          RTL Compiler  (RC) 2005.1.2
  Generated on:          Jan 28 2005 10:20:27 AM
  Module:                top_counter
  Technology library:    slow 1.0
  Operating conditions:  slow
  Wireload mode:         segmented
=====================================================

Block        Cells   Cell Area   Net Area     Wireload
-----------------------------------------------------------
top_counter     56        1873          0   CDE18_Conservative (D)
  I2            24         880          0   CDE18_Conservative (D)
  I1            24         863          0   CDE18_Conservative (D)
 (D) = wireload is default in technology library
```

➤ To generate a report that shows a profile of all library cells inferred during synthesis, type the following command:

```
rc:/> report gates
```

RTL Compiler generates a report listing all the gates, the number of instances in the design, and the total area for all these instances.

```
=====================================================
  Generated by:          RTL Compiler  (RC) 2005.1.2
  Generated on:          Jan 28 2003  10:20:33 AM
  Module:                top_counter
  Technology library:    slow 1.0
  Operating conditions:  slow
  Wireload mode:         segmented
=====================================================
   Gate   Instances   Area  Library
-----------------------------------
AND2X2           10   166.3    slow
AOI21X1           2    33.3    slow
AOI2BB2X1         2    46.6    slow
DFFRHQX1         13   910.7    slow
DFFRHQX2          3   260.1    slow
INVX1             2    20.0    slow
```

```
INVX3              2    20.0    slow
NAND2X1            3    29.9    slow
NAND3BX1           2    39.9    slow
NAND4BX1           1    23.3    slow
NOR2X1             4    39.9    slow
NOR3X1             1    16.6    slow
NOR4X1             1    20.0    slow
OAI2BB2X1          8   186.3    slow
XNOR2X1            2    59.9    slow
-----------------------------------
total             56  1872.7


   Type    Instances   Area  Area %
-----------------------------------
sequential         16  1170.8   62.5
inverter            4    39.9    2.1
logic              36   662.0   35.3
-----------------------------------
total              56  1872.7  100.0
```

At the end of the gate report, RTL Compiler shows the total number of instances and the area for all the sequential cells, inverters, buffers, logic, and timing-models, if any.

To get a report on the total combinational area, add the logic, inverter, and buffer area numbers.

Summarizing Messages

Use the `report messages` command to summarize all the info, warning, and error messages that were issued by RTL Compiler in a particular session. The report contains the number of times the message has been issued, the severity of the message, the ID, and the message text.

The `report messages` command has various options that can selectively print message types or print all the messages that have been issued in a particular session. Typing the `report messages` command without any options prints all the error messages that have been issued *since the last time report messages was used*. Therefore, if no messages were issued since the last time `report messages` was used, RTL Compiler returns nothing. Consult the *RTL Compiler Command Reference* for more information on the `report messages` command.

The following example is the first request to `report messages` in a session:

```
rc:/> report messages
================ Message Summary ================
Num     Sev     Id          Message Text
-------------------------------------------------------
1       Info    ELAB-VLOG-9 Variable has no fanout. This variable is not driving
anything and will be simplified
3       Info    LBR-30      Promoting a setup arc to recovery. Setup arcs to
asynchronous input pins are not supported
3       Info    LBR-31      Promoting a hold arc to removal. Hold arcs to
asynchronous input pins are not supported
1       Info    LBR-54      Library has missing unit. Current library has missing
unit.
```

If `report messages` were typed again (with no intermediate commands or actions), RTL Compiler would return nothing.

Redirecting Reports

The `report` command sends the output to `stdout` by default You can redirect `stdout` information to a file or variable with the `redirect` command. If you use the `-append` option, the file is opened in append mode instead of overwrite mode.

Example

- To write the `report gates` report to a file called `gates.rep`, type the following command:

  ```
  rc:/> report gates > gates.rep
  ```

  or

  ```
  rc:/> redirect gates.rep “report gates”
  ```

- To append information into the existing `gates.rep` file, type the following command:

  ```
  rc:/> redirect -append gates.rep “report gates”
  ```

- To send the reports to `stdout` and to a file on the disk, type the following command:

  ```
  rc:/> report gates
  rc:/> report gates > gates.rep
  ```

  or

  ```
  rc:/> redirect -tee gates.rep “report gates”
  ```

Customizing the report Command

The `lib/cdn/rc/report.tcl` file contains commands that make it easy to create custom reports. These commands allow you to create a report header and to tabulate data into columns. You can even add your report as a subcommand of RTL Compiler's `report` command.

Analyzing the Log File

Log files contain information recorded during any activity within the tool, including all manually typed commands and all messages printed to `stdout`.

The following topics will be useful for analysis if you encounter an issue and the complete log file cannot be sent.

- Status Messages on page 217
- Reporting Area in the Log File on page 218
- Incremental Optimization on page 218
- Reporting Run Time on page 219
- Generating Target Timing Values on page 220
- Global Map Report on page 221
- Tracking Total Negative Slack on page 221

Status Messages

During certain processes, like optimization, RTL Compiler will print status messages that indicate its activity level or progression. For example, during optimization, you might encounter the following short messages:

```
Pruning unused logic...
Analyzing hierarchical boundaries...
Performing redundancy-removal...
```

These messages correspond to internal events that are occurring and are printed to provide you with a status, not as an aid for debugging. They can be viewed as textual representations of the hourglass that appears when launching GUI based applications: they covey that the tool is actively trying to process something.

Reporting Area in the Log File

The area report found in the log file is identical to the one generated through the report area command. See Generating Area Reports on page 213 for a detailed explanation of area reporting.

Incremental Optimization

Incremental optimization is the process of incrementally optimizing mapped gates. Therefore, it is only available after the `synthesize -to_mapped` command has been issued or the gates of the design have already been mapped from a previous synthesis session. The following information shows the current slack and critical path start points and end points:

```
Incremental optimization status
==============================
                         Group
                         Total
                  Total  Worst
Operation          Area  Slacks Worst Path
-------------------------------------------------------------------------------
 incr_delay    2470126    306  ifu/xidpPCpipe/idpPC1F_reg[60]/cp -->
                               ifu/xidpPCpipe/idpPC1B_reg[24]/d
    C2C (Wt.: 1) (Slack: -223)  ifu/xidpPCpipe/idpPC1F_reg[60]/cp -->
                               ifu/xidpPCpipe/idpPC1B_reg[24]/d
    C2O (Wt.: 1) (Slack: -83)  biu/bdeDBrdg/bdeDSysBusReq1X_reg/cp -->
....
```

This information in the incremental optimization phase shows the different routines that are called, their run time, and so on.

```
      Trick     Calls    Accepts   Attempts          Time
-------------------------------------------------------------
 glob_delay        10  (        0 /       10 )  21430
  crit_upsz      7831  (      788 /     1616 )  47890
  crit_dnsz      2484  (      118 /     1581 )  54230
  load_swap       668  (       64 /      260 )  3440
  crit_swap       557  (       37 /      147 )  2600
        dup       353  (        2 /       37 )  1430
  un_buffer         0  (        0 /        0 )  0
       fopt       423  (       12 /      125 )  2770
   setup_dn         6  (        0 /        6 )  3870
        exp        18  (       14 /       54 )  5200
```

```
Final optimization status
=========================
                         Group
                         Total  - - DRC Totals - -
                  Total  Worst       Max    Max
Operation          Area  Slacks      Cap   Fanout
-------------------------------------------------------------------------------
 incr_drc       2472247    250   7074330   100960
             Path: iu/arf/arfRs0BypsToIU0/arfRs0ReadData1A_reg[1]/cp -->
                 lsu/dpcdPrimCache/dpcdPriCacheMem/mem256x128r1w2/dpccL0WriteS1A
....
```

In addition to the optimization operation, total area, maximum capacitance, and maximum fanout values, a group total worst slack value is reported. This value reports the sum of the worst violations among all cost groups.

Reporting Run Time

If you want to retrieve the run time that includes the first issued command to the end of the last command, query the `runtime` attribute. This feature is available at anytime and does not include the run time query itself. This is a root attribute.

➤ To report RTL Compiler's design process time in CPU seconds, not actual clock time, type the following command:

```
rc:/> get_attribute runtime /
```

The value is printed to `stdout`. To format the output, use the following command:

```
puts "The RUNTIME is [get_attribute runtime /]"
```

➤ To report memory utilization, type the following command:

```
rc:/> get_attribute memory_usage /
```

or

```
rc:/> puts "The MEMORY USAGE is [get_attr memory_usage]" /
```

RTL Compiler will return the memory usage in kilobytes.

Tip

To get reference points throughout the synthesis process, use these commands in your script after elaboration, `synthesize -to_generic`, `synthesize -to_mapped`, and at the end of the session.

Generating Target Timing Values

Target timing values help you determine whether the design goals are realistic. RTL Compiler can generate a target timing number before completing synthesis. This number is based upon the fastest speed that the design can accommodate given the specified clock period.

This number is generated after roughly one third of the total synthesis run time, so you can decide whether or not to let RTL Compiler proceed with synthesis.

To generate this number in the log file (or in `stdout`), set the following variable in your script before loading or elaborating your design:

```
set map_timing 1
```

When this Tcl variable is used, the log file will have the output similar to the example shown below.

```
...
max rise (pin: top_counter/flag5[0])      target   100
Target worst path (pre-mapping):
counter29/cout_reg_0/q (DFFRHQX4)                    arr=  227r  fo=3
I2/cout[0]                                           arr=  278r  fo=0
mux_ctl_0xi/I2_cout[0]                               arr=  278r  fo=1
    NAND4X4/(D->Y) (0xae96de8/f=+n)                  arr=  419f  fo=1
    NOR4X2/(D->Y) (0x931a490/f=-n)                   arr=  644r  fo=0
mux_ctl_0xi/flag5[0]                                 arr=  644r  fo=0
top_counter/flag5[0]                                 arr= 1644r
                                                     req= 2000
...
```

- When the `target` is positive, RTL Compiler can achieve a faster clock speed, which is the specified clock period minus the target number.
- When the `target` is negative, RTL Compiler might produce a violation by this target value by the end of optimization.
- When `target` is a large negative number, you might want to reconsider your constraints for more realistic values.

Along with the target number, RTL Compiler will show the probable critical path. You should verify if this is a valid path in your design.

Note: In some cases, unspecified false paths might show up as the critical path.

Global Map Report

In the global mapping status report, RTL Compiler shows the worst critical path with the corresponding total area and the worst negative slack on different processing stages (`global_map`, `fine_map`, `area_map`). As each step is processed, RTL Compiler tries to meet or improve the timing and then to reduce the area without degrading the worst critical path timing.

```
Global mapping status
=====================
                          Worst
                 Total     Neg
Operation         Area    Slack  Worst Path
-------------------------------------------------------------------------------
 global_map       8143     -139  decode_reg_10/CK --> go_data_reg/D
 fine_map         7238     -181  decode_skip_one_reg/CK --> go_prog_reg/D
 area_map         7245     -111  decode_reg_10/CK --> read_data_reg/D
 area_map         7192     -117  decode_reg_14/CK --> two_cycle_reg/D
 area_map         7212     -111  decode_reg_10/CK --> go_data_reg/D
```

Tracking Total Negative Slack

During the optimization process (`synthesize -to_mapped`) RTL Compiler reports the Worst Negative Slack information in the log files. This information will be listed under the Global mapping status, Local delay optimization status, and Final optimization status sections of the log file.

```
Global mapping status
=====================
                         Worst
                Total     Neg
Operation        Area    Slack  Worst Path
-------------------------------------------------------------------------
 global_map      2764     1403  I2/cout_reg_3/CK --> flag5
...
Incremental optimization status
===============================
                         Worst
                Total     Neg
Operation        Area    Slack  Worst Path
-------------------------------------------------------------------------
 init_delay      2671     1368  I1/cout_reg_0/CK --> flag5
...
```

```
Final optimization status
=========================
                         Worst  - - DRC Totals - -
                 Total    Neg        Max        Max
Operation         Area   Slack      Trans      Cap
-------------------------------------------------------------------------
init_drc         2671    1368          0         0
            Path: I1/cout_reg_0/CK --> flag5
...
```

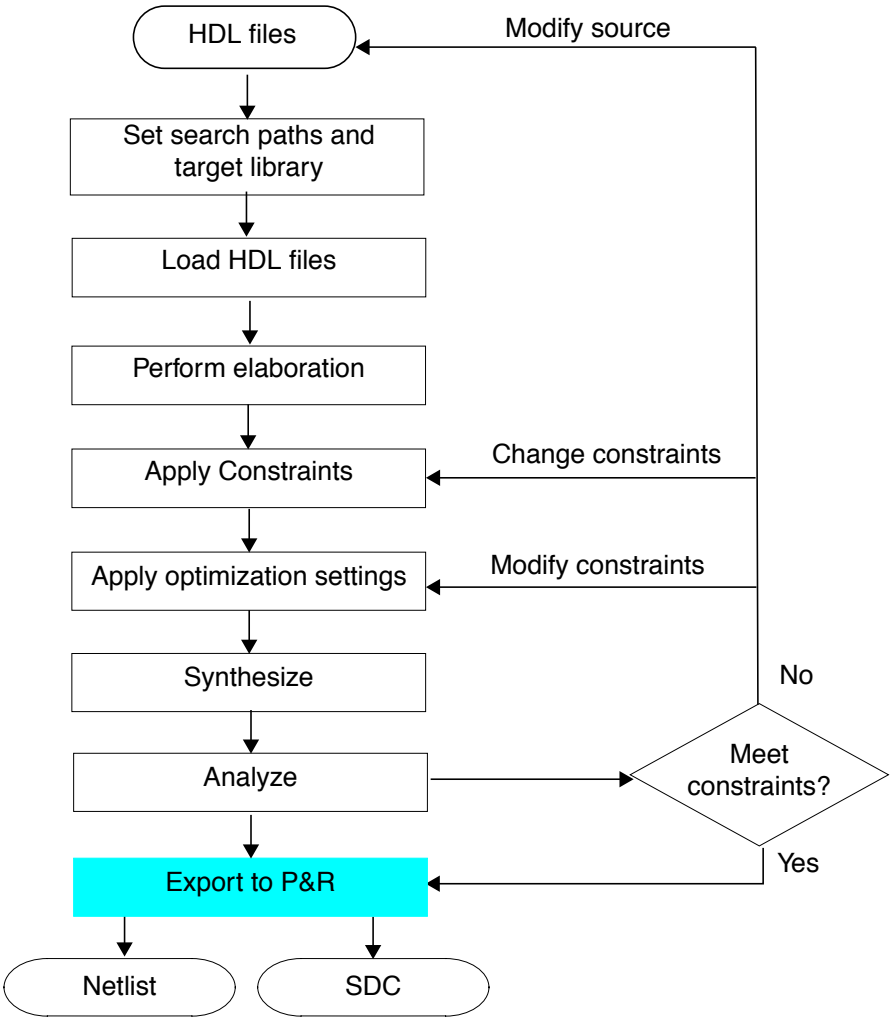
Depending on whether the *endpoint_slack_opto* attribute is turned on, RTL Compiler will work on either the Worst Negative slack or all the violating paths. You can track RTL Compiler's progress by looking at the Worst Path column in the log file.

As RTL Compiler works on the paths, the Total Area will be adjusted accordingly.

13

Interfacing to Place and Route

Overview



This chapter describes how to write out the synthesized design so that the netlist and constraints can interface smoothly with third-party tools.

Tasks

Writing Out the Design Netlist

The final part of the RTL Compiler flow involves writing out the netlists and constraints. This section describes how to write the design to a file using the `write_hdl` command. Use file redirection (>) to create a design file on disk, otherwise the `write_hdl` command, like all `write` commands, will direct its output to `stdout`.

Only two representations of the gate-level netlist are relevant to RTL Compiler:

- Mapped gate-level netlist
- RTL Compiler generic library mapped netlist

In order to write out a gate-level netlist, you must have mapped the RTL design to technology specific gates through the `synthesize -to_mapped` command. Alternatively, you could have loaded an already mapped netlist from a previous synthesis session.

➤ To write the gate-level netlist to a file called `design.v`, type the following command:

```
rc:/> write_hdl > design.v
```

Note: If you issue the `write_hdl` command before issuing the `synthesize -to_mapped` command, then a generic netlist will be written out since only such a netlist is available at that time.

➤ To write out only a specific design, specify the design name with the `write_hdl` command. The following command writes out the design `top` to a file called `top.v`:

```
rc:/> write_hdl  /designs/top/ > top.v
```

If you wanted to write out a specific subdesign, without its parent or child design, use the `write_hdl` command with the `unresolved` attribute:

```
rc:/> set_attribute unresolved true \
    [ get_attribute instance [ get_attribute subdesign bottom ] ]
rc:/> write_hdl [ find / -subdesign middle ] > middle.v
```

In this example, even though the `middle` design instantiates the `bottom` subdesign, only the `middle` design is written out to `middle.v`. This was intentionally done by setting the `unresolved` attribute to `true` on the `bottom` design.

➤ To write out a subdesign and any child designs it instantiates, specify the top-level design with the `write_hdl` command.

```
rc:/> write_hdl /designs/top/subdesign/middle/ > middle_and_bottom.v
```

In this example, the `middle` and its subdesign, `bottom`, were written out to `middle_and_bottom.v`.

For debugging and analysis purposes, it is sometimes useful to generate a gate-level representation of a design without using technology-specific cells. In this case, use the `-generic` option. You can write out a generic netlist either after issuing the `synthesize -to_generic` command or after `synthesize -to_mapped`.

➤ To create a gate-level netlist that is not technology specific, type the following command:

```
rc:/> write_hdl -generic > example_rtl.v
```

However, if you plan to use your netlist in a third-party tool, write out the technology specific gate-level netlist.

Writing SDC Constraints

After synthesizing your design, you can write out the design constraints in SDC format along with your gate-level netlist.

➤ To write out SDC constraints, type the following command:

```
rc:/> write_sdc
```

Like the other RTL Compiler commands, `write_sdc` prints the results to `stdout` unless specified otherwise. Therefore, make sure to specify the redirection character '>' along with the command.

➤ To write out the SDC constraints into `constraints.sdc` file, type the following command:

```
rc:/> write_sdc > constraints.sdc
```

Note: RTL Compiler writes out the SDC constraints in SDC format.

Connecting Pins, Ports, and Subports

The `edit_netlist connect` command connects two specified objects, and anything they might already be connected to, into one net. For example, if `A` and `B` are already connected and `C` and `D` are already connected, when you connect `A` and `C`, the result is a net connecting `A`, `B`, `C`, and `D`.

You can create nets that have multiple drivers and you can use `connect` to create combinational loops.

You cannot connect:

- Pins, ports, or subports that are in different levels of hierarchy. This is illegal Verilog.
- Pins, ports, or subports that are already connected
- An object to itself.
- A object that is driven by a logic constant to an object that already has a driver. This prevents you from shorting the logic constant nets together.
- To those objects that would require a change to a preserved module.

Disconnecting Pins, Ports, and Subports

The `edit_netlist disconnect` command disconnects a single subport, port, or pin from all its connections. For example, if `A`, `B`, and `C` are connected together and you disconnect `A`, then `B` and `C` remain connected to each other, but `A` is now connected to nothing else.

- You cannot disconnect any object that would require changes to a preserved module.
- You cannot disconnect an object that is not currently connected to anything else. If you disconnect an inout pin, it still remains connected to the other side.

Creating New Instances

The `edit_netlist new_instance` command creates an instance type in a specified level of the design hierarchy. You can instantiate inside a top-level design or a subdesign. There is an optional `name` subcommand.

- You cannot instantiate objects that require a change to a preserved module.
- You cannot create a hierarchical loop. If subdesign `A` contains subdesign `B`, then you cannot instantiate `A` again somewhere underneath `B`.

The `logic0` and `logic1` pins are visible in the directory so that you can connect to and disconnect from them. They are in a directory called `constants` and are called `1` an `0`. The following is how the top-level `logic1` pin appears in a design called `add`:

```
/designs/add/constants/1
```

The following is how a `logic0` pin appears deeper in the hierarchy:

```
/designs/add/instances_hier/ad/constants/0
```

You can refer to them by their shorter names:

```
add/1
add/ad/0
```

Each level of hierarchy has its own dedicated logic constants that can only be connected to other objects within that level of hierarchy.

Overriding Preserved Modules

If you have a script that you want to apply to all modules, even preserved modules, set the root attribute `ui_respects_preserve` to `false`.

The following code is a simple script that inserts a dedicated `tie-hi` or `tie-lo` to replace every constant in a design. The script demonstrates the edit netlist feature. This script could be extended to share the tie-offs up to some fanout limit.

```
# Iterate over all subdesigns and the top design
foreach module [find . -subdesign -design *] {
  # find the directory for this module where the logic constants live
  if {[string match [what_is $module] “design”]} {
    # we're at the top design
    set const_dir $module/constants
  } else {
    # we're at a subdesign
    set inst_dir [lindex [get_attribute instances $module] 0]
```

```
    set const_dir $inst_dir/constants
  }
  # Work on both logic constants
  foreach const {0 1} libpin {TIELO/Y TIEHI/Y} {
    # Find the logic 0 or logic 1 pin within this module
    set const_pin $const_dir/$const
    # find the libcell that we want to instantiate
    set libcell [find / -libcell [dirname $libpin]]

    # Find all the loads driven by this logic constant pin
    set net [get_attribute net $const_pin]
    if {[llength $net]} {
      foreach load [get_attribute loads $net] {
        # At each load instantiate a tie_inst
        set tie_insts \
          [edit_netlist new_instance -name "tie_${const}_cell" \
               $libcell $module]
        set tie_inst [lindex $tie_insts 0]
        # Find the output pin of the tie_inst to connect to
        set tie_pin $tie_inst/[basename $libpin]
        # Disconnect the load from the logic constant
        edit_netlist disconnect $load
        # Connect to the new tie_pin instead
        edit_netlist connect $load $tie_pin
        # Rename the net for extra credit
        mv -flexible [get_attribute net $load] "logic_${const}_net"
      }
    }
  }
}
```

Changing Names

When interfacing with different place and route tools, you may need to make modifications in the naming scheme of the gate-level netlist to suit the relevant back-end tool. To change the naming scheme, use the `change_names` command before writing out the netlist in your synthesis script file.

When you change the naming scheme with the `change_names` command, the change occurs immediately.

- To rename all `subdesign` objects with the `top_` prefix in the output netlist, type the following command:

  ```
  rc:/> change_names -prefix top_ -subdesign
  ```

- To add the suffix `_m` on all the `design` and `subdesign` objects, type the following command and options:

  ```
  rc:/> change_names -design -subdesign -suffix _m
  ```

- The following example will change all instances of lowercase `n` with uppercase `N` and underscores (_) with hyphens (-).

  ```
  rc:/> change_names -map {{“n”, “N”} {“_”, “-”}}
  ```

- In the following example, all instances of `@` will be replaced with `at`. If the `replace_char` option is not specified, the default character of underscore (_) will be used.

  ```
  rc:/> change_names -restricted “@” -replace_char “at”
  ```

- If the `case_insensitive` option is specified, then names which are otherwise differentiated will be considered identical based on the case of their constituent letters. For example, `n1` and `N1` will be considered as identical names.

  ```
  rc:/> change_names -case_insensitive
  ```

- The following example changes the brackets on an instance to “x”s.

  ```
  rc:/> change_names -instance -restricted {\[ \]} -replace_char "x"
  ```

- You cannot change the left bracket, "[", and the right bracket, "]" when they are a part of the bus name referencing individual bits of the bus. For example:

  ```
  rc:/designs/test/ports_in> ls
  ./       SI2      clk1     in1[0]   in2[0]   in2[3]   in3[2]   in3[5]
  rc:/designs/test/ports_in> change_names -port_bus -map {{"[" "("} {"]" ")"}}
  rc:/designs/test/ports_in> ls
  ./       SI2      clk1     in1[0]   in2[0]   in2[3]   in3[2]   in3[5]
  ```

- All changes are global unless the `-local` option is specified, in which case only the current directory is affected.

Creating Unique Parameter Names

Use the `parameter_naming_style` attribute to define the naming style for each binding (*`parameter`*, *`value`*).

- To specify naming style, type the following command:

  ```
  rc:/> set_attribute parameter_naming_style “_%d” /
  ```

  Ensure that you specify the attribute on the root-level (“/”).

Table 13-1 on page 231 illustrates the naming style results of various `parameter_naming_style` settings for the following example:

```
foo #(1,2) u0();
```

where the Verilog module is defined as:

```
module foo();
    parameter p = 0;
        parameter q = 1;
    endmodule
```

Table 13-1 Specifying Naming Styles

Naming Style Setting	Resulting Naming Style
`set_attribute parameter_naming_style "_%d" /`	`foo_1_2`
`set_attribute parameter_naming_style "_%s_%d" /`	`foo_p_1_q_2`
`set_attribute parameter_naming_style "" /`	`foo`

Note: This is the default attribute setting.

You can match the names generated by Design Compiler with the following variable settings in your script:

```
set hdlin_template_naming_style "%s_%p"
set hdlin_template_parameter_style "%d"
set hdlin_template_separator_style "_"
set hdlin_template_parameter_style_variable "%d"
```

➤ To match the names generated by Design Compiler, type the following command:

```
rc:/> set_attribute parameter_naming_style "_%d" /
```

Note: Values greater-than-32-bits are truncated in the name and parameter values are used in the name even if they are default values. Only one `%d` or a combination of `%d` and `%s` are accepted in this attribute.

Naming Flops

RTL Compiler uses the following default flop naming styles:

■ For vectored variables, that is `cout_reg_1`

<*var_name*>_reg_<idx>

- For scalar variables, that is `out1_reg`

  `<var_name>_reg`

You can change the naming style on the flops to match third-party requirements on the netlist. To customize the default naming scheme, use the following attributes:

- `hdl_reg_naming_style_vector`

  The default setting is:

  ```
  rc:/> set_attribute hdl_reg_naming_style_vector %s_reg_%d /
  ```

- `hdl_reg_naming_style_scalar`

  The default setting is:

  ```
  rc:/> set_attribute hdl_reg_naming_style_scalar %s_reg /
  ```

  where `%s` represents the signal name, and `%d` is the bit vector, or bit index.

Synopsys Design Compiler Compatibility Settings

To match Design Compiler nomenclature, specify the following:

```
rc:/> set_attribute hdl_reg_naming_style_vector %s_reg\[%d\]
rc:/> set_attribute hdl_reg_naming_style_scalar %s_reg
```

Two dimensional arrays will then be represented in the following format in the RTL Compiler output netlist.

```
<var_name>_reg_<idx1>_<idx2>
```

That is `cout_reg_1_1`

Removing Assign Statements

The generated gate-level netlist could contain `assign` statements like the following example:

```
... wire n_7, n_9;
assign dummy_out[0] = 1'b0;
assign dummy_out[1] = 1'b0;
assign dummy_out[2] = 1'b0;
assign dummy_out[3] = 1'b0;
assign dummy_out[4] = 1'b0;
assign dummy_out[5] = 1'b0;
...
assign dummy_out[15] = 1'b0;
DFFRHQX4 cout_reg_0(.D (n_15), .CK (clock), .RN (n_13), .Q
(cout[0]));
...
```

Since some place and route tools cannot recognize `assign` statements, you can use the `remove_assigns` command to replace them with buffer instantiations.

- In the following example, all `assign` statements will be recursively removed from the top-level design

  ```
  rc:/> remove_assigns
  ```

- In the following example, `assign` statements will only be removed from the `stormy` subdesign:

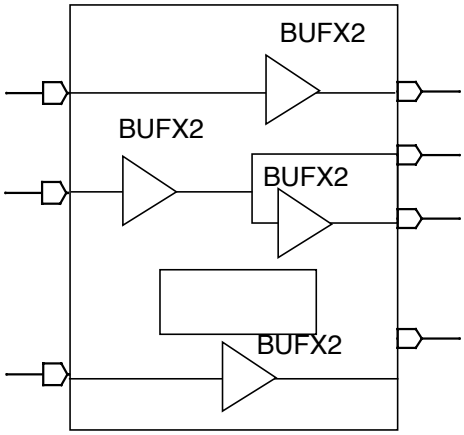
  ```
  rc:/> remove_assigns [find / -subdesign stormy]
  ```

- To specify a particular buffer to replace the `assign` statements, use the `-buffer` option. The following example recursively replaces `assign` statements with the `BUFX2` cell:

  ```
  rc:/> remove assigns -buffer BUFX2
  ```

The `remove_assigns` command is only valid after you synthesize the design with the `synthesize -to_mapped` command. Figure 13-1 graphically illustrates the result of using the `insert_io_buffers -remove_assigns` command.

Figure 13-1 Assign Statements Replaced with Buffers



Handling Bit Blasted Port Styles

Bit blasting is the process of individualizing multi-bit ports through nomenclature. For example, the following Verilog port `A` is defined to be four bits:

```
A[0:3]
```

Bit blasting this `A` port can produce the following result in the netlist:

```
A_0
```

```
A_1
A_2
A_3
```

This section describes the following two attributes available for handling bit-blasted ports:

- `bit_blasted_mapped_port`
- `bit_blasted_port_style`

Some place and route tools prefer to see port names in expanded format, rather than as vector representations, which is how RTL Compiler generates the gate-level netlists:

```
module addinc(A, B, Carry, Z);
    input [7:0] A, B;
...
```

Since this format may not be fully recognized by some tools, you have to set the attribute `bit_blasted_mapped_port` to `true`.

```
rc:/> set_attribute bit_blast_mapped_port true
```

Then the generated netlist will look like this:

```
module addinc(A_7, A_6, A_5, A_4, A_3, A_2, A_1, A_0, B_7,
B_6, B_5, B_4, B_3, B_2, B_1, B_0, Carry, Z_8, Z_7,
Z_6, Z_5, Z_4, Z_3, Z_2, Z_1, Z_0);
  input A_7;
  input A_6;
....
```

To control the bit blasted port naming style, use the `bit_blasted_port_style` attribute. If you type the following command for the above example

```
rc:/> set_attribute bit_blasted_port_style %s\[%d\]
```

The output netlist will look like this:

```
module addinc(\A[7] , \A[6] , \A[5] , \A[4] , \A[3] , \A[2] , \A[1] ,
    \A[0] , \B[7] , \B[6] , \B[5] , \B[4] , \B[3] , \B[2] , \B[1] ,
    \B[0] , Carry, \Z[8] , \Z[7] , \Z[6] , \Z[5] , \Z[4] , \Z[3] ,
    \Z[2] , \Z[1] , \Z[0] );
  input \A[7] ;
  input \A[6] ;
...
```

Naming Generated Components

The `gen_module_prefix` attribute sets all internally generated modules, such as arithmetic, logic, register-file modules, and so on, with a user-defined prefix. This enables you to identify these modules easily. Otherwise, the modules will have the RTL Compiler internally generated names.

For example, if you were to set the attribute to `CDN_DP_` by typing:

```
rc:/> set_attribute gen_module_prefix  CDN_DP_ /
```

this will generate the modules with the `CDN_DP_` prefix.

If you prefer to remove or ungroup these modules, you should type the following command after the design is synthesized:

```
rc:/> edit_netlist ungroup [find /des* -subdesign CDN_DP*]
```

Changing the Instance Library Cell

After RTL Compiler completes the optimization and maps the design to the technology library cells, all the instances in the design will refer to the technology library.

You can find out the corresponding library cell name by checking the *libcell* attribute on each instance.

For example, if you want to find out what the `cout_reg_5` instance is mapped to in the technology library, type the following command:

```
rc:/> get_attribute libcell designs/top_counter/instances_hier/I2/instances_seq/
cout_reg_5
```

RTL Compiler will show the library cell and its source library:

```
/libraries/slow/libcells/DFFRHQX4
```

To manually force the instance to have a different library cell, you can use the same *libcell* attribute. If you want to replace one pin with another, the pin mappings must be equal.

For example, if you want to use `DFFRHQX2` instead of `DFFRHQX4` on the `cout_reg_5` instance, type the following command:

```
rc:/> set_attribute libcell [find / -libcell DFFRHQX2] \
   designs/top_counter/instances_hier/I2/instances_seq/cout_reg_5
```

This command will force the instance to be mapped to *DFFRHQX2*.

Note: Make sure to generate all the reports, especially a timing report, to ensure that no violations exist in the design.

Handling Bit-Blasted Constants

Typically, before sending the netlist to the place and route flow, there will be some netlist preparation. In order for some of the netlist changes to take effect, you should execute a low

effort, incremental synthesis to realize the full effects of the changes. That is, the changes should be followed by typing the following command:

```
rc:/> synthesize -to_mapped -effort low -incremental
```

For example, some place and route tools cannot properly handle bus constants in the netlist. If you are using APR tools with this limitation, set the `bit_blast_constants` attribute to `true` before writing out the netlist:

```
rc:/> set_attr bit_blast_constants true /
```

This attribute tells RTL Compiler to bit blast the constants. For example, if there is a constant `7'b0`, then it will be represented as `{1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0}`.

This attribute is only used by the `write` command, so there is no need to issue the `synthesize` command with its `-to_mapped` and `-incremental` options.

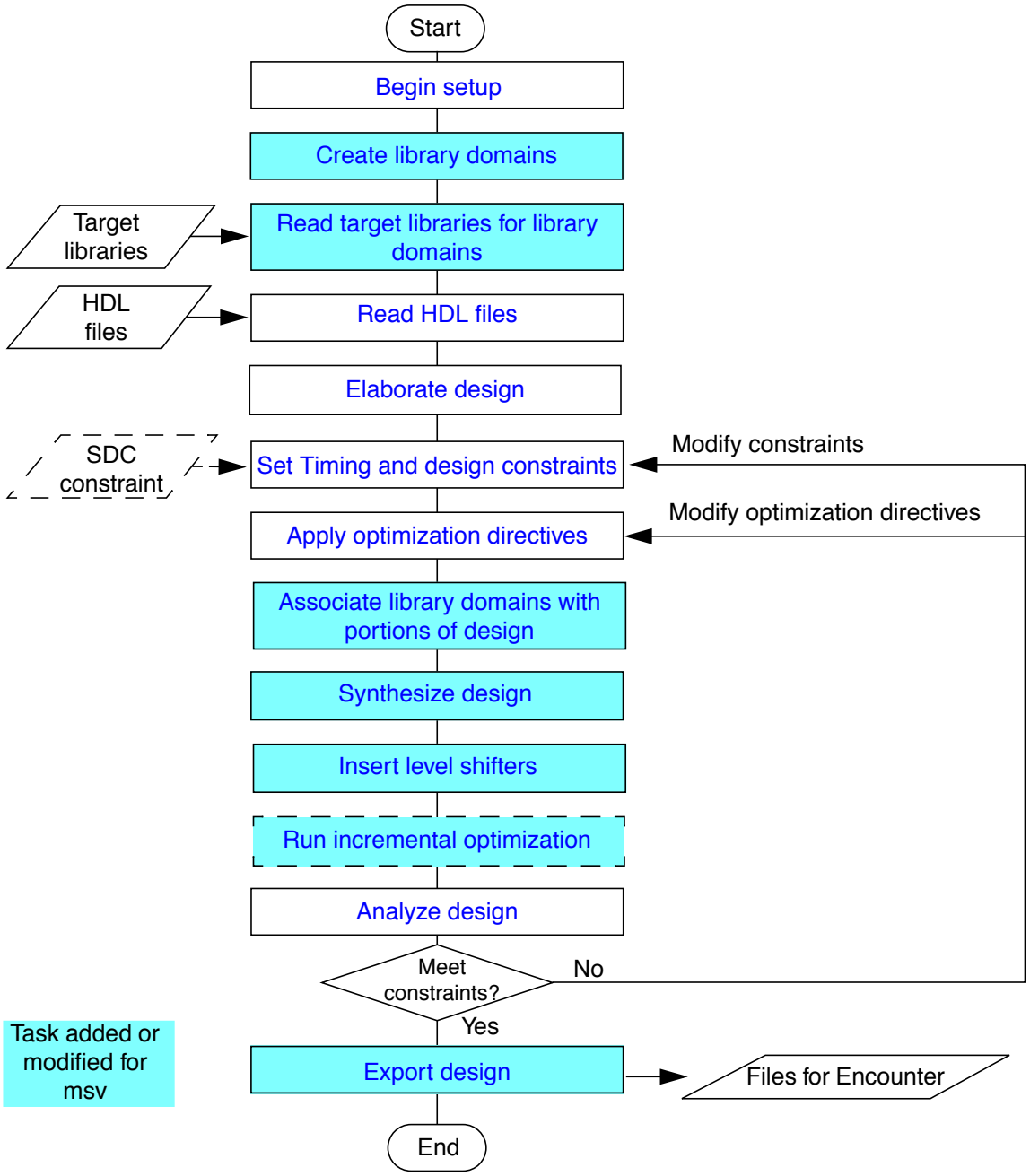
14

The Multiple Supply Voltage Flow

Overview

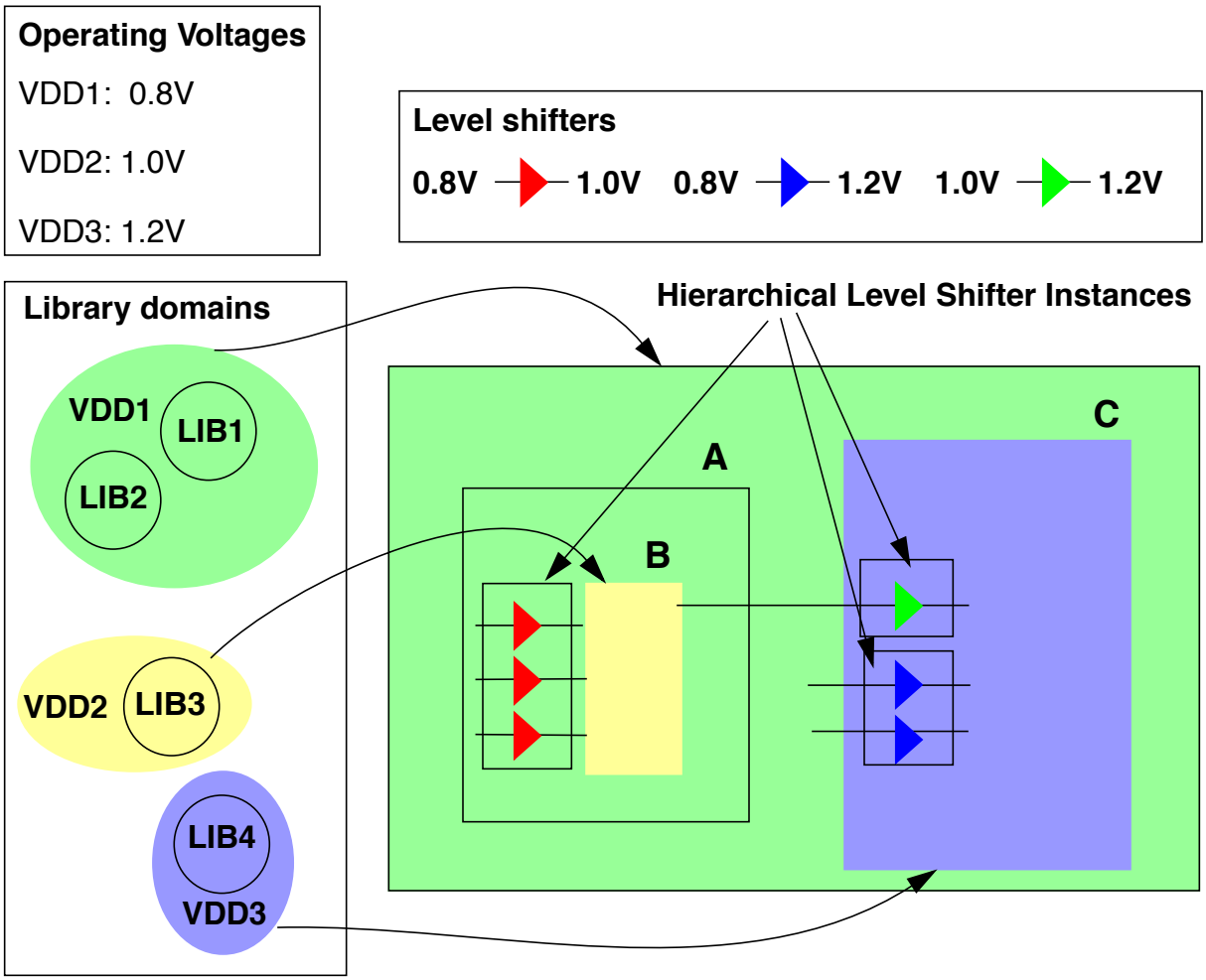
This chapter describes the top-down synthesis flow using multiple supply voltages. This flow, further referred to as the *MSV* flow, is shown in Figure 14-1. A script for the common MSV flow is shown in Example 14-1 on page 241.

Figure 14-1 Top-Down MSV Flow



In an MSV design, multiple supply voltages can be used for the core logic. You can group libraries that were characterized for the same nominal operating conditions in a *library domain*. These library domains can then be associated with different portions of the design, for example to indicate the voltage on which these portions are operating. Level shifters are needed to pass data signals between portions of the design that operate on different voltages. In Figure 14-2, the top design and subdesign A operate on voltage `VDD1` while subdesign B operates on voltage `VDD2` and subdesign C operates on voltage `VDD3`. For each voltage, a library domain with the corresponding name was created. Libraries `LIB1` and `LIB2` were associated with library domain `VDD1`, and so on. Three types of level shifters were defined. RTL Compiler groups level shifters inserted to pass signals between the same domains in a hierarchical instance. For example, the level shifters on signals between domain `VDD1` and `VDD2` are stored in the originating domain (`from_domain`).

Figure 14-2 Library Domains and Level Shifters



Because the generic flow was described in Chapter 2, “The RTL Compiler Work Flow,” this chapter will focus on the additional steps for the MSV flow. For more information, see Flow Steps.

MSV Information in the Design Information Hierarchy shows where the additional MSV-specific information is stored.

When using multiple supply voltages in the design you can also perform a what-if analysis to determine which configuration results in better timing and power. This analysis and some other tasks that are not part of the Flow Steps are described in Additional Tasks.

Tip

Library domains can also be used for designs where you want to use specific libraries for some of the blocks. For more information, refer to Using Library Domains for Non-MSV Design.

Example 14-1 Script for Common Top-Down MSV Flow

```
# general setup
#--------------
set_attributer lib_search_path ...
set_attribute hdl_search_path ..

# create library domains
#-----------------------
create_library_domain domain_list

# specifiy the target libraries for each library domain
#------------------------------------------------------
set_attribute library library_list1 [find /libraries -library_domain domain1] /
set_attribute library library_list2 [find /libraries -library_domain domain2] /
...

# load and elaborate the design
#------------------------------
read_hdl design.v
elaborate

# specify timing and design constraints
#--------------------------------------
# specify the following constraints per library domain
#-----------------------------------------------------
set_attr operating_conditions string [find /libraries -library_domain domain]
set_attr wireload_selection string [find /libraries -library_domain domain]

# set target library domain for top design
#-----------------------------------------
set_attribute library_domain library_domain design

# set target library domain for blocks
#-------------------------------------
edit_netlist uniquify subdesign
set_attribute library_domain library_domain subdesign

#synthesize the design
#---------------------
synthesize -to_mapped

# insert level shifters
#----------------------
define_level_shifter_group -from_library_domain library_domain \
-to_library_domain library_domain [-name string] \
-libcells cell_list

level_shifter insert [-from_library_domain library_domain] \
[-to_library_domain library_domain] [-location {from | to}] \
[-dedicate_level_shifter libcell]

report level_shifter -hier -detail

# analyze design
------------------
report timing
report gates
report power

# export design
#--------------
write_fe_msv [-design design] [-output_directory path] \
[-template_config file] [-template_setup file]\
[-template_level_shifter file] [-reference_config file]\
[-lef_list_file file] [> file]
```

Flow Steps

Begin Setup

For more information on the setup, see

- Generating Log Files on page 31
- Generating the Command File on page 32
- Setting Information Level and Messages on page 32
- Specifying Explicit Search Paths on page 33

Create Library Domains

A library domain is a collection of libraries. You use library domains to indicate that portions of your design operate on different *voltages*. You must create a library domain for each supply voltage.

➤ To create library domains, use the `create_library_domain` command:

```
create_library_domain domain_list
```

Note: There is no limitation on the number of library domains you can create.

This command returns the directory path to the library domains that it creates. You need this information when loading in the target libraries.

To get meaningful timing results, all libraries within a library domain *should* have been characterized for the same nominal operating conditions. If the libraries have different operating conditions, the nominal operating conditions of the last library will be used and thus the last library also determines the voltage of the library domain. For example to find the active operating condition of a specific domain, use

```
get_att active_operating_conditions [find / -library_domain lib*/*domains/domain]
```

To find the voltage of a domain, you need use the active operating condition of the domain:

```
get_att voltage [get_att active_operating_conditions \
[find / -library_domain lib*/library_domains/domain]]
```

Tip

You can always rename or remove a library domain. For more information on removing library domains, refer to Removing a Library Domain.

Read Target Libraries for Library Domains

RTL Compiler needs to know which library cells to use for a block that uses a particular supply voltage. Therefore, you need to associate the libraries that are characterized for a specific voltage (or set of operating conditions) with the library domain that corresponds to that supply voltage.

➤ To read in the libraries for a specific voltage, set the `library` attribute for the corresponding domain:

```
set_attribute library library_list [find /libraries -library_domain domain]
```

Note: There is no limitation on the number of libraries you can read in per domain.

The first library domain for which you read in the libraries, becomes the default library domain.

➤ To change the default library domain, set the following attribute:

```
set_attribute is_default true desired_library_domain
```

For more information on the use of the default library domain, refer to Removing a Library Domain.

Read HDL Files

For more information on reading HDL files, see Loading HDL Files on page 102.

Elaborate Design

For more information on elaborating a design, see Performing Elaboration on page 127.

Set Timing and Design Constraints

Except for the following constraints, which *should* be set per library domain, all other (SDC and RC native) constraints are set as in the regular top-down flow.

```
set_attr operating_conditions string [find /libraries -library_domain domain]
set_attr wireload_selection string [find /libraries -library_domain domain]
```

For more information on setting design constraints, see "Applying Constraints" on page 147.

Tip

When you set the `force_wireload` attribute on a design or subdesign, make sure that the wireload model you set matches a wireload model defined for a library in the associated library domain.

Apply Optimization Directives

For more information on optimization strategies and related commands, see "Defining Optimization Settings" on page 153.

Associate Library Domains with Different Blocks of Design

Different blocks of your design may operate on different voltages or you may want to use dedicated libraries for some blocks of the design. To inform RTL Compiler about the voltage usage or special library use, you need to associate these blocks with the appropriate library domains.

- To set the target library domain for the top design, specify the `library_domain` attribute on the design:

  ```
  set_attribute library_domain library_domain design
  ```

- To set the target library domain for a subdesign, do the following

  a. Uniquify the subdesign:

  ```
  edit_netlist uniquify subdesign
  ```

  See Example 14-2 for more information.

  b. Specify the `library_domain` attribute on the subdesign:

  ```
  set_attribute library_domain library_domain subdesign
  ```

  Note: This attribute is hierarchical. It applies to all instances of the specified design or subdesign. So the order in which you specify the target domains is important. See Example 14-3 for more information.

When you *change* the library domain for a subdesign, RTL Compiler copies all attributes from the original mapped instances to the new mapped instances.

Important

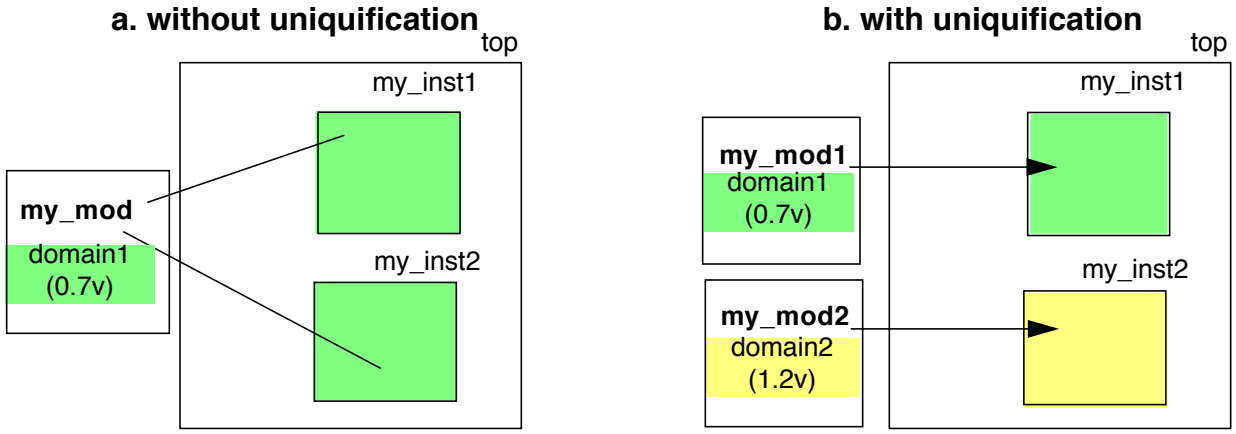
If you marked an instance *preserved*, the library domain that the instance is associated with will still be changed. However, the instance will still be marked preserved even though it will probably be pointing to another library cell in the new library domain. In other words, the `library_domain` attribute has a higher priority than the `preserve` attribute.

If RTL Compiler cannot find the corresponding cell in the libraries that belong to the new library domain, the instance becomes *unresolved*.

Example 14-2 Uniquifying a Subdesign before Associating the Library Domain

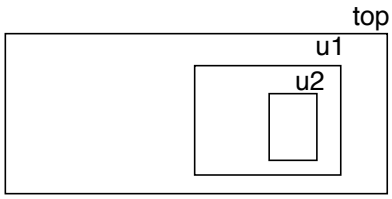
In the following design, module `top` has two instantiations of module `my_mod`. If you associate subdesign `my_mod` with library domain `domain1` before uniquifying subdesign `my_mod`, both instances `my_inst1` and `my_inst2` are associated with library domain `domain1`. This is illustrated in Figure 14-3. To associate instance `my_inst1` with library domain `domain1`, and instance `my_inst2` with library domain `domain2`, you first need to uniquify subdesign `my_mod`, and then associate both subdesigns with their domains individually.

Figure 14-3 Uniquifying a Subdesign before Associating the Library Domain



Example 14-3 Setting Target Library Domains

Assume a design `top` which has a subdesign `u1`. Subdesign `u1` has a subdesign `u2`.



Assume that all instances of the u2 should be mapped using libraries from library domain `dom1`. All instances of u1, except for the instances of u2, should be mapped using libraries from library domain `dom2`. The instances in the remainder of the design should be mapped using libraries from library domain `dom1`.

To ensure the correct mapping, make the assignments in the following order:

```
set_attribute library_domain [find / -library_domain dom1] /designs/top
set_attribute library_domain [find / -library_domain dom2] [find / -subdesign u1]
set_attribute library_domain [find / -library_domain dom1] [find / -subdesign u2]
```

Synthesize Design

After the constraints and optimizations are set for your design, you can proceed with synthesis by issuing the `synthesize` command.

➤ To synthesize your design using the `synthesize` command, type:

```
synthesize -to_mapped
```

Note: The design can be synthesized top down, without the need for manual partitioning.

The different portions of the design that are associated with different library domains will be mapped to the target libraries of those library domains and optimized.

For details on the `synthesize` command, see "Synthesizing your Design" on page 172.

Insert Level Shifters

When passing data signals between library domains representing different voltages, you need to insert level shifters.

1. To specify which library cells to use between two specific library domains, use the `define_level_shifter_group` command:

   ```
   define_level_shifter_group -from_library_domain library_domain
   -to_library_domain library_domain [-name string]
   -libcells cell_list
   ```

   This command creates a level shifter group which points to the appropriate library cells for the specified library domains.

   The specified cells must belong to the same library domains and must have the same functionality. Each cell can belong to only one level-shifter group.

   Any library cells that you refer to in the `define_level_shifter_group` command must follow the Level Shifter Requirements described in the *Library Guide for Encounter RTL Compiler.*

Tip

You can rename a level-shifter group using the `mv` command. You can remove a level-shifter group using the `rm` command.

2. To insert level-shifter instances, use the `level_shifter insert` command:

   ```
   level_shifter insert [-from_library_domain library_domain]
   [-to_library_domain library_domain] [-location {from | to}]
   [-dedicate_level_shifter libcell]
   ```

You can insert all level shifters at once, or you can insert them for the specified domains only using either existing level-shifter cells or using a dedicated library cell (AND gate or buffer). However, only level-shifter cells that belong to the same level-shifter group can be used for resizing.

Important

RTL Compiler can only insert level shifters between two library domains if you defined the appropriate level-shifter group or you specified a dedicated library cell.

RTL Compiler will not insert level shifters on

- primary inputs and outputs
- constant nets
- dangling pins or nets
- multiple-driver nets if all or some drivers reside in different library domains and you request to insert the level shifters in the destination library domain (to domain)
- multiple-fanout nets if all or some loads reside in different library domains and you request to insert the level shifters in the originating library domain (from domain)

3. To report all the level shifters inserted in the design, use the `report level_shifter` command:

```
report level_shifter -hier -detail
```

Finding Level Shifters in the Hierarchy

RTL Compiler creates a separate subdesign for the level shifters connecting two library domains. You can find the level-shifter subdesigns in the design hierarchy at

`/designs/`*`design`*`/subdesigns/RC_LS_MOD*`

You can find the path to a level-shifter instance in the design hierarchy using the `find` command:

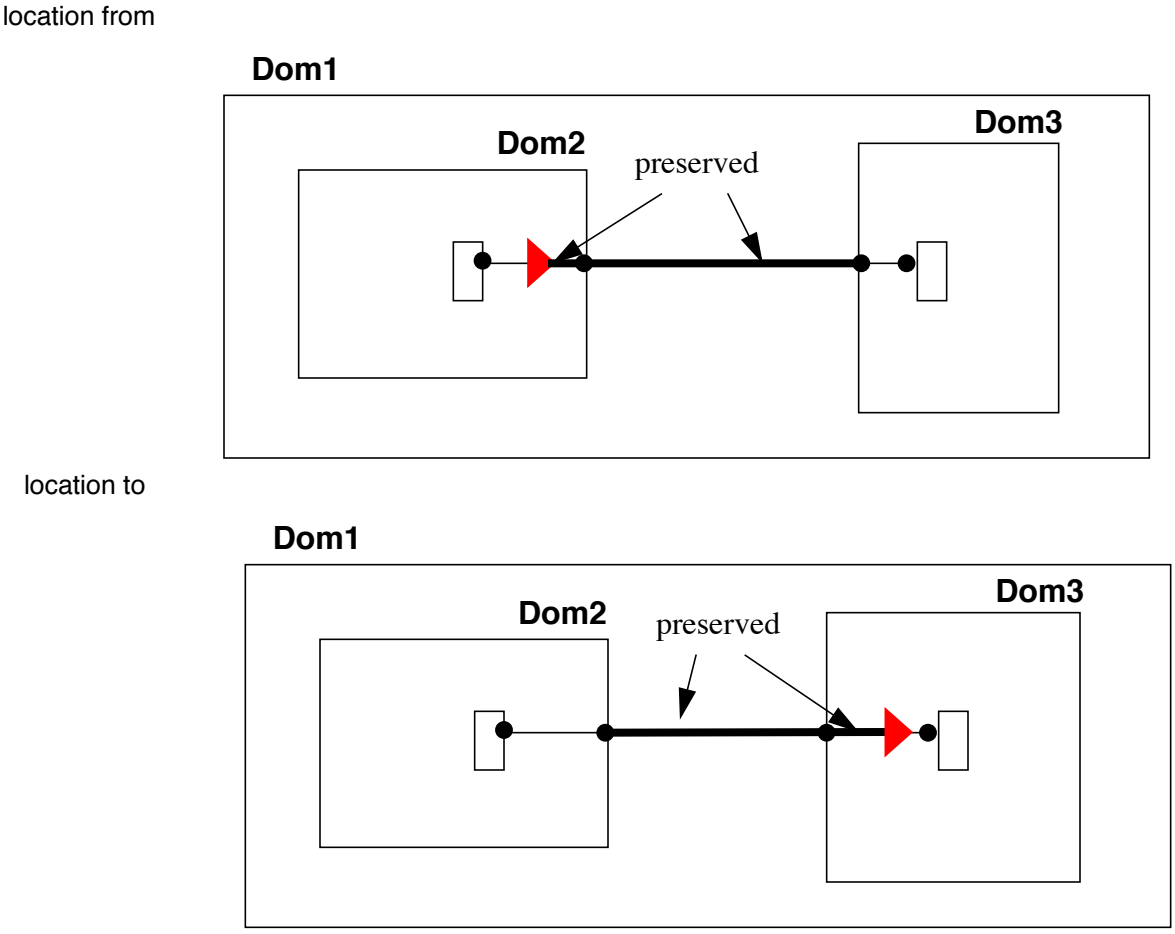
```
find / -instance RC_LS_HIER_INST*
```

Preserve Actions

RTL Compiler automatically sets the `preserve` attribute on the level-shifter subdesigns to `size_delete_ok`. However, if you use a dedicated library cell for level shifter insertion, RTL Compiler sets the `preserve` attribute value to `true` on the level-shifter subdesigns.

When a net crosses multiple library domains, RTL Compiler automatically sets the preserve attribute on some portions of the net to prevent buffer insertion as illustrated in Figure 14-4.

Figure 14-4 Preserving Nets to Prevent Buffer Insertion



Run Incremental Optimization

Inserting the level shifters can have an impact on the timing.

To fix any timing issues, run incremental optimization:

```
synthesize -incremental
```

Analyze Design

After synthesizing the design, you can generate detailed timing and area reports using the various `report` commands.

For more information on generating reports for analysis, see "Generating Reports" on page 209 and "Analysis Commands" in the *Command Reference for Encounter RTL Compiler*.

Most reports reflect information for the library domains. The following example shows the top of the `report gates` report.

```
rc:/> report gates
============================================================
Generated by:             RTL Compiler-D (RC) version
Generated on:             date
  Module:                 top
  Library domain:         1v
    Domain index:         0
    Technology libraries: lib1_1v08 1.1
                          lib2_1v08 1.1
    Operating conditions: oc_1v08 (balanced_tree)
  Library domain:         07v
    Domain index:         1
    Technology libraries: lib1_0v70 1.1
                          lib2_0v70 1.1
                          ls_0v70_1v08 1.0
    Operating conditions: oc_0v70 (balanced_tree)
  Library domain:         08v
    Domain index:         2
    Technology libraries: lib1_0v80 1.1
                          lib2_0v80 1.1
                          ls_0v80_1v08 1.0
                          ls_0v70_0v80 1.0
    Operating conditions: oc_0v80 (balanced_tree)
  Wireload mode:          enclosed
============================================================

                                                      Domain
      Gate      Instances    Area          Library      Index
------------------------------------------------------------
AND2X                   1     5.884    lib2_1v08         0
AOI22X                 16   131.792    lib2_0v70         1
AOI22X                 15   123.555    lib2_1v08         0
AOI22X                  7    57.659    lib2_0v80         2
CLKINVX1                1     3.530    lib2_0v70         1
....
```

The `Domain index` in the report header is introduced for easy referencing in the remainder of the report. This is most useful when the library domains have long names.

This report shows that RTL Compiler used the same cell `AOI22X` from three library domains.

In the timing report (shown in Figure 14-5 on page 251), the domain index allows you to see how the critical path goes from one library domain to the next.

Figure 14-5 Timing Report

```
rc:/> report timing
Warning : Possible timing problems have been detected in this design. [TIM-11]
        : The design is 'top'.
============================================================
Generated by:              RTL Compiler-D (RC) version
Generated on:              date
  Module:                  top
  Library domain:          0.8v
    Domain index:          0
    Technology library:    ss_0v80 1.1
    Operating conditions: oc_0v80 (balanced_tree)
  Library domain:          0.9v
    Domain index:          1
    Technology library:    ss_0v90 1.1
    Operating conditions: oc_0v90 (balanced_tree)
  Library domain:          lvl
    Domain index:          2
    Technology libraries: tutorial 1.0
                           slow_0v80_0v90 1.0
    Operating conditions: typical_case (balanced_tree)
  Wireload mode:           enclosed
============================================================

       Pin               Type       Fanout Load Slew Delay Arrival
                      (Domain index)        (fF) (ps)  (ps)   (ps)

-------------------------------------------------------------------
(clock clock)          launch                                   0 R
(ls_opt.sdc_line_2)    ext delay                      +0        0 R
in1                    in port            1  7.5    0    +0        0 R
i1/in1
  g4/B                                                +0        0
  g4/Y                 AND2X1(0)          1 13.2   92  +190      190 R
i1/out
ls/in
  i/A                                                 +0      190
  i/Y                  LevLHX8(2)         1  3.7   63  +535      725 R
ls/out
i2/in1
  g4/A                                                +0      725
  g4/Y                 AND2X4(1)          1  0.5   48  +125      850 R
i2/out
top/out                out port                        +0      850 R
(ls_opt.sdc_line_3)    ext delay                       +0      850 R
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
(clock clock)          capture                                  0 R
-------------------------------------------------------------------
Cost Group   : 'clock' (path_group 'clock')
Timing slack :    -850ps (TIMING VIOLATION)
Start-point  : in1
End-point    : out
```

Export to Place and Route

➤ To export the necessary files for the Encounter place and route tool, use the following command:

```
write_fe_msv [-design design] [-output_directory path]
[-template_config file] [-template_setup file]
[-template_level_shifter file] [-reference_config file]
[-lef_list_file file] [> file]
```

This command writes the netlist file, the SDC constraint file, templates for the Encounter config file, Encounter setup file, and Encounter level-shifter table file.

Important

This command creates template files. It is your responsibility to fill in the remaining information needed to start the Encounter place and route tool.

Encounter Command Template

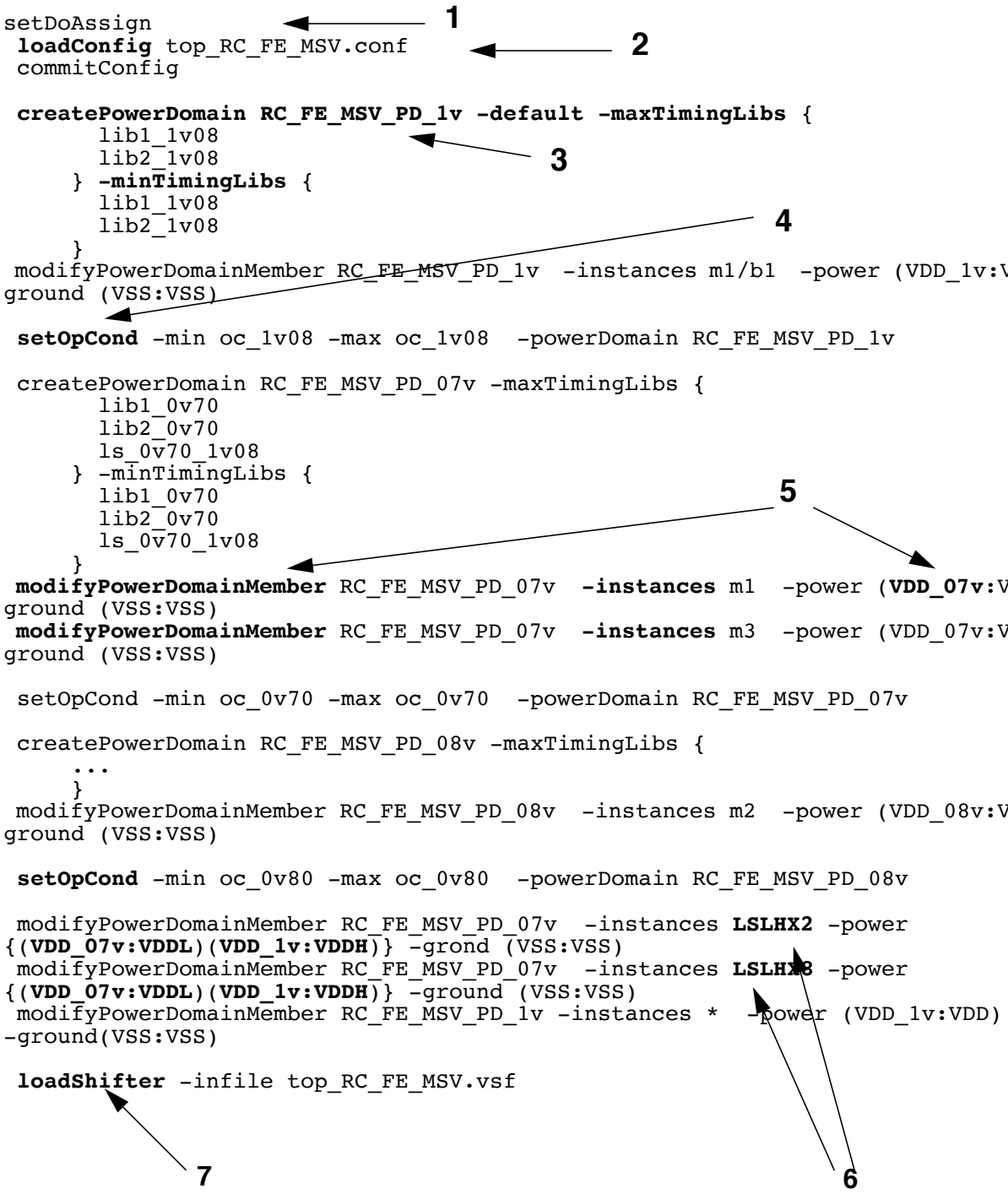
The `write_fe_msv` command creates a *template* for the Encounter command file and sets the following information:

1. Removes the assignment statement for the netlist (see **1** in Figure 14-6).
2. Loads in the Encounter config template file that it created (see **2** in Figure 14-6).
3. Creates power domains in Encounter (see **3** in Figure 14-6).
   - The `write_fe_msv` command adds a prefix (`RC_FE_MSV_PD`) to the library domain name defined in RTL Compiler if that name starts with a digit.
   - Changes the *default* library domain to the domain associated with the design because that is the default in Encounter.
   - The `write_fe_msv` command lists the *same* libraries for `maxTimingLibs` and `minTimingLibs`.
4. Specifies the operating conditions for each power domain (see **4** in Figure 14-6).

   Note: Encounter does not read the operating conditions from the SDC constraint file.
5. Associates the design and instances with the power domains (see **5** in Figure 14-6).
   - If several instances are associated with the same power domain, a separate entry is required for Encounter.
   - The `write_fe_msv` command names the power net differently for each library domain.
6. Specified the power and ground pin for each level-shifter library cell (see **6** in Figure 14-6).
7. Loads the level shifter file (see **7** in Figure 14-6).

Figure 14-6 Sample Encounter Command Template File

```
setDoAssign
 loadConfig top_RC_FE_MSV.conf
 commitConfig

 createPowerDomain RC_FE_MSV_PD_1v -default -maxTimingLibs {
       lib1_1v08
       lib2_1v08
     } -minTimingLibs {
       lib1_1v08
       lib2_1v08
     }
 modifyPowerDomainMember RC_FE_MSV_PD_1v  -instances m1/b1  -power (VDD_1v:VDD)
ground (VSS:VSS)

 setOpCond -min oc_1v08 -max oc_1v08  -powerDomain RC_FE_MSV_PD_1v

 createPowerDomain RC_FE_MSV_PD_07v -maxTimingLibs {
       lib1_0v70
       lib2_0v70
       ls_0v70_1v08
     } -minTimingLibs {
       lib1_0v70
       lib2_0v70
       ls_0v70_1v08
     }
 modifyPowerDomainMember RC_FE_MSV_PD_07v  -instances m1  -power (VDD_07v:VDD)
ground (VSS:VSS)
 modifyPowerDomainMember RC_FE_MSV_PD_07v  -instances m3  -power (VDD_07v:VDD)
ground (VSS:VSS)

 setOpCond -min oc_0v70 -max oc_0v70  -powerDomain RC_FE_MSV_PD_07v

 createPowerDomain RC_FE_MSV_PD_08v -maxTimingLibs {
     ...
     }
 modifyPowerDomainMember RC_FE_MSV_PD_08v  -instances m2  -power (VDD_08v:VDD)
ground (VSS:VSS)

 setOpCond -min oc_0v80 -max oc_0v80  -powerDomain RC_FE_MSV_PD_08v

 modifyPowerDomainMember RC_FE_MSV_PD_07v  -instances LSLHX2 -power
{(VDD_07v:VDDL)(VDD_1v:VDDH)} -grond (VSS:VSS)
 modifyPowerDomainMember RC_FE_MSV_PD_07v  -instances LSLHX8 -power
{(VDD_07v:VDDL)(VDD_1v:VDDH)} -ground (VSS:VSS)
 modifyPowerDomainMember RC_FE_MSV_PD_1v -instances *  -power (VDD_1v:VDD)
-ground(VSS:VSS)

 loadShifter -infile top_RC_FE_MSV.vsf
```



Encounter Config Template

The `write_fe_msv` command creates a *template* for the Encounter config file that fills in the following information (as shown in Figure 14-7): the name of the netlist file (**1**), the type of the netlist file (**2**), the names of the timing libraries (**3**), the name of the constraint file (**4**), (if provided) the list of the LEF files (**5**), the power net names that RTL Compiler created based on the domain names (**6**). The power and net names are also used in the command file.

Figure 14-7 Sample Encounter Config Template File

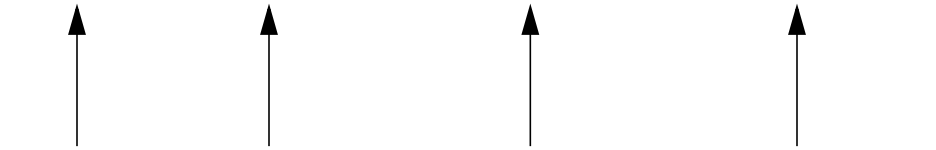
```
global rda_Input
set rda_Input(import_mode) {-treatUndefinedCellAsBbox 0  -keepEmptyModule 1 -
useLefDef56 1 }
set rda_Input(ui_netlist) " top.v "                          <-- 1
set rda_Input(ui_netlisttype) {Verilog}                      <-- 2
set rda_Input(ui_rtllist) ""
set rda_Input(ui_ilmlist) ""
set rda_Input(ui_ilmspef) ""
set rda_Input(ui_settop) {1}
set rda_Input(ui_topcell) {top}
set rda_Input(ui_celllib) ""
set rda_Input(ui_iolib) ""
set rda_Input(ui_areaiolib) ""
set rda_Input(ui_blklib) ""
set rda_Input(ui_kboxlib) ""
set rda_Input(ui_gds_file) ""
set rda_Input(ui_timelib,min)  " \                           <-- 3
/my_path/lib1_1v08.lib \
...
"
set rda_Input(ui_timelib,max)  " \
/my_path/lib1_1v08.lib \
...
"
set rda_Input(ui_timelib,typ) ""
set rda_Input(ui_timelib)  ""
set rda_Input(ui_smodDef) ""
set rda_Input(ui_smodData) ""
set rda_Input(ui_dpath) {}
set rda_Input(ui_tech_file) ""
set rda_Input(ui_io_file) ""
set rda_Input(ui_timingcon_file) "top.sdc"                   <-- 4
set rda_Input(ui_latency_file) ""
set rda_Input(ui_scheduling_file) ""
set rda_Input(ui_buf_footprint) {}
set rda_Input(ui_delay_footprint) {}
set rda_Input(ui_inv_footprint) {}
set rda_Input(ui_leffile) "                                  <-- 5
my_tech.lef \
...
"
...
set rda_Input(ui_pwrnet) { VDD_1v VDD_07v VDD_08v }          <-- 6
set rda_Input(ui_gndnet) {VSS}
...
```

Encounter Shifter Table File

The `write_fe_msv` command also creates a shifter table file. A sample is shown in Figure 14-8.

Figure 14-8 Sample Encounter Shifter Table File

```
LSLHX2   RC_FE_MSV_PD_07v  RC_FE_MSV_PD_1v RC_FE_MSV_PD_07v
LSLHX8   RC_FE_MSV_PD_07v  RC_FE_MSV_PD_1v RC_FE_MSV_PD_07v
```



Level shifter .lib cell name | From power domain | To power domain | Location is same as from domain.

LEF File List

You can specify the LEF files that the `write_fe_msv` command needs to fill in the Encounter config template file. Enter the path to each LEF file on a separate line. A sample is shown in Figure 14-9.

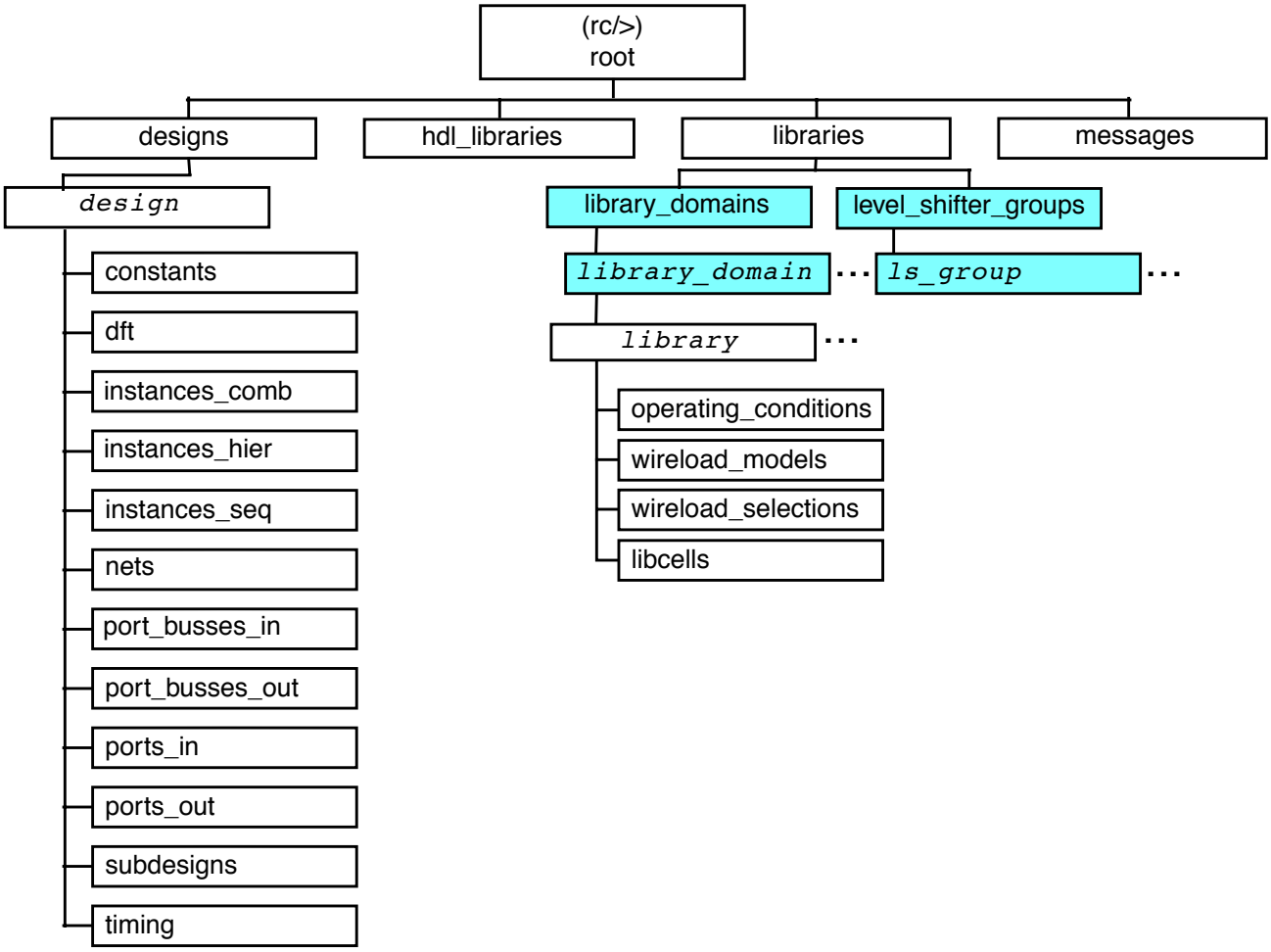
Example 14-4 Sample LEF Files List

```
/mypath/LIB/my_tech.lef
/mypath/LIB/my_macros.lef
/mypath/LIB/my_lvl.lef
/mypath/LIB/myhvt_macros.lef
```

MSV Information in the Design Information Hierarchy

RTL Compiler stores the original design data along with additional information added by RTL Compiler in the design information hierarchy in the form of attributes. Figure 14-9 highlights the MSV information in the design information hierarchy.

Figure 14-9 Design Information Hierarchy



Additional Tasks

Removing a Library Domain

➤ To remove a library domain, use

```
rm [find /libraries -library_domain domain]
```

When you remove a library domain, RTL Compiler removes

- The libraries that are part of that library domain
- Any level-shifter group that was referring from or to this library domain

Additional Notes

- RTL Compiler links any instances in the subdesigns associated with a library domain that is being removed to the default library domain. While relinking, RTL Compiler copies all attributes from the original mapped instance to the new mapped instance.

Important

If you marked an instance *preserved*, the library domain that the instance is associated with will still be removed. However, the instance will still be marked preserved even though it will probably be pointing to another library cell in the default library domain. In other words, the `library_domain` attribute has a higher priority than the `preserve` attribute.

If RTL Compiler cannot find the corresponding cell in the libraries that belong to the default library domain, the instance becomes *unresolved*.

- You can remove the library domain that is marked the default library domain, by first setting the `is_default` attribute on another library domain.

  In the following example, `dom_1` is the default library domain. You can only remove library domain `dom1`, after changing the `is_default` for example to library domain `dom_2`,

```
rc:/> get_attribute is_default [find /libraries -library_domain dom_1]
true
rc:/> set_attribute library typical.lib dom_2
Setting attribute of library_domain dom_2: 'library' = typical.lib
rc:/> set_attribute is_default true [find /libraries -library_domain dom_2]
Info    : Default library domain has been set. [LBR-109]
        : Default domain changed from: /libraries/library_domains/dom_1 to /
libraries/library_domains/dom_2

  Setting attribute of library_domain dom_2: 'is_default' = true
rc:/> rm [find /libraries -library_domain dom_1]
```

- You can only remove the default library domain if it is the only library domain that remains. If you remove the default library domain and a design was loaded, all instances become unresolved. In that case, none of the instances have timing, power or area information.

Removing Level Shifter Instances

If you reset the target library domain for a portion of the design, you should also remove the level shifters going to or from that portion of the design.

➤ To remove a level-shifter hiearchical instance, use

```
level_shifter remove [find / -instance ls_instance]
```

Saving Information

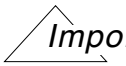
➤ To save the information for a later synthesis session, use the following commands:

```
write_hdl > design.v
write_script > design.scr
write_sdc > design.sdc
```

These commands will save MSV-specific information such as library domain-associations with portions of the design.

What If Analysis

You can check the effect on timing and power by using different library domains for some portions of the designs. The what-if analysis can be done before or after mapping, although you can get more meaningful results when you perform it after mapping



To do a what-if analysis, RTL Compiler expects that the libraries in the library domain you want to switch to, have the same set of cells with the same names as the libraries in the original library domain.

When you change the library domain assignment of a subdesign, RTL Compiler tries to rebind each cell in the subdesign by searching for a cell with the same name in a library in the new library domain. If RTL Compiler finds such cell, it uses the timing and power information of this cell from the new library domain to perform timing and power analysis and optimization. Otherwise, the original instance becomes an unresolved instance with the library cell name as its subdesign name.

What-If Analysis before Mapping Flow

1. Setup.
2. Create library domains.
3. Load libraries
4. Read netlist.
5. Elaborate design.
6. Set constraints.
7. Set optimization directives.
8. Assign library domains to portions of design.
9. Report timing and power.

What-if Analysis Steps

10. Reassign the target library domain for a portion of the design.
11. Report on timing and power.
12. Repeat steps 10 through 11 until you are satisfied with results.

13. Map design.
14. Continue flow.

What-If Analysis after Mapping Flow

1. Setup.
2. Create library domains.
3. Load libraries
4. Read netlist.
5. Elaborate design.
6. Set constraints.
7. Set optimization directives.
8. Assign library domains to portions of design.
9. Map design.
10. Report timing and power.

What-if Analysis Steps

11. Save information (see Saving Information).
12. Reassign the target library domain for a portion of the design.
13. Either remap or do incremental synthesis.
14. Report on timing and power.
15. Repeat steps 11 through 14 until you are satisfied with results.
16. Restore information depending on the what-if results.

17. Insert level shifters.
18. Continue flow.

Other Flows

MSV Flow when Starting from Structural Netlist

In case you start the MSV flow with a structural netlist, follow these steps:

1. Setup.

   ```
   set_attribute lib_search_path ... /
   set_attribute hdl_search_path ... /
   ...
   ```

2. Create library domains.

   ```
   create_library_domain domain_list
   ```

3. Specify the target libraries for each library domain.

   ```
   set_attribute library library_list [find /libraries -library_domain domain] /
   ```

4. Read netlist saved in previous session.

   ```
   read_hdl -struct design.v
   ```

5. Elaborate the design.

   During elaboration of a structural netlist, RTL Compiler will link the cells in the structural netlist to library cells of the *default* library domain.

   Note: You might get unresolved reference warnings if cells present in the netlist are not present in the default library domain. However, after binding to the appropriate library domains (done when reading in the script in the next step), a search for unresolved should return empty.

6. Read in script saved in previous session (see Saving Information)

   ```
   include design.scr
   ```

7. Read in modified SDC constraints.

8. Continue flow.

MSV with DFT Flow

1. Setup.

   ```
   set_attribute lib_search_path ... /
   set_attribute hdl_search_path ... /
   ...
   ```

2. Create library domains.

   ```
   create_library_domain domain_list
   ```

3. Specify the target libraries for each library domain.

   ```
   set_attribute library library_list [find /libraries -library_domain domain] /
   ```

4. Read netlist.

   ```
   read_hdl hdl_files
   ```

5. Elaborate design.

   ```
   elaborate
   ```

6. Set constraints.

7. Set optimization directives.

8. Set target library domain for top design

   ```
   set_attribute library_domain library_domain design
   ```

9. Set target library domain for blocks.

   ```
   edit_netlist uniquify subdesign
   set_attribute library_domain library_domain subdesign
   ```

10. Specify the DFT setup and run the DFT rule checker.

    ```
    define_dft shift_enable ...
    define_dft test_mode ... (design dependent)
    check_dft_rules
    report dft_registers
    ```

11. (Optional) Fix DFT rule violations.

    ```
    fix_dft_violations ....
    ```

12. (Optional) Insert testability logic.

    ```
    insert_dft shadow_logic ....
    insert_dft test_point ...
    ```

13. Synthesize and map to scan.

    ```
    synthesize -to_mapped ...
    ```

14. Configure and connect top-level chains.

```
define_dft scan_segment -name segment_name ...
define_dft scan_chain ...
connect_scan_chains -preview -auto_create_chains
connect_scan_chains -auto_create_chains
report dft_chains > DFTchains
```

15. Insert level shifters.

```
define_level_shifter_group -from_library_domain library_domain \
-to_library_domain library_domain [-name string] \
-libcells cell_list
level_shifter insert [-from_library_domain library_domain] \
[-to_library_domain library_domain] [-location {from | to}] \
[-dedicate_level_shifter libcell]
report level_shifter -hier -detail
```

16. To fix any timing issues, run incremental optimization.

```
synthesize -incremental
```

17. Analyze design.

```
report timing
report area
report dft_setup
```

18. Export the design.

```
write_fe_msv
```

MSV with Power Flow

1. Setup

```
set_attribute lib_search_path ... /
set_attribute hdl_search_path ... /
...
```

2. Create library domains.

```
create_library_domain domain_list
```

3. Specify the target libraries for each library domain.

```
set_attribute library library_list [find /libraries -library_domain domain] /
```

4. Enable clock gating.

```
set_attribute lp_insert_clock_gating true /
set_attr lp_clock_gating_prefix string /
```

5. Enable RTL power analysis.

```
set_attr hdl_track_filename_row_col true /
```

6. Load and elaborate the design.

```
read_hdl hdl_files
elaborate
```

7. Set target library domain for top design and for blocks:

```
set_attribute library_domain library_domain design
# See Associate Library Domains with Different Blocks of Design
edit_netlist uniquify subdesign
set_attribute library_domain library_domain subdesign
```

Tip

To perform multi-Vth leakage power optimization, you can read in multiple Vth libraries per library domain.

8. Estimate RTL power.

```
set_attribute lp_power_unit mW /
report power -rtl -detail
```

9. Set constraints.

```
read_sdc sdc_file
report timing -lint
```

10. Specify the following constraints per library domain.

```
set_attr operating_conditions string [find /libraries -library_domain domain]
set_attr wireload_selection string [find /libraries -library_domain domain]
```

11. Set optimization directives.

12. Set clock-gating directives

```
set_attribute lp_clock_gating_cell cg_libcell_name /
set_attribute lp_clock_gating_add_reset
set_attribute lp_clock_gating_style
```

13. Set power optimization directives

```
set_attribute lp_power_analysis_effort medium /
```

To enable the use of state retention power-gating (SRPG) flops, set the following attributes:

```
set_attribute lp_map_to_srpg_cells true /designs/design
set_attribute lp_map_to_srpg_cells true instance
set_attribute lp_srpg_pg_driver string /designs/design
```

To only optimize leakage power, set the following attributes:

```
set_attribute max_leakage_power integer [find / -des *]
# the next atribute should only be set if you read in multiple vth libraries
set_attribute lp_multi_vt_optimization_effort medium /designs/design
```

To only optimize dynamic power, set the following attributes:

```
set_attribute max_dynamic_power integer [find / -des *]
```

To optimize both leakage and dynamic power, but optimize dynamic power first, set the following attributes:

```
set_attribute max_leakage_power integer [find / -des *]
set_attribute max_dynamic_power integer [find / -des *]
set_attribute lp_optimize_dynamic_power_first true /designs/design
# set_attribute lp_power_optimization_weight float /designs/design
```

14. Annotate switching activities.

```
set_attribute lp_toggle_rate_unit /ns /
read_tcf [-scale scale_factor] tcf_file_for_RTL
```

15. Synthesize the design.

```
synthesize -to_mapped ...
```

16. Insert level shifters

```
define_level_shifter_group -from_library_domain library_domain \
-to_library_domain library_domain [-name string] -libcells cell_list
```

```
level_shifter insert [-from_library_domain library_domain] \
[-to_library_domain library_domain] [-location {from | to}] \
[-dedicate_level_shifter libcell]
```

```
report level_shifter -hier -detail
```

17. Annotate switching activities.

```
read_tcf tcf_file_for_gate_netlist
```

18. Analyze design.

```
report power
report timing
report area
```

```
report gates -power
report clock_gating
```

19. Export the design.

```
write_fe_msv
```

Using Library Domains for Non-MSV Design

The MSV flow can also be used for designs for which you want to use specific libraries for some of the blocks, or for designs for which you want to use different configurations of the same library for different blocks in the same design.

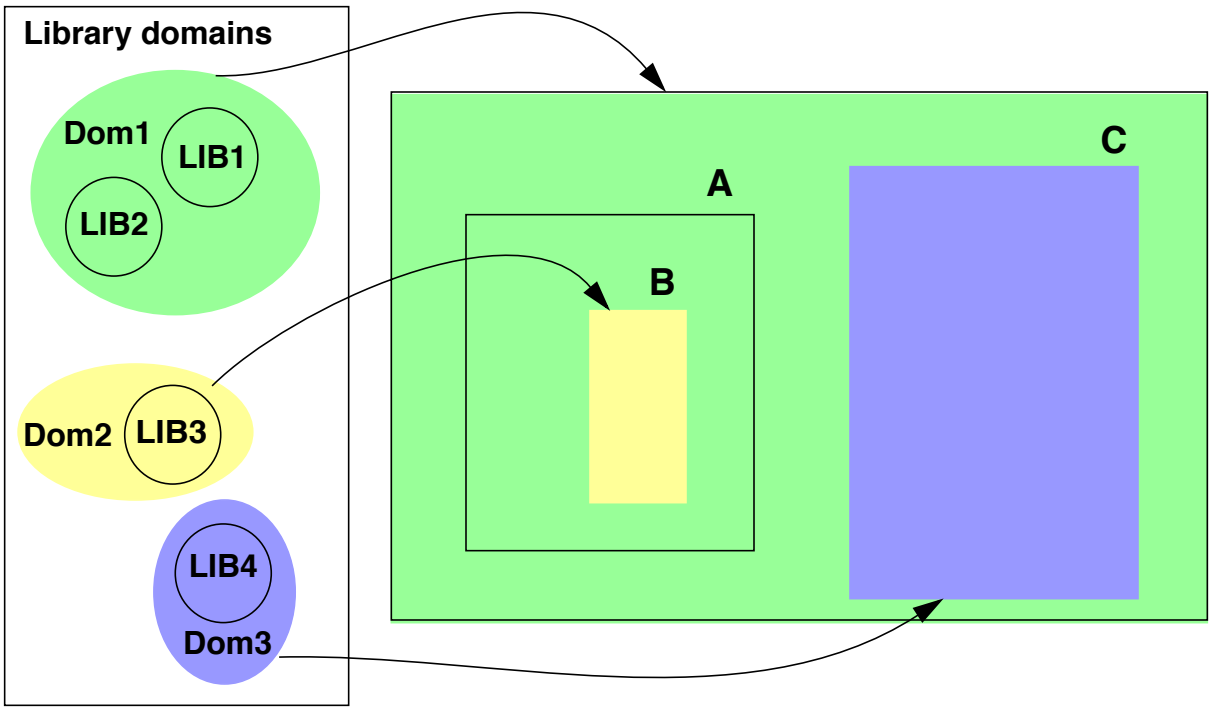
- Associate Dedicated Libraries with Different Portions of the Design
- Target Different Cells from Same Library for Different Portions of Design

Note: Because the design uses only a single supply voltage, there is no need for level-shifter insertion in this flow.

Associate Dedicated Libraries with Different Portions of the Design

For example, the design in Figure 14-10 uses three different library sets. You want to use libraries `LIB1` and `LIB2` for the top-level and for block A, library `LIB3` for block B and library `LIB4` for block C.

Figure 14-10 Use of Dedicated Libraries in Single Supply Voltage Design



1. Begin setup.
2. Create as many library domains as there are portions in the design for which you want to use dedicated libraries.

   In the example of Figure 14-10, you want to use three different sets of libraries, which requires you to create three library domains.
3. Read in the library (libraries) for each library domain.
4. Read the HDL files.
5. Elaborate the design.
6. Set the constraints.
7. Associate library domains with the design and blocks for which you want to use the dedicated cells.

   `set_attribute library_domain` *`library_domain`* {*`design`*|*`subdesign`*}

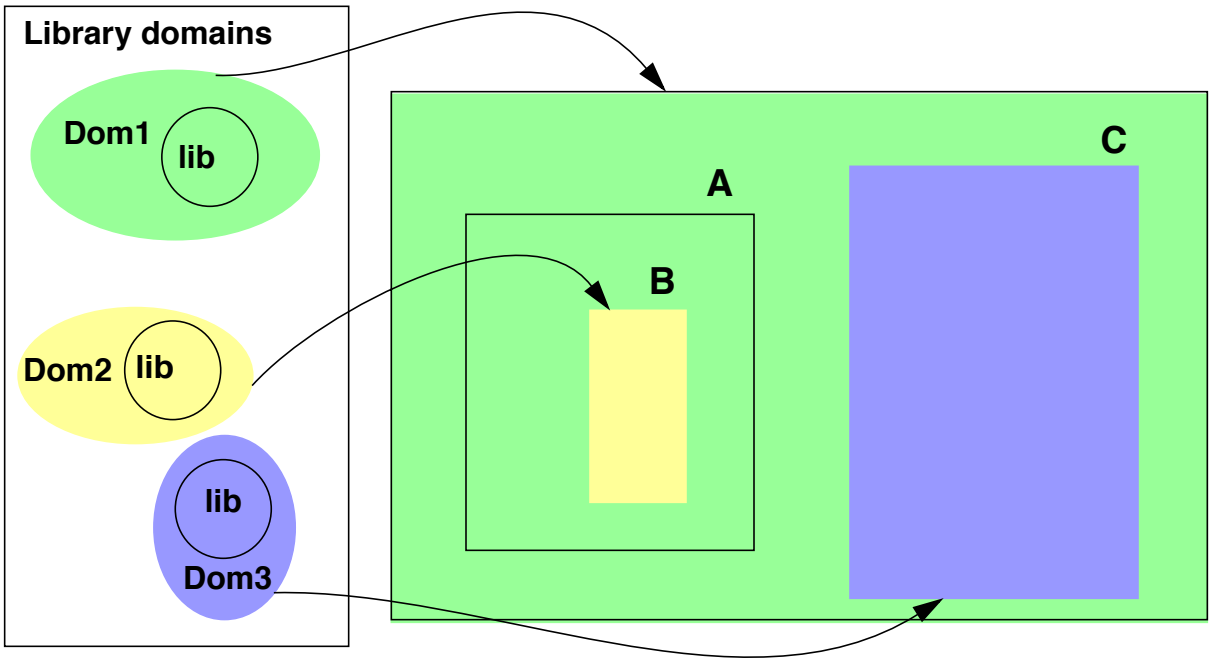
   For more information, refer to Associate Library Domains with Different Blocks of Design.
8. Synthesize the design.

   The different portions of the design that are associated with different library domains will be mapped to the target libraries of those library domains and optimized.
9. Analyze the design.

Target Different Cells from Same Library for Different Portions of Design

For example, the design in Figure 14-11 uses the same library for the entire design, but you would like to use a limited set of cells to map block B and a different set of cells to map block C, and you allow the use of all cells for the top-level and block A.

Figure 14-11 Use of Targeted Cells



1. Begin setup.
2. Create the same number of library domains as number of library cell sets that you want to use.

   In the example above you want to use three different cell sets, so you need to create three library domains.
3. Read in the library (libraries) for each library domain.

```
set_attribute library lib [find /libraries -library_domain dom1]
set_attribute library lib [find /libraries -library_domain dom2]
set_attribute library lib [find /libraries -library_domain dom3]
```

4. Exclude library cells.

   Because library `lib` is used in three library domains, RTL Compiler treats it as if three different libraries were read in. This allows to exclude cells in one library domain, while you allow the use of them in a different library domain.

Within each library domain, you can exclude the library cells that the mapper should not use. To exclude a cell you can use the `avoid` attribute.

```
set_attribute avoid true /libraries/library_domains/dom2/lib/libcells/AO22X2M
set_attribute avoid true [find /*/dom2/lib -libcell AO22X2M]
```

5. Read the HDL files
6. Elaborate the design
7. Set the Constraints
8. Associate library domains with the design and blocks for which you want to use the dedicated cells.

   ```
   set_attribute library_domain library_domain {design|subdesign}
   ```

   For more information, refer to Associate Library Domains with Different Blocks of Design.
9. Synthesize the design.

   The different portions of the design that are associated with different library domains will be mapped to the targeted cells of those library domains and optimized.
10. Analyze the design.

A

Simple Synthesis Template

The following script is a simple script which delineates the very basic RTL Compiler flow.

```
# ******************************************************
# *
# * A very simple script that shows the basic RTL Compiler flow
# *
# ******************************************************

set_attribute lib_search_path <full_path_of_technology_library_directory> /
set_attribute hdl_search_path <full_path_of_hdl_files_directory> /

set_attribute library <technology_library> /
read_hdl <hdl_file_names>

elaborate <top_level_design_name>

set clock [define_clock –period <periodicity> –name <clock_name> [clock_ports]]
external_delay –input <specify_input_external_delay_on_clock>
external_delay –output <specify_output_external_delay_on_clock>

synthesize -to_mapped

report timing >  <specify_timing_report_file_name>
report area   > <specify_area_report_file_name>

write –mapped > <specify_netlist_name>
write_script > <script_file_name>

quit
```

B

Encrypting Libraries

To protect proprietary data, you can encrypt the ASCII library files. Use the `lib_encrypt` utility to perform the encryption. The `lib_encrypt` utility is installed along with the Encounter software. To encrypt the ASCII library file, use the following command:

```
lib_encrypt [-ogz] [-help] in_file out_file
```

Options and Arguments

`-help`	Displays the syntax of the `lib_encrypt` command.
`in_file`	Specifies the name of library file to be encrypted.
`-ogz`	Creates a gzip file of the encrypted output library file.
`out_file`	Specifies the name of the output file.

To check the .lib technology library files for any errors or compatibility issues, use the `check_library` command within the rcl environment. The `rcl` command invokes the rcl environment. The syntax of the `rcl` command is:

```
rcl [-no_custom] [-files file]
```

Options and Arguments

`-no_custom`	Specifies to read only the master `.synth_init` file, located in the installation directory. By default, RTL Compiler also loads the initialization file in your home directory and in your current design directory
`-files`	Specifies the name of a script (or command file) to execute.

Specify the libraries to check with the `check_library` command from within the rcl environment. The `lib_encrypt` format is supported. The following example illustrates how to check the two libraries named `a.lib` and `b.lib`, starting from the unix environment:

```
unix> rcl
rcl:/> check_library { a.lib b.lib }
```

Index

Symbols

A

B

C

D

E

F

G

H

I

L

T

U

V

W