Contents

Preface ........................................................................................................................................... 7
About This Manual ........................................................................................................................... 8
Additional References ....................................................................................................................... 8
How to Use the Documentation Set .................................................................................................. 9
Reporting Problems or Errors in Manuals ......................................................................................... 10
Customer Support .......................................................................................................................... 10
  SourceLink Online Customer Support ......................................................................................... 10
  Other Support Offerings ............................................................................................................... 10
Messages ......................................................................................................................................... 11
Man Pages ....................................................................................................................................... 12
Command-Line Help ........................................................................................................................ 12
  Getting the Syntax for a Command ............................................................................................... 12
  Getting the Syntax for an Attribute ............................................................................................. 13
  Searching for Attributes .............................................................................................................. 13
  Searching For Commands When You Are Unsue of the Name .................................................... 13
Documentation Conventions ......................................................................................................... 14
  Text Command Syntax ............................................................................................................... 14

1 Getting Started with the Generic Flow .......................................................................................... 17
Overview ......................................................................................................................................... 18
Tasks ............................................................................................................................................... 19
  Starting RTL Compiler ............................................................................................................... 20
  Generating Log Files ................................................................................................................... 22
  Generating the Command File ...................................................................................................... 22
  Setting Information Level and Messages ..................................................................................... 23
  Specifying Explicit Search Paths ............................................................................................... 23
  Setting the Target Technology Library ...................................................................................... 23
  Setting the Appropriate Synthesis Mode .................................................................................... 24
  Loading the HDL Files ............................................................................................................... 24
  Performing Elaboration ............................................................................................................... 25
Encounter RTL Compiler Synthesis Flows

Applying Constraints ......................................................... 25
Applying Optimization Constraints ......................................... 27
Performing Synthesis .......................................................... 27
Analyzing the Synthesis Results ............................................. 28
Writing Out Files for Place and Route ...................................... 28
Exiting RTL Compiler ............................................................ 29
Summarizing the Flows .......................................................... 29

2
Physically Aware Flow ......................................................... 31
Overview ............................................................................. 32
Tasks ................................................................................. 33
 Setting the LEF Files ......................................................... 33
 Setting the Capacitance Information ...................................... 34
 Consistency Checks Between the LEF and Capacitance Table File .................................................................. 35
 Setting the Appropriate Synthesis Mode .................................. 36
 Analyzing the Results .......................................................... 36
 Summarizing the Flows .......................................................... 36

3
Physically Aware Flow with QoS Prediction .............................. 39
Overview ............................................................................. 40
Tasks ................................................................................. 42
 Importing LEF Files .......................................................... 42
 Specifying Capacitance Information ...................................... 44
 Consistency Checks Between the LEF and Capacitance Table File .................................................................. 45
 Loading the Encounter Configuration File ................................. 45
 Importing DEF Files .......................................................... 46
 Synthesizing in the Physical Flow .......................................... 47
 Estimating and Optimizing for Silicon .................................... 47
 Analyzing the Results .......................................................... 47
 Exporting Files for Place and Route ........................................ 48
 Summarizing the Flows .......................................................... 50
4
Design For Manufacturing Flow ........................................... 53

Overview ............................................................................. 54

Tasks .................................................................................. 55
  Specifying the Yield Coefficients Information ..................... 55
  Setting the DFM Flow ......................................................... 55
  Analyzing the Yield Information ....................................... 55
  Summarizing the Flows ..................................................... 57
Preface

This preface contains the following sections:

- About This Manual on page 8
- Additional References on page 8
- How to Use the Documentation Set on page 9
- Customer Support on page 10
- Messages on page 11
- Man Pages on page 12
- Command-Line Help on page 12
- Documentation Conventions on page 14
About This Manual

Additional References

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


- 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)


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

Installation and Configuration
- Cadence Installation Guide
- Cadence License Manager
- README File

New Features and Solutions to Problems
- README File
- What's New in Encounter RTL Compiler
- Known Problems and Solutions in Encounter RTL Compiler

Tasks and Concepts
- Using Encounter RTL Compiler
- HDL Modeling in Encounter RTL Compiler
- Setting Constraints and Performing Timing Analysis in Encounter RTL Compiler
- Datapath Synthesis in Encounter RTL Compiler
- Library Guide for Encounter RTL Compiler
- Low Power in Encounter RTL Compiler
- Design for Test in Encounter RTL Compiler

References
- Attribute Reference for Encounter RTL Compiler
- Command Reference for Encounter RTL Compiler
- ChipWare in Encounter RTL Compiler
- GUI Guide for Encounter RTL Compiler
- Quick Reference for Encounter RTL Compiler
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:


Other Support Offerings

- **Support 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:
   
   ```bash
   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:

   ```bash
   man check_dft_rules
   man cell_leakage_power
   ```

You can also use the `more` command, which behaves like its UNIX counterpart. If the output of a manpage is too small to be displayed completely on the screen, use the `more` command to break up the output. Use the spacebar to page forward, like the UNIX `more` command.

```bash
rc:/> more report timing
```

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
```

You can type a sequence of letters after the set_attribute command and press Tab to get a list of all attributes that contain those letters.

```
rc:/> set_attr li
```

ambiguous "li": lib_lef_consistency_check_enable lib_search_path libcell liberty_attributes libpin library library_domain line_number

**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.
Getting Started with the Generic Flow

- Overview on page 18
- Tasks on page 19
  - Starting RTL Compiler on page 20
  - Generating Log Files on page 22
  - Generating the Command File on page 22
  - Setting Information Level and Messages on page 23
  - Specifying Explicit Search Paths on page 23
  - Setting the Target Technology Library on page 23
  - Setting the Appropriate Synthesis Mode on page 24
  - Loading the HDL Files on page 24
  - Performing Elaboration on page 25
  - Applying Constraints on page 25
  - Applying Optimization Constraints on page 27
  - Performing Synthesis on page 27
  - Analyzing the Synthesis Results on page 28
  - Writing Out Files for Place and Route on page 28
  - Exiting RTL Compiler on page 29
  - Summarizing the Flows on page 29
Overview

Figure 1-1 on page 18 shows the generic RTL Compiler work flow. The term “generic” merely illustrates that whatever flow you use, you will most likely use most or all of the steps in the generic flow. This section briefly and sequentially describes all the tasks within the work flow.

Figure 1-1 Generic RTL Compiler Work Flow
Tasks

- Starting RTL Compiler on page 20
- Generating Log Files on page 22
- Generating the Command File on page 22
- Setting Information Level and Messages on page 23
- Specifying Explicit Search Paths on page 23
- Setting the Target Technology Library on page 23
- Setting the Appropriate Synthesis Mode on page 24
- Loading the HDL Files on page 24
- Performing Elaboration on page 25
- Applying Constraints on page 25
- Applying Optimization Constraints on page 27
- Performing Synthesis on page 27
- Analyzing the Synthesis Results on page 28
- Writing Out Files for Place and Route on page 28
- Exiting RTL Compiler on page 29
- Summarizing the Flows on page 29
Starting RTL Compiler

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

```
rc [-32] [-64] [-cmdfile string] [-execute command] [-files script_file]
    [-logfile log_file_name] [-lsf_cpu integer] [lsf_queue queue_name] [-no_custom]
    [-unique] [-nologfile] [queue] [-use_license RTL_Compiler_L | RTL_Compiler_Ultra
    RTL_Compiler_Verification | First_Encounter_GXL | SOC_Encounter_GXL | FE_GPS |
    SOC_Encounter_GPS | Virtuoso_Digital_Implement] [-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 (regardless of the platform):

  ```
  unix> rc
  ```

- You can specify the logfile name with the `-logfile` option. The default name is `rc.log` if there is no other logfile in the directory from which RTL Compiler is launched.

  ```
  unix> rc -logfile pov.log
  ```

  Do not use the UNIX `tee` command and pipe (`|`) to specify your logname: doing so would not allow you to use the `control-c` key sequence to gracefully exit a process like incremental optimization.

- 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.

**Note:** You can set the `CDS_AUTO_64BIT` environment variable to `ALL` (```
(setenv CDS_AUTO_64BIT ALL)```) to launch not only RTL Compiler, but all Cadence
tools in 64 bit. You will not need to specify the \texttt{-64} option if you use this variable.

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

\begin{verbatim}
unix> rc < script_file_name &
\end{verbatim}

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

\begin{verbatim}
unix> rc -files script_file_name < /dev/null
\end{verbatim}

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

\begin{verbatim}
unix> rc -execute "set_attribute script_search_path pathname"
\end{verbatim}

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

\begin{verbatim}
unix> rc -files script_file_name -execute "set variable_name value"
\end{verbatim}

➤ RTL Compiler supports super-threading on LSF. Use the \texttt{-lsf_cpus} option to specify the number of processes to send to LSF and the \texttt{-lsf_queue} option to specify a particular LSF queue:

\begin{verbatim}
unix> rc -lsf_cpus 4 -lsf_queue teagan_queue
\end{verbatim}

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

\begin{verbatim}
unix> rc -lsf_cpus 4 -lsf_queue teagan_queue
...
rc:// set_attribute super_thread_servers {lsf lsf}
rc:// set_attribute bsub_options stormy_queue
\end{verbatim}

\textbf{Tip}

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

\begin{verbatim}
unix> rc -ver
\end{verbatim}

Just using \texttt{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 23.

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

➤ If a log file already exists, the new log file will have the number “1” appended to it.

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

If a command file already exists, the new command file will have the number “1” appended to it.
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.

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:

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

➤ To specify a single library compressed with gzip:

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

➤ To append libraries:

```bash
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.

### Setting the Appropriate Synthesis Mode

RTL Compiler has two modes: `wireload` and `ple`. These modes are set using the `interconnect_mode` attribute. The default mode is `wireload`. In `wireload` mode, you use wire-load models to drive synthesis. In `ple` mode, you use Physical Layout Estimators (PLE) to drive synthesis. PLE is the process of using physical information, such as LEF libraries, to provide better closure with back-end tools.

- To synthesize in `wireload` mode, do nothing: `wireload` mode is the default mode.

### 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:
  ```bash
  read_hdl file1.v
  read_hdl file2.v
  read_hdl file3.v
  ```

- Or you can load the files simultaneously:
  ```bash
  read_hdl { file1.v file2.v file3.v }
  ```

**Caution**

*Your files may have extra, hidden characters (e.g. line terminators) if they are transferred from Windows/Dos to the dos2unix UNIX environment. Be sure to eliminate them because RTL Compiler will issue an error when it encounters these characters.*
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 *Elaborating the Design*.

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.

*Setting Constraints and Performing Timing Analysis in Encounter RTL Compiler* gives a broader overview on constraints.
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.

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 Performing Synthesis.

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* 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*.

**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*. 
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.

### Summarizing the Flows

The following example show broad overviews of each of the flows. The first example is the flow discussed in this chapter. The subsequent examples are the flows discussed in the other chapters.

**Example 1-1  Generic Flow**

```
set_attribute library library_name
read_hdl filename
elaborate
read constraints
synthesize -to_mapped
write_hdl
```

**Example 1-2  Physical Flow without predict_qos**

```
set_attribute library library_name
set_attribute lef_library {tech.lef cell.lef}
set_attribute cap_table_file cap_file
read_hdl filename
elaborate
read sdc
synthesize -to_mapped
write_hdl
```

**Example 1-3  Physical Flow with predict_qos, LEF, and Capacitance Table**

```
set_attribute library library_name
set_attribute lef_library {tech.lef cell.lef}
```
set_attribute cap_table_file cap_file
read_hdl filename
elaborate
read sdc
read_def def_file
synthesize -to_mapped
predict_qos
write_encounter design

Example 1-4 Physical Flow with predict_qos and Encounter Configuration File
read_encounter config config_file
read_def def_file
synthesize -to_mapped
predict_qos
write_encounter design
Physically Aware Flow

- Overview on page 32
- Tasks on page 33
  - Setting the LEF Files on page 33
  - Setting the Capacitance Information on page 34
  - Consistency Checks Between the LEF and Capacitance Table File on page 35
  - Setting the Appropriate Synthesis Mode on page 36
  - Analyzing the Results on page 36
  - Summarizing the Flows on page 36
Overview

This chapter describes the physically aware flow. It is brief and does not differ much from the generic flow except that you will be using LEF files and capacitance tables to drive synthesis. Any steps that overlap with the generic flow will not be covered in this chapter. Refer to Getting Started with the Generic Flow on page 17 for more information on the generic flow.

Figure 2-1 Physically Aware Flow
**Tasks**

The tasks below list only those that are different from the generic flow or illustrate a new step.

- Setting the LEF Files on page 33
- Setting the Capacitance Information on page 34
- Setting the Appropriate Synthesis Mode on page 36
- Summarizing the Flows on page 36

**Setting the LEF Files**

LEF files are required for any physical flow. LEF files are ASCII files that contain library cell physical information such as layer, via, placement site type, routing design rules, process information, standard cell definitions, and macro cell definitions. This information is usually divided across separate LEF files for easier manageability.

When you read in LEF libraries, you will automatically be switched to `ple` mode. By default, you will be in the `wireload` mode. If you want to be in `wireload` mode after loading the LEF libraries, you must manually set the `interconnect_mode` attribute back to `wireload`.

```
rc:/> set_attribute interconnect_mode wireload /
```

To import LEF files, use the `lef_library` attribute. Specify all LEF files, not just the technology LEF or the cell library LEF. The following example imports a technology and cell library LEF files. It does not matter in which order you specify the LEF files.

```
rc:/> set_attribute lef_library {tech.lef cell.lef}
```

The following example differs from the above example: it replaces the existing LEF file because it specifies the files separately with two `set_attribute` commands as opposed to a Tcl list with one `set_attribute` command.

```
rc:/> set_attribute lef_library tech.lef
rc:/> set_attribute lef_library cell.lef
```

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

```
rc:/> get_attribute lef_library
tech.lef
cell.lef
```

RTL Compiler will check whether the following definitions are in the LEF file:

- `CAPACITANCE CPERSQ`
If any of these definitions are missing, RTL Compiler will issue a warning message.

If there is at least one MACRO definition in the LEF file, RTL Compiler checks if all the cells in the timing library have a corresponding definition in the LEF library. Any cells that are defined in the timing library but not in the LEF will be marked as avoid (they will not be used during synthesis) and a warning message will be issued. To turn off this consistency checking, set the lib_lef_consistency_check_enable attribute to false:

rc:/> set_attribute lib_lef_consistency_check_enable false /

If you do not have LEF files but wish to synthesize in ple mode, specify the appropriate values for the resistance_per_unit_len and capacitance_per_unit_len attributes. After these values are specified, set the interconnect_mode attribute to ple. For example:

rc:/> set_attribute capacitance_per_unit_len .002
rc:/> set_attribute resistance_per_unit_len .39
rc:/> set_attribute interconnect_mode ple

The resistance and capacitance information can also be found in the capacitance table file.

RTL Compiler supports LEF 5.3 and above. Refer to the LEF/DEF Language Reference for more information on LEF files.

**Setting the Capacitance Information**

Capacitance tables files contain the same type of information as LEF files but the values are different. The capacitance in the capacitance table is almost always much better than it is in the LEF file. The granularity is also much finer. The capacitance in a LEF comes from a foundry and is generated by whatever process it sees as appropriate. The capacitance info in a capacitance table comes from the same process definition files that drive sign off extraction as well as the various other extractors used in Cadence tools. The process definition files define layer thicknesses, compositions, and spacings so there is no mystery as to from where the values in a capacitance table have come.

To load the capacitance information, use the cap_table_file attribute:

rc:/> set_attribute cap_table_file avy.cap
Encounter RTL Compiler Synthesis Flows
Physically Aware Flow

You should specify both LEF and capacitance table files. However, you can specify only the
LEF files if the capacitance table files are not available.

Scaling factors are used to align a design with a particular process. A capacitance table is
process specific where as a scaling factor is design specific. The scaling factors are provided
to be consistent with Encounter. Only use a scaling factor if it will also be used in the back-end.

RTL Compiler will check if the following definitions are in the capacitance table file:
- PROCESS_VARIATION
- BASIC_CAP_TABLE
- width
- Cc
- Carea
- Cfrg

If any of these definitions are missing, RTL Compiler will issue a warning message. It will
purposely disregard the EXTENDED_CAP_TABLE section because the PLE is intended to
synchronize with a view of the design where fast extractors are typically used.

If you have neither the capacitance nor the LEF files but wish to synthesize in ple mode,
specify the appropriate values for the resistance_per_unit_len and
capacitance_per_unit_len attributes. After these values are specified, set the
interconnect_mode attribute to ple. For example:
rc:/> set_attribute capacitance_per_unit_len .002
rc:/> set_attribute resistance_per_unit_len .39
rc:/> set_attribute interconnect_mode ple

The resistance and capacitance information can be found in the capacitance table file.

Consistency Checks Between the LEF and Capacitance Table File

After you load both your LEF and capacitance table files, RTL Compiler will perform
consistency checks between the two files. This happens automatically, much like the check
between the LEF and timing library files.

- Number of Layers — RTL Compiler will check to determine if the number of layers
defined in the LEF and the capacitance table files are equal.
If the LEF has more layers than the capacitance table, then an error message will be issued and you will need to manually check both of the files to resolve the inconsistency.

If the capacitance table has more layers than the LEF, a warning message will be issued and the number of routing layers will be set to the number specified in the capacitance table.

- **Width of Layers** — RTL Compiler will check to determine if the width of the layers defined in the LEF and the capacitance table files are equal. A warning will only be issued if the width difference defined in the two files is greater than 10%.

**Setting the Appropriate Synthesis Mode**

RTL Compiler has two modes: **wireload** and **ple**. These modes are set using the `interconnect_mode` attribute. The default mode is **wireload**. In **wireload** mode, you use wire-load models to drive synthesis. In **ple** mode, you use Physical Layout Estimators (PLE) to drive synthesis. PLE is the process of using physical information, such as LEF libraries, to provide better closure with back-end tools.

**Analyzing the Results**

After synthesis and optimization, you can view the LEF denominated results. If you do not use the `-physical` options of the `report` commands, the measured units will be based on the timing library (.lib). The following command reports the LEF denominated area results:

```plaintext
cr:/> report area -physical
===============================================================================
... Module: controller
Technology library: grey 1.2
Operating conditions: slow
Interconnect mode: ple
===============================================================================
Instance Cells Cell Area Net Area
-------------------------------
controller 34 694 102
```

**Summarizing the Flows**

The following example show broad overviews of each of the flows. The first example is the flow discussed in this chapter. The subsequent examples are the flows discussed in the other chapters.
Example 2-1 Physical Flow without predict\_qos

```plaintext
set_attribute library library\_name
set_attribute lef_library {tech.lef cell.lef}
set_attribute cap_table_file cap\_file
read_hdl filename
elaborate
read sdc
synthesize -to_mapped
write_hdl
```

Example 2-2 Physical Flow with predict\_qos, LEF, and Capacitance Table

```plaintext
set_attribute library library\_name
set_attribute lef_library {tech.lef cell.lef}
set_attribute cap_table_file cap\_file
read_hdl filename
elaborate
read sdc
read_def def\_file
synthesize -to_mapped
predict_qos
write_encounter design
```

Example 2-3 Physical Flow with predict\_qos and Encounter Configuration File

```plaintext
read_encounter config config\_file
read_def def\_file
synthesize -to_mapped
predict_qos
write_encounter design
```

Example 2-4 Generic Flow

```plaintext
set_attribute library library\_name
read_hdl filename
elaborate
read constraints
synthesize -to_mapped
write_hdl
```
Physically Aware Flow with QoS Prediction

- **Overview** on page 40
- **Tasks** on page 42
  - Importing LEF Files on page 42
  - Specifying Capacitance Information on page 44
  - Consistency Checks Between the LEF and Capacitance Table File on page 45
  - Loading the Encounter Configuration File on page 45
  - Importing DEF Files on page 46
  - Synthesizing in the Physical Flow on page 47
  - Estimating and Optimizing for Silicon on page 47
  - Analyzing the Results on page 47
  - Exporting Files for Place and Route on page 48
  - Summarizing the Flows on page 50
Overview

The physically aware flow with quality of silicon (QoS) prediction allows you to use physical information to drive RTL Compiler’s global focused synthesis. You still use the standard synthesis inputs (RTL or netlist, .lib, .sdc) and outputs (netlist, .sdc). However, you can also use LEF libraries, capacitance table files, and DEF files (which specify the floorplan) as the incremental inputs. The ability to consider and utilize information from LEF and DEF files during synthesis ensures both accuracy and predictable closure with back-end tools. The use of physical layout estimators (PLE) eliminate the traditional need for wire-load models.

Specifically, the physical flow will:

■ Use physical process information along with areas 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

■ Use a SiliconVirtual Prototype to predict the physical effects on the quality of silicon

Commands and attributes access these physical dynamics of the tool.
Encounter RTL Compiler Synthesis Flows  
Physically Aware Flow with QoS Prediction

HDL files, netlist, timing libraries, LEF, DEF, .conf

Set timing libraries
Set LEF libraries
Set capacitance table
Load HDL files or Netlist
Perform elaboration

Modify constraints

Apply physical constraints / DEF file

Modify source

Apply constraints
Synthesize

Estimate and optimize for silicon with predict_qos
Analyze
Incremental optimization
Update physical estimation with predict_qos
Export design

Netlist, SDC, DEF, .conf

Meet constraints?  No
Yes

Required Task for Physical Flow with predict_qos
Tasks

The physical flow is very similar to that of the “standard” synthesis flow. Therefore, you can perform the same tasks with the same command. For example, elaboration is always performed with the `elaborate` command and timing reports are always obtained with the `report timing` command.

The tasks below list only those that are different from the generic flow or illustrate a new step.

- Importing LEF Files on page 42
- Specifying Capacitance Information on page 44
- Loading the Encounter Configuration File on page 45
- Importing DEF Files on page 46
- Synthesizing in the Physical Flow on page 47
- Estimating and Optimizing for Silicon on page 47
- Exporting Files for Place and Route on page 48
- Summarizing the Flows on page 50

Importing LEF Files

LEF files are required for any physical flow. You will need LEF files for physical operations like optimizing for silicon with the `predict_qos` command. LEF files are ASCII files that contain library cell physical information such as layer, via, placement site type, routing design rules, process information, standard cell definitions, and macro cell definitions. This information is usually divided across separate LEF files for easier manageability.

When you read in LEF libraries, you will automatically be switched to `ple` mode. By default, you will be in the `wireload` mode. If you want to be in `wireload` mode after loading the LEF libraries, you must manually set the `interconnect_mode` attribute back to `wireload`.

```
rc:/> set_attribute interconnect_mode wireload /
```

To import LEF files, use the `lef_library` attribute. Specify all LEF files, not just the technology LEF or the cell library LEF. The following example imports a technology and cell library LEF files. It does not matter in which order you specify the LEF files.

```
rc:/> set_attribute lef_library {tech.lef cell.lef}
```
The following example differs from the above example: it replaces the existing LEF file because it specifies the files separately with two `set_attribute` commands as opposed to a Tcl list with one `set_attribute` command.

```plaintext
cr:/> set_attribute lef_library tech.lef
cr:/> set_attribute lef_library cell.lef
```

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

```plaintext
cr:/> get_attribute lef_library
    tech.lef
    cell.lef
```

RTL Compiler will check whether the following definitions are in the LEF file:

- CAPACITANCE CPERSQ
- EDGECAPACITANCE
- RESISTANCE RPERSQ
- SITE
- WIDTH

If any of these definitions are missing, RTL Compiler will issue a warning message.

If there is at least one MACRO definition in the LEF file, RTL Compiler checks if all the cells in the timing library have a corresponding definition in the LEF library. Any cells that are defined in the timing library but not in the LEF will be marked as `avoid` (they will not be used during synthesis) and a warning message will be issued. To turn off this consistency checking, set the `lib_lef_consistency_check_enable` attribute to `false`:

```plaintext
cr:/> set_attribute lib_lef_consistency_check_enable false /
```

If you do not have LEF files but wish to synthesize in `ple` mode, specify the appropriate values for the `resistance_per_unit_len` and `capacitance_per_unit_len` attributes. After these values are specified, set the `interconnect_mode` attribute to `ple`. For example:

```plaintext
rc:/> set_attribute capacitance_per_unit_len .002
rc:/> set_attribute resistance_per_unit_len .39
rc:/> set_attribute interconnect_mode ple
```

The resistance and capacitance information can also be found in the capacitance table file.

**Note:** RTL Compiler supports LEF 5.3 and above. Refer to the *LEF/DEF Language Reference* for more information on LEF files.
Specifying Capacitance Information

Capacitance tables files contain the same type of information as LEF files but the values are different. The capacitance in the capacitance table is almost always much better than it is in the LEF file. The granularity is also much finer. The capacitance in a LEF comes from a foundry and is generated by whatever process it sees as appropriate. The capacitance info in a capacitance table comes from the same process definition files that drive sign off extraction as well as the various other extractors used in Cadence tools. The process definition files define layer thicknesses, compositions, and spacings so there is no mystery as to from where the values in a capacitance table have come.

To load the capacitance information, use the `cap_table_file` attribute:

```
rc:/> set_attribute cap_table_file avy.cap
```

You should specify both LEF and capacitance table files. However, you can specify only the LEF files if the capacitance table files are not available.

Scaling factors are used to align a design with a particular process. A capacitance table is process specific where as a scaling factor is design specific. The scaling factors are provided to be consistent with Encounter. Only use a scaling factor if it will also be used in the back-end.

RTL Compiler will check if the following definitions are in the capacitance table file:

- `PROCESS_VARIATION`
- `BASIC_CAP_TABLE`
- `width`
- `Cc`
- `Carea`
- `Cfrg`

If any of these definitions are missing, RTL Compiler will issue a warning message. It will purposely disregard the `EXTENDED_CAP_TABLE` section because the PLE is intended to synchronize with a view of the design where fast extractors are typically used.

If you have neither the capacitance nor the LEF files but wish to synthesize in `ple` mode, specify the appropriate values for the `resistance_per_unit_len` and `capacitance_per_unit_len` attributes. After these values are specified, set the `interconnect_mode` attribute to `ple`. For example:

```
rc:/> set_attribute capacitance_per_unit_len .002
rc:/> set_attribute resistance_per_unit_len .39
```
rc:/> set_attribute interconnect_mode ple

The resistance and capacitance information can be found in the capacitance table file.

**Consistency Checks Between the LEF and Capacitance Table File**

After you load both your LEF and capacitance table files, RTL Compiler will perform consistency checks between the two files. This happens automatically, much like the check between the LEF and timing library files.

- **Number of Layers** — RTL Compiler will check to determine if the number of layers defined in the LEF and the capacitance table files are equal.

  If the LEF has more layers than the capacitance table, then an error message will be issued and you will need to manually check both of the files to resolve the inconsistency.

  If the capacitance table has more layers than the LEF, a warning message will be issued and the number of routing layers will be set to the number specified in the capacitance table.

- **Width of Layers** — RTL Compiler will check to determine if the width of the layers defined in the LEF and the capacitance table files are equal. A warning will only be issued if the width difference defined in the two files is greater than 10%.

**Loading the Encounter Configuration File**

The Encounter configuration file contains Tcl variables that describe design information such as the RTL or netlist, technology libraries, LEF information, constraints, capacitance tables, resistance scaling factors, capacitance scaling factors, and floorplan parameters.

Load the Encounter configuration file through the `read_encounter` command. If you load an Encounter configuration file, the only other input you need to load is the DEF file (for the floorplan).

rc:/> read_encounter config teagan.conf
Encounter RTL Compiler Synthesis Flows
Physically Aware Flow with QoS Prediction

Tip

If you load an Encounter configuration file, you do not need to load the timing library, LEF library, capacitance table file, RTL or netlist, and constraints.

```bash
set_attribute library
set_attribute lef_library
set_attribute cap_table_file
read_hdl
set SDC
read_encounter config
```

Importing DEF Files

DEF files are ASCII files that contain information that represent the design at any point during the layout process. DEF files can pass both logical information to and physical information from place-and-route tools.

- Logical information includes internal connectivity (represented by a netlist), grouping information, and physical constraints.
- Physical information includes the floorplan, placement locations and orientations, and routing geometry data.

RTL Compiler supports DEF 5.3 and above. Refer to the LEF/DEF Language Reference for more information on DEF files.

In RTL Compiler, the most common use for the DEF file will be to specify the floorplan and placement information. To import a DEF file, use the `read_def` command.

```bash
rc:/> read_def tutorial.def
```

RTL Compiler will perform a consistency check between the DEF and the Verilog netlist and issue relevant messages if necessary. The DEF file must define the die size. For better synthesis results, you should also have the pin, macro locations, and standard cell placement specified in the DEF, although it is not required. After reading the DEF the floorplan can be viewed in the GUI.

To check which DEF is loaded in the tool, use the `def_file` attribute:

```bash
rc:/> get_attribute def_file
```

**Note:** You must load the DEF file in this physical flow although it is optional in the “standard” flow. The DEF file must be loaded after elaboration.
Encounter RTL Compiler Synthesis Flows
Physically Aware Flow with QoS Prediction

Synthesizing in the Physical Flow

Like in the standard flow, you must synthesize your design with the `synthesize` command. There are no special arguments or options with the `synthesize` command in the physical flow. The operations performed in both flows are the same.

Estimating and Optimizing for Silicon

After synthesizing the design, you need to use the `predict_qos` command to analyze and optimize the design for silicon. The `predict_qos` command invokes Encounter. For example, RTL Compiler will use the SPEF file that Encounter generates during `predict_qos`. You will need an Encounter license to be available prior to the command’s execution.

**Note:** The `predict_qos` command requires an Encounter executable of version 5.2 or later. However, it is highly recommended that you use the same versions of Encounter and RTL Compiler. For example, if you are using RTL Compiler version 6.1, then use Encounter version 6.1 also.

The `predict_qos` command generates a Silicon Virtual Prototype (SVP) to gauge the quality of silicon of the design. The steps in the SVP creation process include:

- Placement
- Trial route
- Parasitic extraction

The detailed placement information and the resistance and capacitance parasitics are then used for delay calculation and annotation of physical delays. The `predict_qos` command will operate in incremental mode if the standard cells are placed. Use `-abandon_existing_placement` option to suppress this behavior. The `predict_qos` command will perform virtual buffering by default.

```
rc:/> predict_qos
```

The `predict_qos` command will not work with encrypted netlists. Therefore, de-encrypt your netlist before using the `predict_qos` command.

Analyzing the Results

After synthesis and optimization, you can view the LEF denominated results. If you do not use the `-physical` options of the `report` commands, the measured units will be based on the timing library (.lib). The following command reports the LEF denominated area results:
rc:/ report area -physical
==================================================================
...
Module: controller
Technology library: grey 1.2
Operating conditions: slow
Interconnect mode: ple
==================================================================
Instance Cells Cell Area Net Area
---------------------------------------------
controller 34 694 102

Exporting Files for Place and Route

The final part of the physical flow involves exporting the data for place and route processing. This is done through the write_encounter design command.

The write_encounter command not only outputs the mapped gate-level netlist, but also the SDCs, Scan DEF file, Encounter Configuration file, and many others. The file types and their extensions are provided below:

- Netlist (.v)
- Encounter configuration file (.conf)
- SDC constraints in SDC format (.sdc)
- Tcl script (.enc_setup.tcl)
- Mode file (.mode)
- Scan DEF file (.scan.def)
- MSV-related files (.msv.tcl, .msv.vsf)
- Multiple timing mode (.mmode.tcl)

**Note**: The full DEF file that is outputted is the exact same DEF file that was loaded or generated by predict_qos. However, RTL Compiler generates the information for the Scan DEF file (.scan.def). Although the scan chains will be re-ordered in the back-end once the placement is determined, any scan reordering done in synthesis is based on the current placement. This placement may not be carried forward. For example, the placement will change if more optimization is done in RTL Compiler. There will always be slight adjustments to the scan order, which are best accomplished in the back-end. The scan DEF file is outputted for continual convergence: getting closer to the final result with each reordering.
To export all the results and information from the synthesis session, use the `write_encounter design` command without any options or arguments. The default basename for all the files will be `rc_enc_des/rc`.

```
rc:/> write_encounter design
unix> ls /home/avyscott/rc/rc_enc_des
```

rc.conf
rc.def
rc.enc_setup.tcl
rc.mode
rc.scan.def
rc.sdc
rc.v

Use the `-basename` option to specify both an output directory and a custom basename:

```
rc:/> write_encounter design -basename design_output/final
unix> ls /home/avyscott/rc/design_output
```

final.conf
final.def
final.enc_setup.tcl
final.mode
final.scan.def
final.sdc
final.v

As this physical flow illustrates, the last step before `write_encounter design` should be `predict_qos`. If you perform further optimizations after initially using the `predict_qos` command, use the `predict_qos` command a second time. Repeat this process of always ending with the `predict_qos` command.

⚠️ **Caution**

*Although optimizations after using the `predict_qos` command is allowed, do not perform any netlist changes such as renaming, ungrouping, grouping or bit blasting after `predict_qos`. There is a high probability that some of the information in the DEF will no longer be valid and will be lost.*
Summarizing the Flows

The following example show broad overviews of each of the flows. The first two examples are the flows discussed in this chapter. The subsequent examples are the flows discussed in the other chapters.

Example 3-1 Physical Flow with predict_qos, LEF, and Capacitance Table

```
set_attribute library library_name
set_attribute lef_library {tech.lef cell.lef}
set_attribute cap_table_file cap_file
read_hdl filename
elaborate
read sdc
read_def def_file
synthesize -to_mapped
predict_qos
write_encounter design
```

Example 3-2 Physical Flow with predict_qos and Encounter Configuration File

```
read_encounter config config_file
read_def def_file
synthesize -to_mapped
predict_qos
write_encounter design
```

Example 3-3 Physical Flow without predict_qos

```
set_attribute library library_name
set_attribute lef_library {tech.lef cell.lef}
set_attribute cap_table_file cap_file
read_hdl filename
elaborate
read sdc

synthesize -to_mapped
write_hdl
```

Example 3-4 Generic Flow

```
set_attribute library library_name
read_hdl filename
elaborate
```
read constraints
synthesize -to_mapped
write_hdl
Design For Manufacturing Flow

- **Overview** on page 54
- **Tasks** on page 55
  - Specifying the Yield Coefficients Information on page 55
  - Setting the DFM Flow on page 55
  - Analyzing the Yield Information on page 55
  - Summarizing the Flows on page 57
Overview

RTL Compiler allows you to perform Design for Manufacturing (DFM) optimizations and discover yield information in the DFM flow. During synthesis, RTL Compiler estimates the probability for library cell failure and computes the overall impact on “defect-limited yield” for the whole design. While keeping this impact as a global cost function, RTL Compiler picks cells which have the best combination of timing, area, probability of cell failure, and power during logic structuring.
Tasks

The tasks below list only those that are different from the generic flow or illustrate a new step.

- Specifying the Yield Coefficients Information on page 55
- Setting the DFM Flow on page 55
- Analyzing the Yield Information on page 55
- Summarizing the Flows on page 57

Specifying the Yield Coefficients Information

The yield coefficients file provides the probability for failure of each library cell. The cell failure rates are typically characterized by some library analysis methods based on the cell layout and fabrication yield characterization data.

To load the yield coefficients file, use the `read_dfm` command:

```bash
rc:/> read_dfm penny.dfm
```

RTL Compiler will annotate the defect probability of any matching cells between the coefficients file and the timing library.

If you have multiple coefficients file, use the `read_dfm` command for each file:

```bash
rc:/> read_dfm penny.dfm
rc:/> read_dfm flame.dfm
```

Setting the DFM Flow

Before synthesis, you must set RTL Compiler into yield synthesis mode through the `optimize_yield` attribute.

```bash
rc:/> set_attribute optimize_yield true /
```

The default value of this attribute is `false`.

Analyzing the Yield Information

After synthesizing the design to gates, find the yield cost and yield percentage for each instance with the `report yield` command:

```bash
rc:/> report yield
```
The command shows the defect-limited yield impact for library cell defects.

To find the yield cost and yield percentage values for each library cell, use the 
-yield option of the report gates command:

```
rc:/> report gates -yield
```

```
Gate  Instances   Area       Cost       Yield    Library
---------------------------------------------------------------------
flopdrs  33  264.000  3.39278e-06  99.9997    tutorial
inv1    103   51.500   1.5022e-06  99.9998    tutorial
nand2   315  315.000  1.08311e-05  99.9989    tutorial
nor2    19   28.500  6.79989e-07  99.9999    tutorial
---------------------------------------------------------------------
total   470  659.000  1.64061e-05  99.9984
```

```
Type    Instances   Area  Area %
------------------------------------
sequential        33 264.000   40.1
inverter         103  51.500    7.8
logic            334 343.500   52.1
------------------------------------
total            470 659.000  100.0
```

Chip will have other effects (for example, systematic and random) that also lower yield. If a given chip had zero defect-limited yield losses, it would still not reach 100% yield due to these other effects. In logic synthesis, RTL Compiler consider only the cell defects.

You must contact the fabrication facility to understand what percentage of failures are due to the cell based defect-limited yield effects. For example, report yield may estimate that the yield is 90% while the true yield may only be 80% due to other effects that are not currently modeled. Cell failure rates are given as a simple failure rate per instance of the cell used.
To find the total yield for the design, use the *yield* attribute with the `get_attribute` command. The following example finds the yield for the design `penny`:

```plaintext
rc:> get_attribute yield [find . -design penny]
0.999983594076
```

### Summarizing the Flows

The following example show broad overviews of each of the flows. The first example is the flow discussed in this chapter. The subsequent examples are the flows discussed in the other chapters.

#### Example 4-1  DFM Flow

```plaintext
set_attr library *library_name*
read_dfm *coefficients_file*
read_hdl *filename*
elaborate
read sdc
set_attribute optimize_yield true /
synthesize -to_mapped
report yield
report gates -yield
get_attribute yield
```

#### Example 4-2  Physical Flow with predict_qos, LEF, and Capacitance Table

```plaintext
set_attribute library *library_name*
set_attribute lef_library {tech.lef cell.lef}
set_attribute cap_table_file *cap_file*
read_hdl *filename*
elaborate
read sdc
read_def *def_file*
synthesize -to_mapped
predict_qos
write_encounter design
```

#### Example 4-3  Physical Flow with predict_qos and Encounter Configuration File

```plaintext
read_encounter config *config_file*
read_def *def_file*
synthesize -to_mapped
predict_qos
```
write_encounter design

Example 4-4  Physical Flow without predict_qos
set_attribute library  library_name
set_attribute lef_library {tech.lef cell.lef}
set_attribute cap_table_file cap_file
read_hdl  filename
elaborate
read sdc
synthesize -to_mapped
write_hdl

Example 4-5  Generic Flow
set_attribute library  library_name
read_hdl  filename
elaborate
read constraints
synthesize -to_mapped
write_hdl