Genus User Guide

Product Version 22.1
May 2023

Contents

Preface 13

About This Manual 14

Additional References 14

Reporting Problems or Errors in Manuals 15

Customer Support 16

- Cadence Online Support 16
- Other Support Offerings 16

Supported User Interfaces 18

Messages 19

Man Pages 20

Command-Line Help 21

- Getting the Syntax for a Command 21
- Getting Attribute Help 21
- Searching For Commands When You Are Unsure of the Name 21

Documentation Conventions 22

- Text Command Syntax 22

1
Introduction 25

Overview 27

Installing the Genus Software 27

Licensing 27

Getting Started with Genus 28

- The CDN_SYNTH_ROOT Variable 28
- Using the Initialization File 28
- Invoking Genus 29
- Customizing the Log File and Command File Names 32
- Setting Information Level and Messages 33

Working in the Genus Shell 34

- Navigation 34
- Objects and Attributes 35

Output Redirection . . . 36
Scripting . . . 37
Using SDC Commands . . . 38
Getting Help . . . 39
Getting Help on a Command or an Attribute . . . 39
Getting Help on an Attribute . . . 39
Genus Messages: Errors, Warnings, and Information . . . 41
Tips and Shortcuts . . . 42
Accessing UNIX Environment Variables from Genus . . . 42
Working with Tcl in Genus . . . 43
Using Command Abbreviations . . . 44
Using Tab Completion . . . 45
Using Wildcards . . . 46
Using the Command Line Editor . . . 47

2
Genus Design Information Hierarchy . . . 51

Overview . . . 52
Setting the Current Design . . . 52
Specifying Hierarchy Names . . . 53
Describing the Design Information Hierarchy . . . 54
Working in the Top-Level (root) Directory . . . 54
Working in the designs Directory . . . 57
Working in the Library Directory . . . 68
Working in the hdl_libraries Directory . . . 72
Working in the obj_types Directory . . . 82
Manipulating Objects in the Design Information Hierarchy . . . 84
Ungrouping Modules During and After Elaboration . . . 84
Finding Information in the Design Information Hierarchy . . . 87
Using the vcd Command to Navigate the Design Information Hierarchy . . . 87
Using the vls Command to List Directory Objects and Attributes . . . 88
Using the get_db Command to Search for Information . . . 90
Navigating a Sample Design . . . 94
Saving the Design Information Hierarchy . . . 100

3
Using the Libraries 101
Overview 102
Tasks 103
  Specifying Explicit Search Paths 103
  Specifying Implicit Search Paths 104
  Specifying Settings that Influence Handling of Library Cells 105
  Setting the Target Technology Library 105
  Preventing the Use of Specific Library Cells 107
  Forcing the Use of Specific Library Cells 107
  Working with Liberty Format Technology Libraries 108
  Using Voltage Scaling 109
  Library Scaling in Multi-Mode Multi-Corner (MMMC) Flow 111
Troubleshooting 112
  Cells Identified as Unusable 112
  Cells Identified as Timing Models 117

4
Loading Files 123
Overview 124
Tasks 125
Updating Scripts through Patching 125
Running Scripts 126
Reading HDL Files 127
  Loading HDL Files 127
  Specifying the HDL Language Mode 130
  Specifying HDL Search Paths 132
Reading Verilog Files 133
  Defining Verilog Macros 133
Reading VHDL Files 141
  Specifying the VHDL Environment 141
  Verifying VHDL Code Compliance with the LRM 143
  Specifying Illegal Characters in VHDL 143
  Showing the VHDL Logical Libraries 143

Using Arithmetic Packages from Other Vendors . . . 143
Modifying the Case of VHDL Names . . . 144
Reading Designs with Mixed Verilog and VHDL Files . . . 145
Reading in Verilog Modules and VHDL Entities with Same Names . . . 145
Using Case Sensitivity in Verilog/VHDL Mixed-Language Designs . . . 145
Reading and Elaborating a Structural Netlist Design . . . 147
Reading a Partially Structural Design . . . 148
Keeping Track of Loaded HDL Files . . . 150
Importing the Floorplan . . . 150

5
Elaborating the Design . . . 151
Overview . . . 152
Tasks . . . 153
Performing Elaboration . . . 153
Specifying Top-Level Parameters or Generic Values . . . 154
Specifying HDL Library Search Paths . . . 156
Elaborating a Specified Module or Entity . . . 156
Naming Individual Bits of Array and Record Ports and Registers . . . 156
Naming Individual Bits of Multi-Bit Wires . . . 158
Naming Parameterized Modules . . . 159
Keeping Track of the RTL Source Code . . . 162
Grouping an Extra Level of Design Hierarchy . . . 163

6
Applying Constraints . . . 181
Overview . . . 182
Tasks . . . 183
Importing and Exporting SDC . . . 183
Applying Timing Constraints . . . 183
Importing Physical Information . . . 184
Applying Design Rule Constraints . . . 184
Creating Ideal Objects . . . 185

7
Defining Optimization Settings 187
Overview 188
Preserving Instances and Modules 189
Grouping and Ungrouping Objects 190
Grouping 190
Ungrouping 191
Partitioning 193
Setting Boundary Optimization 194
Mapping to Complex Sequential Cells 196
Deleting Unused Sequential Instances 197
Controlling Merging of Combinational Hierarchical Instances 198
Optimizing Total Negative Slack 199
Making DRC the Highest Priority 200
Creating Hard Regions 201
Deleting Buffers and Inverters Driven by Hard Regions 201
Preventing Boundary Optimization through Hard Regions 202

8
Reducing Runtime Using Super-Threading 203
Overview 204
Licensing and CPU Usage 205
Using Super-Threading on Local Host 206
Using Super-Threading on Remote Shell 206
Using Super-Threading on Platform Load Sharing Facility (LSF) 207
Using Super-Threading on Sun Grid Engine (SGE) 209

9
Performing Synthesis 211
Overview 212
RTL Optimization 213
Global Focus Mapping 213
Global Incremental Optimization 213
Incremental Optimization (IOPT) 214

Tasks . . . 215
Synthesizing your Design . . . 216
Synthesizing Submodules . . . 218
Synthesizing Unresolved References . . . 219
Re-synthesizing with a New Library (Technology Translation) . . . 219
Setting Effort Levels . . . 221
Quality of Silicon Prediction . . . 222
Generic Gates in a Generic Netlist . . . 223
Generic Flop . . . 224
Generic Latch . . . 225
Generic Mux . . . 225
Generic Dont-Care . . . 227
Writing the Generic Netlist . . . 228
Reading the Netlist . . . 238
Analyzing the Log File . . . 239

10
Clock Mapping Flow . . . 245
Overview . . . 246
Clock Mapping Flow . . . 246
Flow Example . . . 247

11
Retiming the Design . . . 249
Overview . . . 250
Retiming for Timing . . . 251
Retiming for Area . . . 251
Tasks . . . 252
Retiming Using the Automatic Top-Down Retiming Flow . . . 252
Manual Retiming (Block-Level Retiming) . . . 255
Incorporating Design for Test (DFT) and Low Power Features . . . 257
Localizing Retiming Optimizations to Particular modules . . . 260
Controlling Retiming Optimization . . . 261
Retiming Registers with Asynchronous Set and Reset Signals . . . 262
Identifying Retimed Logic . . . 266

Retiming Multiple Clock Designs 267

12
Performing Functional Verification 269

Overview 270
Tasks 270
Writing Out dofiles for Formal Verification 270

13
Generating Reports 271

Overview 272
Tasks 273
Generating Timing Reports 273
Generating Area Reports 275
Tracking and Saving QoR Metrics 277
Summarizing Messages 285
Redirecting Reports 286
Customizing the report_* Commands 286
. . . . 286

14
Using the Genus Database 287

Overview 288
Tasks 289
Saving the Netlist and Setup 289
Restoring the Netlist and Setup 289
Splitting the Database 289

15
Interfacing to Place and Route 291

Overview 292
Preparing the Netlist for Place-and-Route or Third-Party Tools 293
Changing Names 293

- Naming Flops 294
- Removing Assign Statements 295
- Inserting Tie Cells 297
- Handling Bit Blasted Port Styles 298
- Handling Bit-Blasted Constants 299

Generating Design and Session Information 300

- Saving and Restoring a Session in Genus 301

Writing Out the Design Netlist 302

Writing SDC Constraints 305

Writing an SDF File 306

16

Modifying the Netlist 307

Overview 308

Connecting Pins, Ports, and hports 309

Disconnecting Pins, Ports, and hports 309

Creating New Instances 310

Overriding Preserved Modules 311

Creating Unique Parameter Names 312

Naming Generated Components 313

Changing the Instance Library Cell 313

17

IP Protection 315

Overview 316

- Decryption and Encryption using xmprotect 316

Supported Encryption Flows 317

- Variation due to encryption pragma 317
- Variation due to type of encryption key 319

Levels of Protection 321

Round-trip Protection Flow 322

Details and Examples of Protection Features 323

- Encrypting Designs within Genus 323
- Encrypting Designs outside Genus 324
- Loading Encrypted Designs 325

Writing Encrypted Designs 326
Attributes — "protected" and "encrypted" 326

A
Simple Synthesis Template 329

Index 331

Preface

- About This Manual on page 14
- Additional References on page 14
- Reporting Problems or Errors in Manuals on page 15
- Customer Support on page 16
- Supported User Interfaces on page 18
- Messages on page 19
- Man Pages on page 20
- Command-Line Help on page 21
- Documentation Conventions on page 22

About This Manual

This manual describes how to use Genus using the common user interface.

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
- *Practical Programming in Tcl and Tk*, Brent Welch and Ken Jones
- 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. 1364-2005)
- IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language (IEEE STD 1800-2009)
- IEEE Standard VHDL Language Reference Manual (IEEE Std. 1076-1987)
- IEEE Standard VHDL Language Reference Manual (IEEE Std. 1076-1993)
- IEEE Standard VHDL Language Reference Manual (IEEE Std. 1076-2008)

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

Reporting Problems or Errors in Manuals

The Cadence® Help online documentation, lets you view, search, and print Cadence product documentation. You can access Cadence Help by typing cdnshelp from your Cadence tools hierarchy.

Contact Cadence Customer Support to file a CCR if you find:

- An error in the manual
- An omission of information in a manual
- A problem using the Cadence Help documentation system

Customer Support

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

Cadence Online Support

The Cadence® online support website 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, case tracking, up-to-date release information, full site search capabilities, software update ordering, and much more. For more information on Cadence online support go to http://support.cadence.com

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.
- **University software program support**—Provides you with the latest information to answer your technical questions.
- **Training Offerings—**Cadence offers the following training courses for Genus:
  - Genus Synthesis Solution
  - Basic Static Timing Analysis
  - Fundamentals of IEEE 1801 Low-Power Specification Format
  - Advanced Synthesis with Genus Synthesis Solution
  - Low-Power Synthesis Flow with Genus Synthesis Solution

  The courses listed above are available in North America. For further information on the training courses available in your region, visit Cadence Training or write to training_enroll@cadence.com.

  Note: The links in this section open in a new browser.
- **Digital Badge Exams—**Cadence offers **Digital Badge** exams for all the Genus courses for our customers and University students. The exams can be taken from Training Courses (cadence.com).

Upon passing the exams, you get certificates (verified by https://credly.com/), which can then be shared on your LinkedIn profiles, Facebook, and Twitter. For more Information, refer to 'Become Cadence Certified'.

- **Video Library**

  Several videos are available on the support website: Genus: Video Library

For more information on the support offerings go to http://www.cadence.com/support

Supported User Interfaces

Genus supports the following user interfaces:

- **Unified User Interface.** Genus, Innovus and Tempus offer a fully unified Tcl scripting language and GUI environment. This unified user interface (also referred to as Stylus common UI) streamlines flow development and improves productivity of multi-tool users.

  When you start Genus, you will by default start with the Stylus common UI. You will see the following prompt:

  ```
  genus@root:>
  ```

- **Legacy User Interface.** Genus can also operate in legacy mode which supports RTL Compiler commands/attributes and scripting.

  To start Genus with legacy UI, you can

  - Start the tool with legacy UI as follows:

    ```
    %genus -legacy_ui -files script
    ....
    legacy_genus:/>
    ```

  - Switch to legacy UI if you started the tool with the default Stylus common UI.

    ```
    %genus
    genus@root:1> set_db common_ui false
    legacy_genus:/>
    ```

Important

This document provides information specific to the Stylus common User Interface.

Messages

- You can get detailed information for each message issued in your current Genus run using the `report_messages` command.

  ```
  genus@root:3> report_messages
  ```

  The report also includes a summary of how many times each message was issued.

- You can also get specific information about a message.

  For example, to get more information about the `TUI-613` message, you can type the following command:

  ```
  genus@root:6> vls -a TUI-613
  message:TUI/TUI-613 (message)
      Attributes:
        base_name = TUI-613
        count = 0
        escaped_name = TUI/TUI-613
        help = The user_speed_grade is only applicable to datapath subdesigns.
        id = 613
        name = TUI/TUI-613
        obj_type = message
        print_count = 0
        priority = 1
        screen_print_count = 0
        severity = Warning
        type = The attribute is not applicable to the object.
  ```

  You can also use the `help` command:

  ```
  genus@root:7> help TUI-613
  Message:
    name:                    TUI/TUI-613
    severity:                Warning
    type:                    The attribute is not applicable to the object.
    help:                    The user_speed_grade is only applicable to datapath
  subdesigns.
  ```

If you do not get the details that you need or do not understand a message, either contact Cadence Customer Support to file a CCR or email the message ID you would like improved to synthesis_pubs@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 Genus.

To use man pages from UNIX shell:

1. Set your environment to view the correct directory:

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

2. Access the manpage by either of the following ways:

   - Enter the name of the command or attribute that you want. For example:

     - `man check_dft_rules`
     - `man max_output_voltage`

   - Specify a section number with `man` command to look for the command or attribute information in the specific section of the on-line manual.

     Commands are in section 1, attributes are in section 2, and messages are in section 3 of the on-line manual. In the absence of section number, `man` will search through sections 1, 2, 3 (in this sequence) and display the first matching manual page.

     This is useful in cases where both commands and attributes exist with the same name. For example:

     - `man 1 retime`

       will display manhelp for `retime` command

     - `man 2 retime`

       will display manhelp for `retime` attribute

Note: Refer to man for more information on the `man` command.

Command-Line Help

You can get quick syntax help for commands and attributes at the Genus 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_pubs@cadence.com

Getting the Syntax for a Command

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

For example:

```
genus@root:5> help path_group
```

This returns the syntax for the `path_group` command.

Getting Attribute Help

Type the following:

```
genus@root:6> help attribute_name
```

For example:

```
genus@root:7> help max_transition
```

This returns the help for the `max_transition` attribute and shows on which object types the attribute can be specified.

Searching For Commands When You Are Unsure of the Name

You can use help to find a command or a Tcl process if you only know part of its name, even as little as one letter. You can type a single letter or sequence of letters and press `Tab` to get a list of all commands and any user-defined Tcl processes that start with that letter(s). For example:

```
    @genus:root: number> ad <Tab>
```

This returns the following commands:

```
add_assign_buffer_options        add_clock_gates_obs
add_clock_gates_test_connection  add_opcg_hold_mux
...
```

Documentation Conventions

To aid the readers understanding, a consistent formatting style has been used throughout this manual.

- UNIX commands are shown following the `unix>` string.
- Genus commands are shown following the `genus@root:>` string.

Text Command Syntax

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

`literal`	Non-italic 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

- Overview on page 27
- Installing the Genus Software on page 27
- Licensing on page 27
- Getting Started with Genus on page 28
    - The CDN_SYNTH_ROOT Variable on page 28
    - Using the Initialization File on page 28
    - Invoking Genus on page 29
    - Customizing the Log File and Command File Names on page 32
    - Setting Information Level and Messages on page 33
- Working in the Genus Shell on page 34
    - Navigation on page 34
    - Objects and Attributes on page 35
    - Output Redirection on page 36
    - Scripting on page 37
    - Using SDC Commands on page 38
- Getting Help on page 39
    - Getting Help on a Command or an Attribute on page 39
    - Getting Help on an Attribute on page 39
    - Genus Messages: Errors, Warnings, and Information on page 41
- Tips and Shortcuts on page 42
    - Accessing UNIX Environment Variables from Genus on page 42
    - Working with Tcl in Genus on page 43

- ❑ Using Command Abbreviations on page 44
- ❑ Using Tab Completion on page 45
- ❑ Using Wildcards on page 46
- ❑ Using the Command Line Editor on page 47

Overview

Genus 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. Genus complements the existing Cadence solutions and delivers the best wires for nanometer-scale designs.

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

Installing the Genus Software

See the online *Cadence Installation Guide* that accompanies the Genus software for a detailed description on how to install Genus.

For updating the Genus software with patches to fix certain issues without waiting for an official release, refer Updating Scripts through Patching on page 125.

Licensing

See the online *Cadence License Manager* that contains details of the Cadence Licensing features and policies. This document also explains how you can customize the *options* file as per your requirements.

Along with the details found in the *Cadence License Manager*, Genus has an additional “License Time-out” feature. With this feature, after one hour of inactivity, Genus informs the license server about the inactivity. The license server waits another TIMEOUT seconds (minimum: 3600 seconds) to take away the license from the session and add it back to the license pool. If, now, you want to return back to your Genus session, you may have to wait for the availability of the license to resume work on the session. But the time-out will occur only if TIMEOUT entry was added to the *options* file. Without a TIMEOUT entry is the *options* file, licenses are never returned to the license pool in case of inactivity.

Getting Started with Genus

- The CDN_SYNTH_ROOT Variable on page 28
- Using the Initialization File on page 28
- Invoking Genus on page 29
- Customizing the Log File and Command File Names on page 32
- Setting Information Level and Messages on page 33

The CDN_SYNTH_ROOT Variable

The `CDN_SYNTH_ROOT` environment variable points to the directory where Genus is installed and is always set to:

```
installation_directory/tools
```

You do not have to manually set this variable and all your other settings that reference `CDN_SYNTH_ROOT` will reflect this path. Manually changing `CDN_SYNTH_ROOT` to a different path will have no effect, since it will always be overridden by Genus when Genus loads.

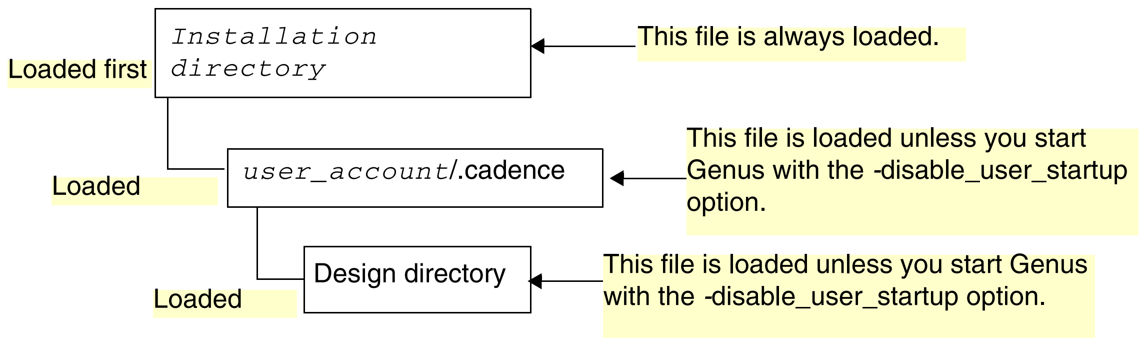
Using the Initialization File

The `genus.tcl` initialization file contains the setup information for Genus. The sourcing sequence for `genus.tcl` file is:

- Genus startup file — `$CDN_SYNTH_ROOT/lib/cdn/genus_startup.tcl`
- The installation root directory — *`installation_dir`*`/etc/genus/genus.tcl` — The file in this directory usually contains the *site-specific* setup.
- The `.cadence` directory in your home directory—`~/.cadence/genus/genus.tcl` — The `genus.tcl` file in this directory contains your *user-specific* setup. This file in not loaded if you launch Genus with the `-n` option.
- The `.cadence` directory in current directory — `./.cadence/genus/genus.tcl`
- The current design directory — `./genus_startup.tcl`.

Figure 1-1 on page 29 illustrates the possible locations and loading priorities of the `genus.tcl` file.

Figure 1-1 Locations of the genus.tcl file


Loaded first
Installation directory
This file is always loaded.
Loaded
user_account/.cadence
This file is loaded unless you start Genus with the -disable_user_startup option.
Loaded
Design directory
This file is loaded unless you start Genus with the -disable_user_startup option.

Invoking Genus

```
genus [-abort_on_error] [-batch] [-del_scale 10]
     [-disable_user_startup] [-execute command]+ [-files string]+
     [-help] [-legacy_ui] [-lic_stack integer] [-lic_startup string]
     [-lic_startup_options string]+ [-log prefix] [-no_gui]
     [-legacy_gui] [-overwrite] [-version] [-wait integer]
```

Note: You can abbreviate the options for the `genus` command as long as there are no ambiguities with other options.

Options and Arguments

`-abort_on_error`	Specifies that Genus must exit if a script error is found.
`-batch`	Exits after processing the scripts specified with the `-files` option.
`-del_scale 10`	Enables support for designs with clock frequencies from 5KHz to 500Hz.
`-disable_user_startup`	Specifies to only read the master init file. Specifies to read only the master `genus.tcl` file, located in the installation directory. By default, Genus also loads the initialization file in your home directory and in your current design directory.
`-execute` *`command`*	Specifies the command or Tool Control Language (Tcl) code to execute as a quoted string before any other files specified with the `-files` option are processed.

`-files` *`file_list`*	Specifies the names of the scripts (or command files) to execute. To specify multiple files, enclose the list in quotes.
`-legacy_gui`	Starts the tool with legacy GUI. The tool will invoke the old Genus GUI when `gui_show` command is used.
`-legacy_ui`	Starts the tool in legacy UI. This means that the tool recognizes most Legacy Genus commands and attributes.
`-lic_stack` *`integer`*	Specifies the number of licenses to use for Virtuoso Digital Implementation (VDI). **Note:** When using a VDI license, you can only stack two licenses, increasing your capacity limit to 100 K instances. *Important* The licenses must be on the same server.

`-lic_startup` *`string`*

Specifies which license to use at startup. If the specified license is unavailable, startup will not continue and the command will fail. When you specify this option multiple times, the command looks for the first available license starting with the first specified one.

If no license is specified, Genus checks out licenses in the following order:

```
Genus_Synthesis
Virtuoso_Digital_Implem
Virtuoso_Digital_Implem_XL
```

If none of the startup licenses are available, Genus cycles the list of provided startup licenses till timeout.
For example,

```
genus -lic_startup Genus_Synthesis -
lic_startup Virtuoso_Digital_Implem_XL -wait 2
```

If both `Genus_Synthesis` and `Virtuoso_Digital_Implem_XL` are not available, Genus cycles checking out between `Genus_Synthesis` and `Virtuoso_Digital_Implem_XL`, license till the timeout in 2 minutes (120 seconds).

`-lic_startup_options` *`string`*

Checks out an optional license at startup.

```
Genus_Low_power_Opt
Genus_Physical_Opt
Vdixl_Capacity_Opt
Joules_RTL_Power
```

For example,

```
genus -lic_startup Virtuoso_Digital_Implem_XL
-lic_startup_options  Vdixl_Capacity_Opt
```

You can also use this option to check out a DFT license. To check out multiple DFT licenses, use a quoted string.

```
Encounter_Test_Architect
Encounter_True_Time
Enc_Test_Adv_MBIST_option
ET_Hierarchical_Option
```

`-log` *`prefix`*

Specifies either the full log and command file names or the prefix for both the `.log` and `.cmd` files. The .log file contains the normal logging output, the `.cmd` file contains the TCL commands that were executed.

- If you specify two arguments, such as `-log "a b"`, Genus uses these names as the file names without adding any extension. If you specify `-log "mylog mycmd"`, Genus creates the `mylog` and `mycmd` files.
- If you specify one argument, Genus uses it as the prefix for the log and command files. If you specify `-log test`, Genus creates the `test.log` and `test.cmd` files.

  If the prefix has a period in it, the last extension is stripped off for the `.cmd` usage. For example, `-log out.log` will result in `out.log` and `out.cmd`, and `-log out.a.log` will result in `out.a.log` and `out.a.cmd`.
- If you do not specify the `-log` option, Genus creates the `genus.log` and `genus.cmd` files by default.

If a log file with the (specified) name already exists in your UNIX directory, the new log file will have either the number “`1`” appended to it, or the number will be incremented with “`1`”.

You can disable this behavior by specifying the `-overwrite` option and allow overwriting an existing `.log` file.

Note: Only the existence of `.log` is checked, the existence of the `.cmd` file is not checked.

	You can prevent creation of a file by using `/dev/null`. For example, `-log "my.log /dev/null"` only creates `my.log`. *Default*: `genus`
`-no_gui`	Starts Genus with the Graphical User Interface (GUI) disabled. **Note:** GUI commands are only available in the GUI version of Genus. See the GUI Text in the *Genus Command Reference* for detailed information on GUI commands. **Note:** If you start the tool with this option, you will not be able to run the GUI during this session even when you specify the `gui_show` command.
`-overwrite`	Allows overwriting of the default and specified log files.
`-version`	Returns the version number without launching the executable.
`-wait` *`integer`*	Specifies the queue wait time-out in minutes. *Default*: 10 minutes or 600 seconds

Customizing the Log File and Command File Names

By default, Genus generates a log file and command file named `genus.log` and `genus.cmd`.

The log file contains the entire output of the current Genus 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 33.

The command history file contains a record of all the commands that were issued in a particular session. This file is created in addition to the log file.

You can customize these file names while invoking Genus or during the synthesis session.

➤ Start Genus with the `-log` option. The following example creates the `test.log` and `test.cmd` files.

```
unix> genus -f script_file_name -log test
```

➤ Suppress the generation of any file by specifying `/dev/null` with the `-log` option when invoking Genus. The following command prevents the creation of the log file:

```
unix> genus -f script_file_name -log /dev/null my.cmd
```

➤ Customize the log file within a Genus session through the `stdout_log` attribute:

```
genus:root: 1> set_db stdout_log log_file_name
```

If a log file already exists, the new log file will have either the number “`1`” appended to it, or the number will be incremented with “`1`”.

➡ To customize the command file name, use the command_log attribute within a Genus session. The following example changes the default name of `genus.cmd` to `genus_command_list.txt`:

```
genus:root: 2> set_db command_log genus_command_list.txt
```

If a command file already exists, the new command file will have the number “`1`” appended to it, or the number will be incremented with “`1`”.

Setting Information Level and Messages

➤ To control the amount of information written in the output logfiles, use the following command:

```
genus:root: 6> set_db information_level value
```

where `value` is an integer value between `0` (minimum) and `9` (maximum). The recommended level is `6`.

Tip

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

Working in the Genus Shell

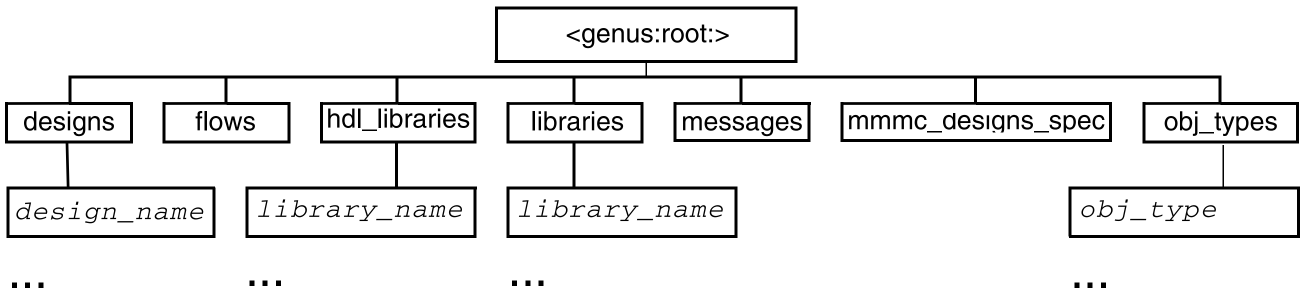
- Navigation on page 34
- Objects and Attributes on page 35
- Output Redirection on page 36
- Scripting on page 37
- Using SDC Commands on page 38

Navigation

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

Genus uses the Design Information Hierarchy to interface with its database. The Design Information Hierarchy is very similar to the UNIX directory structure. The top-level of the Design Information Hierarchy is shown in Figure 1-2 on page 34.

Figure 1-2 Design Information Hierarchy (in Common UI)



Therefore, familiar navigation commands are available to navigate the hierarchy. For example, once you are in Genus, the `vcd` command will change your directory in the Design Information Hierarchy and *not* the UNIX directory tree.

When you invoke Genus, you enter the Design Information Hierarchy at the root directory.

```
genus:root: 11>
```

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

```
genus:root: 12> vls
./                 flows/                  libraries/          tech/
 commands          designs/               hdl_libraries/       messages/
 obj_types/
```

The following command changes the current directory to the `designs` directory:

```
genus:root: 16> vcd designs
```

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

```
genus:root:.designs 27> vpwd
root:.designs
```

For more information regarding the Design Information Hierarchy, refer to Chapter 2, "Genus Design Information Hierarchy." For more information regarding other navigation commands, refer to the Navigation chapter in the *Genus Command Reference*.

Note: Once you are in Genus, you have a limited number of commands (for example, `lcd`, `lls`, `lpwd`, and others) that give you access to the UNIX operating system. For more information about all these commands, refer to the General chapter in the *Genus Command Reference*.

Objects and Attributes

In Genus, objects are general terms for items within the Design Information Hierarchy. For example, an object can be a design, module, 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.

For a complete list of all available attributes, refer to the *Genus Attribute Reference*.

- ➤ To change an attribute setting, use the `set_db` command.
- ➤ To check an attribute value, use the `get_db` command.

Output Redirection

All commands in the Genus shell output their data to the standard output device (`stdout`). To save a record of the data produced, you can 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:

```
genus:root: 20> report_timing > timing.rpt
```

This example appends the timing report to an existing file:

```
genus:root: 21> report_area  >> design.rpt
```

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

Alternatively, you can use the `redirect` command to redirect standard output to a file or a variable.

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, Genus uses the standard scripting language, Tool Control Language (TCL).

In most cases, a Genus script consists of a series of Genus commands listed in a file, in the same format that is used interactively. The script is executed by specifying either the `-f` option with the `genus` command or by using the `include` command from within Genus.

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_db library tech.lib
read_hdl design1.v
elaborate
syn_generic
syn_map
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:/> genus -f design1.g
```

➤ Alternatively, run the script within Genus by typing the following command:

```
genus:root: 1> include design1.g
```

Using SDC Commands

Genus supports Synopsys Design Constraints (SDC). You can either

- Use the `read_sdc` command to read in a Tcl file containing SDC constraints.
- Execute Synopsys Design Constraints (SDC) commands interactively:

  ```
  genus:root: 11> set_output_delay 1.0 -clock foo [get_ports boo*]
  ```

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

  ```
  genus:root: 12>help set_clock_latency
  ```

Important

When you are mixing SDC and Genus 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 Genus command `get_db` will return the load in fF:

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

For a list of supported SDC Commands, refer to SDC Commands in the *Genus Command Reference*.

Getting Help

Online help is available to explain Genus commands, attributes, and messages. You can also access information using the man pages (refer to Man Pages for more information).

This section explains how to get help inside the tool.

■ Getting Help on a Command or an Attribute on page 39

■ Getting Help on an Attribute on page 39

■ Genus Messages: Errors, Warnings, and Information on page 41

Getting Help on a Command or an Attribute

You can get help on a command and its syntax in one of the following ways:

■ Using the `help` command

```
genus:root: 1> help vcd
Commands:
  vcd: sets position in object hierarchy

Usage: vcd [<object>]

    [<object>]:
        dos target directory
```

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

```
genus:root: 2> set_clock_latency -help
```

■ The following command gets help on the `analyzed` attribute.

```
genus:root: 3> help analyzed

Attribute: analyzed (on obj_type: actual_scan_chain)
  Description:            Whether this is an analyzed chain.
  category:               dft
  default_value:          false
  is_computed:            false
  is_settable:            false
  skip_in_db:             false
  type:                   bool { 1 0 true false }
```

Getting Help on an Attribute

To get help on an attribute, use the `help` command.

The following command gets help on the `current_design` attribute.

```
genus:root: 12> help .current_design
Attribute: current_design (on obj_type: root)
  Description:            The current top design.
  category:               netlist
  default_value:          no_value
  is_computed:            false
  is_settable:            false
  skip_in_db:             false
  type:                   object
```

If an attribute is applicable to more than one object type, the help displays the following information:

```
genus:root: 16> help .place_status
Attributes:
  place_status(inst):   # enum { unplaced placed fixed cover soft_fixed }, read
only, default=unplaced, indices={}
                        # Placement status of instance.
  place_status(port):   # enum { unplaced placed fixed cover soft_fixed }, read
only, default=unplaced, indices={}
                        # Placement status of pin.
  place_status(pin):    # enum { unplaced placed fixed cover soft_fixed }, read
only, default=unplaced, indices={}
                    # Placement status of pin (cover, fixed, placed, or unplaced).
  place_status(hpin):   # enum { unplaced placed fixed cover soft_fixed }, read
only, default=unplaced, indices={}
                    # Placement status of pin (cover, fixed, placed, or unplaced).
  place_status(pg_pin): # enum { unplaced placed fixed cover soft_fixed }, read
only, default=unplaced, indices={}
                    # Placement status of pin (cover, fixed, placed, or unplaced).
  place_status(bump):   # enum { unplaced placed fixed cover soft_fixed }, read
only, default=unplaced, indices={}
                        # Placement status of bump.
```

To get help on all a particular object type for a particular attribute, you need to provide the object type in the help command as follows:

```
genus:root: 22> help pin .place_status
Attribute: place_status (on obj_type: pin)
   Description:        Placement status of pin (cover, fixed, placed, or unplaced).
  category:               phys
  default_value:          unplaced
  is_computed:            false
  is_settable:            false
  skip_in_db:             false
  type:                   enum { unplaced placed fixed cover soft_fixed }
```

If you want to return a complete list of both write and read-only attributes, along with their default values, type the following command:

```
genus:root: 24> get_db * *
```

Genus Messages: Errors, Warnings, and Information

If there are any issues during a Genus session, messages categorized as *Errors*, *Warnings*, or *Information* will be issued. All messages allow the process to continue. If you want Genus to fail and stop when it issues an error message, set the `fail_on_error_mesg` root attribute to `true`:

```
genus:root: 11> set_db fail_on_error_mesg true
```

The following messages are examples of warning and information messages:

```
Warning : Could not find scan-equivalent cell [DFT-510]
Info    : Unused module input port [ELABUTL-131]
```

You can use the `help` command to obtain information about particular messages. For example, the following command returns information about the synthesis message `TIM-`11:

```
genus:root: 12> help TIM-11
Message:
  name:                 TIM/TIM-11
  severity:             Warning
  type:                 Possible timing problems have been detected in this design.
  help:                 Use 'report timing -lint' for more information.
```

All messages are located in the `/messages` directory within Genus.

You can also upgrade the severity of a particular message (however, you cannot downgrade the severity). The following example upgrades the severity of the `DFM-200` message from `Warning` to `Error`:

```
genus:message_group:DFM 13> get_db [get_db message:DFM-200] .severity
Warning
genus:message_group:DFM 14> set_db [get_db message:DFM-200] .severity Error
  Setting attribute of message ’DFM-200’: ’severity’ = Error
```

You can also use the `report_messages` command to get a summary of all messages that have been issued in the current run since the last report.

```
report_messages

  =================
   Message Summary
  =================

Num  Sev    Id                       Message Text
--------------------------------------------------------------------
  1 Error TUI-61 A required object parameter could not be found.
                 Check to make sure that the object exists and is
                 of the correct type.  The 'what_is' command can
                 be used to determine the type of an object.
```

Tips and Shortcuts

The following are some helpful tips and shortcuts:

- Accessing UNIX Environment Variables from Genus on page 42
- Working with Tcl in Genus on page 43
- Using Command Abbreviations on page 44
- Using Tab Completion on page 45
- Using Wildcards on page 46
- Using the Command Line Editor on page 47

Accessing UNIX Environment Variables from Genus

You can access your UNIX variables while you are in a Genus session by using the following variable within Genus:

```
$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 Genus, use the `$env` variable with the `init_lib_search_path` attribute:

   ```
   genus:root: 1> set_db init_lib_search_path $::env(LIB_PATH)
   ```

Working with Tcl in Genus

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:

```
genus:root: 12> if {“3.0” == “3”} {puts equal}
equal
```

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

```
genus:root: 14> if {“howisyourevening” eq “howisyourevening”} {puts equal}
equal
```

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

```
genus:root: 18> 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 preceding line are inlined to the line ending in the backslash. For example:

```
genus:root: 19> puts “This will be all\
x==>on one line.”
This will be all on one line.
```

This is Tcl’s idiosyncrasy, not Genus’s.

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**
`assemble_design`	`as`
`hdl_set_vlog_file_extension -sv`	`hdl_set -s`

This abbreviation is possible because there is only one command that starts with “`hd`” which is `hdl_set_vlog_file_extension`, and there is only one reporting option that starts with “-`s`” which is -`sv`.

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 `report_timing` and `report_test_power` both start with “report_t`"`. If you wanted to print out the help for these commands, you would need to abbreviate them as follows:

Complete Command	**Abbreviated Command**
`report_timing`	`report_ti`
`report_test_power`	`report_te`

Using Tab Completion

You can use the Tab key to complete the following items after typing a few letters:

- a command name or a command option
- an attribute name
- a global variable or an environmental variable
- an object path or a file system path

If there are several items that start with that sequence of letters, pressing the Tab key lists all possible items that start with that sequence.

Examples

- If you type the letters `wri` and then press the Tab key, the tool spells out `write_` and then list all commands that start with `write_`:

```
genus:root: 1> wri
write_clp_script            write_congestion_map              write_db
write_def                    write_design            write_dft_abstract_model
write_dft_atpg               write_dft_atpg_other_vendor_files  write_dft_bsdl
write_dft_compression_macro   write_dft_compression_test_points
write_dft_constraints

genus:root: 2> write_
```

- Typing the first letters of a command option and then pressing the Tab key, shows the possible command options:

```
genus:root: 3> read_hdl -l
-language  -library

genus:root: 4> read_hdl -v
```

- Following command shows variable completion:

```
genus:root: 5> lls $env(REGL<tab>
```

  This will complete as

```
genus:root: 6>lls $env(REGLIBS)
```

- If your current file system directory has two directories starting with `my_`, tab completion will show both directories on the file system:

```
genus:root: 9> lls my_
my_design        my_floorplan
```

Using Wildcards

Genus supports the `*` and `?` wild-card characters:

- To specify a unique name in the design.

  For example, the following two commands are equivalent:

  ```
  genus:root: 1> vcd /designs/example1/constants/
  genus:root: 2> vcd /d*/example1/co*/
  ```

  Note: Don't use tab for auto-completion as Genus will convert the path(s) to object(s)

- To specify multiple design elements.

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

  ```
  genus:root: 3> vls *out
  ```

- To find a design with four characters:

  ```
  genus:root: 4> vfind . -design ????
  or
  genus:root: 4> get_db . .designs ????
  ```

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

  ```
  genus:root: 5> vfind . -design ??i
  or
  genus:root: 5> get_db . .designs ????
  ```

The `*` and `?` wild-card characters can also be used together.

Using the Command Line Editor

Genus provides a multi-line editing interface. You can move the cursor to any position and edit any character of a multi-line command before execution.

```
genus:root: 21> set_db \
==> gui_sv_update manual <Enter>
  Setting attribute of root '/': 'gui_sv_update' = manual
```

Multi-line commands are saved as single commands in the command history.

Genus supports several keyboard shortcuts for command-line editing. Using these shortcuts, you can quickly move the cursor within and between the lines of a command before execution. Shortcuts can use independent keys, control characters, or escape sequences. A control character is typed by holding down the Control (Ctrl) key when typing the character. Escape sequences are used by pressing the Escape (Esc) key before pressing the other key(s) in the sequence. The following tables list the supported keyboard shortcuts.

Table 1-1 Keyboard Shortcuts Using Independent Keys

Independent Keys	Result
`up arrow`	Displays the previous command in the history, or in case of a multi-line command, moves to the previous line.
`down arrow`	Displays the next command in the history, or in case of a multi-line command, moves to the next line.
`Home`	Goes to the start of the current line. If already there, goes to the start of the previous line.
`End`	Goes to the end of the current line. If already there, goes to the end of the next line.
`Tab`	Completes the command (option, attribute, variable, and path) or displays all possible commands that start with the current string.

Table 1-2 Keyboard Shortcuts Using Control Characters

Control Characters	Result
`Ctrl-a`	Goes to the beginning of the line.
`Ctrl-b`	Moves the cursor one character to the left.
`Ctrl-d`	Deletes one character at the cursor or lists the directory.
`Ctrl-e`	Goes to the end of the current line.
`Ctrl-f`	Moves the cursor one character to the right.
`Ctrl-g`	Does nothing.
`Ctrl-h`	Deletes one character before the cursor.
`Ctrl-i`	Completes the command or displays all possible commands that start with the current string.
`Ctrl-j`	Submits the line, similar to Enter.
`Ctrl-k`	Deletes all text from the cursor position to the end of the line and copies the content to a yank buffer.
`Ctrl-l`	Clears the screen and re-displays the last line.
`Ctrl-m`	Same as `Ctrl-j`.
`Ctrl-n`	Goes to the next command in the history.
`Ctrl-o`	Accepts the line, moves the history pointer to the next position.
`Ctrl-p`	Goes to the previous command in the history.
`Ctrl-q`	Does nothing.
`Ctrl-r`	Finds in history
`Ctrl-s`	Does nothing.
`Ctrl-t`	Exchanges the characters before the cursor and at the cursor, then moves the cursor one character to the right
`Ctrl-u`	Deletes the line. The content is copied to a yank buffer.
`Ctrl-v`	Does nothing.
`Ctrl-w`	Deletes the characters between the cursor and the position marked by `Esc-space`. The content is copied to a yank buffer.
`Ctrl-x`	Moves the cursor to the position marked by `Esc-space`.

Control Characters	**Result**
`Ctrl-y`	Copies the content from the yank buffer at the cursor position.
`Ctrl-z`	Suspends the session and returns Ctrl to the operating system. Type `fg` to return to the Genus session.
`Ctrl-[`	Does nothing.
`Ctrl-]`	Moves to the next character that is equal to the character under the cursor.
`Ctrl-up arrow`	Displays the previous command in the history, or in case of a multi-line command, moves to the previous command.
`Ctrl-down arrow`	Displays the next command in the history, or in case of a multi-line command, moves to the next command.

Table 1-3 Keyboard Shortcuts Using Escape Characters

Escape Characters	**Result**
`Esc-Ctrl-h`	Deletes a whole word at the left of the cursor.
`Esc-Delete`	Deletes a whole word at the left of the cursor.
`Esc-space`	Marks a position.
`Esc-.`	Inserts the last argument of the last command before the cursor.
`Esc-<`	Displays the first command in the history.
`Esc->`	Displays the last command in the history.
`Esc-?`	Displays a list of all possible file names.
`Esc-b`	Moves the cursor to the beginning of the left word
`Esc-d`	Deletes the word at the right of the cursor
`Esc-f`	Moves the cursor to the beginning of the next word.
`Esc-l`	Changes the characters from the cursor to the end of the word to lower case.
`Esc-P`	Completes the current input by reverse search in the history.
`Esc-u`	Changes the characters from the cursor to the end of the word to upper case.

Escape Characters	**Result**
`Esc-w`	Save the strings between the position marked by `Esc-space,` and the cursor position into the yank buffer.
`Esc-y`	Pastes the yanked string before the cursor.
`Esc-up arrow`	Displays the previous command in the history, or in case of a multi-line command, moves to the previous line.
`Esc-down arrow`	Displays the next command in the history, or in case of a multi-line command, moves to the next line.
`Esc-left arrow`	Moves the cursor one character to the left. Same as `Esc-b`.
`Esc-right arrow`	Moves the cursor one character to the right. Same as `Esc-f`.

2

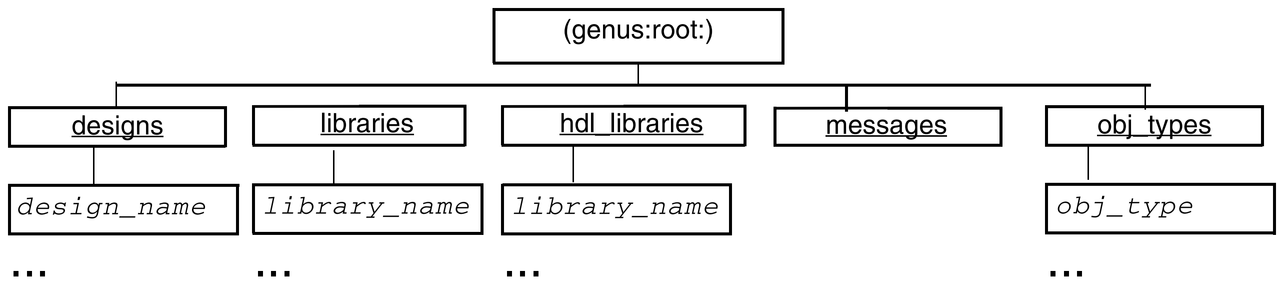
Genus Design Information Hierarchy

- Overview on page 52
    - Setting the Current Design on page 52
    - Specifying Hierarchy Names on page 53
- Describing the Design Information Hierarchy on page 54
    - Working in the Top-Level (root) Directory on page 54
    - Working in the designs Directory on page 57
    - Working in the Library Directory on page 68
    - Working in the hdl_libraries Directory on page 72
    - Working in the obj_types Directory on page 82
- Manipulating Objects in the Design Information Hierarchy on page 84
    - Ungrouping Modules During and After Elaboration on page 84
- Finding Information in the Design Information Hierarchy on page 87
    - Using the vcd Command to Navigate the Design Information Hierarchy on page 87
    - Using the vls Command to List Directory Objects and Attributes on page 88
    - Using the get_db Command to Search for Information on page 90
    - Navigating a Sample Design on page 94
- Saving the Design Information Hierarchy on page 100

Overview

The Design Information Hierarchy contains the design data. When a Genus 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. New hierarchical levels are created within this hierarchy after the libraries are loaded, and the designs loaded and elaborated, as shown in Figure 2-1.

Figure 2-1 Design Information Hierarchy



Setting the Current Design

All Genus operations are only performed on the current design. If you have only one top-level design, then Genus 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:

```
genus:root: 1> vcd design: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` module from optimization, type the following command:

```
genus:root: 2> vcd module:SEQ_MULT/FSH/
genus:module:SEQ_MULT/FSH 3> set_db . .preserve true
```

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

```
genus:root: 3> set_db [get_db modules *FSH] .preserve true
```

`get_db` command can also be used in this case but it needs the design for which the modules are listed. It can be used as follows:

```
genus:root: 3> set_db [get_db design:top_level_design .modules *FSH] .preserve true
```

Specifying Hierarchy Names

You can control the hierarchy names that Genus 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:

```
genus:root: 1> set_db gen_module_prefix name_prefix
```

Genus uses the specified `gen_module_prefix` for all internally generated modules. By default, Genus 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 94 for an example design.

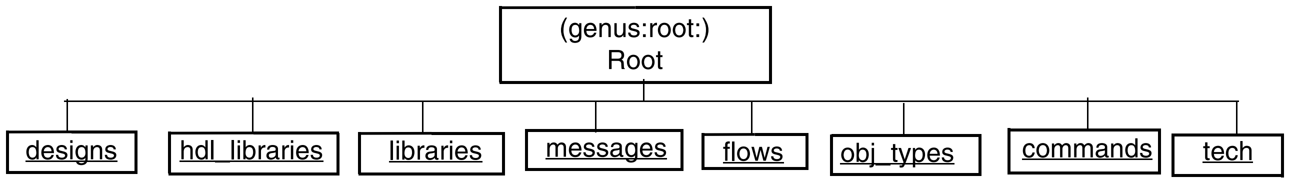
Note: In Genus, anything you can manipulate, such as designs, processes, functions, instances, clocks, or ports are considered "objects".

- Working in the Top-Level (root) Directory on page 54
- Working in the designs Directory on page 57
- Working in the Library Directory on page 68
- Working in the hdl_libraries Directory on page 72
- Working in the obj_types Directory on page 82

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 Genus and is represented by a "`/`", as shown in Figure 2-2. Root attributes contain information about all loaded designs.

Figure 2-2 Top-Level Directory



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

```
genus:design:test 1> vcd
genus:root: 2>
```

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

- `commands`

It contains the list of all commands available in Genus. The subdirectories to the `commands` directory are command options. You can view details about the command options using `vls -a` as follows:

```
genus:command:filter_collection 18> vls -a
```

This will list out all the attributes of the various command options (an object type in Genus Stylus UI).

- `designs`

  Contains all the designs and their associated components. This directory is populated during elaboration and used after elaboration. See Working in the designs Directory on page 57 for detailed information.

- `flows`

  It is a repository for flows and flow steps. Refer to Flowkit in *Genus Command Reference* for details.

- `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.

  See Working in the hdl_libraries Directory on page 72 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 84 for detailed information.

  Note: It is possible to register to ChipWare components with identical component names as long as they do not belong to the same HDL library. However, this practice is discouraged. This name collision of ChipWare components can lead to unexpected results.

- `libraries`

  Contains all the specified technology libraries. See Working in the Library Directory on page 68 for detailed information.

- `messages`

  Contains all messages displayed during a Genus session.

  See the *Genus Message Reference* for a list of all messages.

- `obj_types`

  Contains all the object types in the Design Information Hierarchy. See Working in the obj_types Directory on page 82 for detailed information.

- `tech`

  Contains information obtained from LEF. It has three subdirectories named `layers`, `sites` and `vias` and are populated with information from LEF.

  ```
  genus:root:.tech 27> vls
  ./                  layers/             sites/              vias/
  ```

Working in the designs Directory

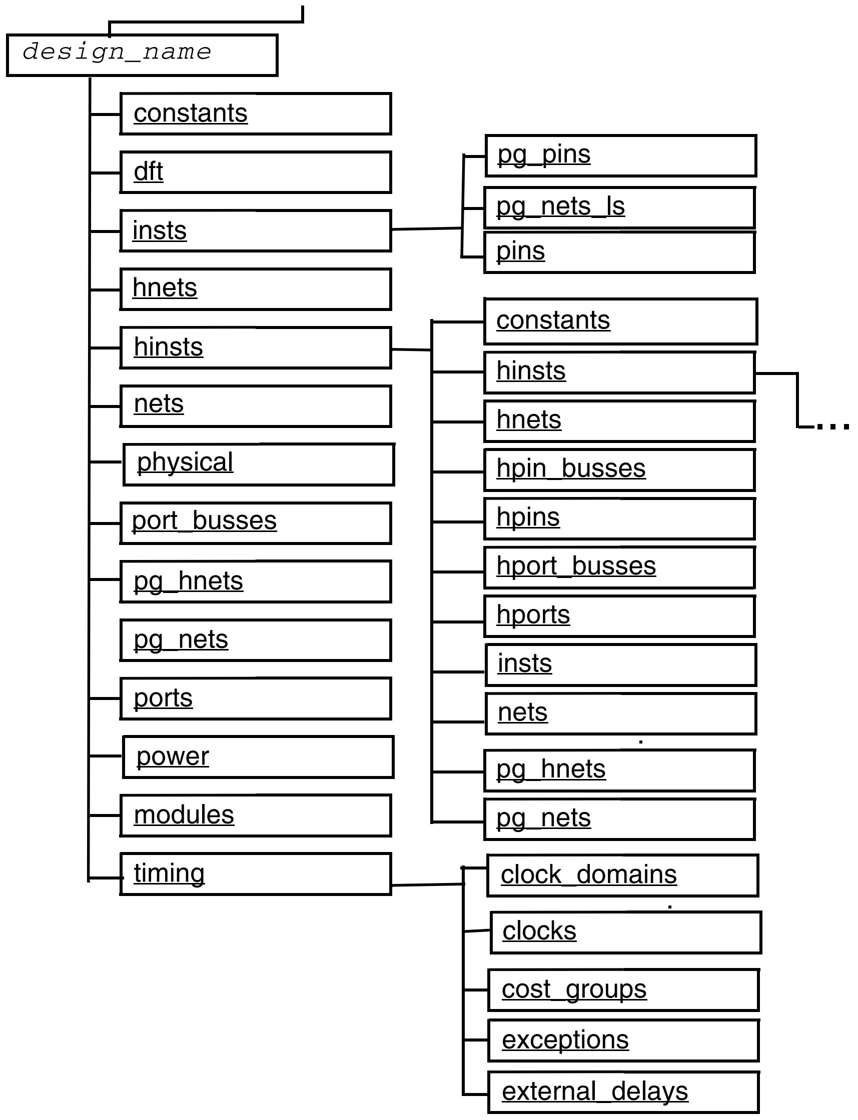
The `designs` directory contains all the designs read in during a Genus session. A `design` corresponds to a module in Verilog that is not instantiated. In other words, it is the top-level Verilog module. The `designs` directory is populated during elaboration and used after elaboration.

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

```
genus:root: 1> vcd designs
```

Figure 2-3 shows the components that the design directory is populated with for each *`design`*.

Figure 2-3 Designs Directory



Each design in the `designs` directory contains the following subdirectories:

- `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`:

  ```
  constant:add/1
  ```

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

```
constant:add/add_b/0
```

- `dft` (Design for Test) contains all the DFT-specific information for the design.

  For more information, refer to DFT Information in the Design Information Hierarchy

- `hinsts` corresponds to a Verilog instantiation. There are four kinds of instances:

  - Instantiated modules—or hierarchical instances—are located in the `hinsts` directory.

    The following is an example of an instantiated module 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/hinsts/s
    ```

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

    ```
    module:my_chip/s
    ```

    Use `get_db` to get the value of the `module` attribute as follows:

    ```
    get_db design:my_chip .module
    ```

  - Instantiated primitives—or leaf level instances that have no instances beneath them—are located in the `insts` directory. It contains all the combinational and the sequential instances.

    Combinational means that the gate output is purely a function of the current values on the inputs, such as a NAND gate or an inverter. Sequential means that the gate has some kind of internal state and typically has a clock input, such as a RAM, flip-flop, or latch.

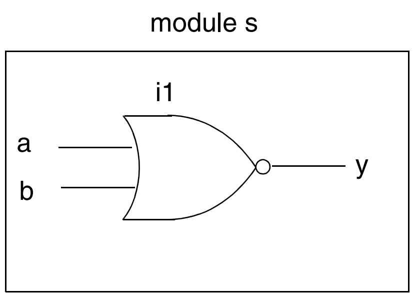
    The following example is an instantiated primitive, which uses `nor` as one of the special Verilog primitive function keywords:

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

    If this instantiation is performed directly inside of module `s`, as shown in Figure 2-4, it would be listed in the design hierarchy as follows:

    ```
       /designs/my_chip/insts/i1
    ```

Figure 2-4 Instantiated Primitive



Instantiated library cells—also referred to as leaf level instances because they have no instances beneath them—are also located in the `insts` directory.

- ❑ To see which instance is combination, sequential or instantiated library cell, you can use the `get_db` command in the following ways:

  `get_db insts -if {.is_combinational == true}` - to return all the combinational instances

  `get_db insts -if {.is_sequential == true}` - to return all the sequential instances

  `get_db insts -if {.hdl_instantiated == true}` - to return all the instantiated library cells

  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/hinsts/s/insts/i1
  ```

- ❑ Unresolved references—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/hinsts/s/insts/u
  ```

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

```
    get_db inst:my_chip/s/u .unresolved
```

- `hnets` refers to a `hierarchical net` 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
  ```

- `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
  ```

- `hpins` and `pins` are a single 1-bit connection point on an instance. `hpins` are hierarchical pins.

  Assume you have the following pins instantiated inside the `my_chip` module:

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

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

  ```
  /designs/my_chip/hinsts/s/hpins/in[0]
  /designs/my_chip/hinsts/s/hpins/in[1]
  /designs/my_chip/hinsts/s/hpins/in[2]
  /designs/my_chip/hinsts/s/hpins/in[3]
  /designs/my_chip/hinsts/s/hpins/out
  ```

- `pg_hnets` and `pg_nets` are the wires connecting the `pg_pins`. `pg_pins` are power and ground pins on instances.

  Power and ground pins can be defined either in the .lib file through the `pg_pin` construct, or in the LEF library with `PIN` definitions that have the `USE` attribute set to either `POWER` or `GROUND`.

  Power and ground pins that are defined as logical pins in the .lib and that have the `USE` attribute set to either `POWER` or `GROUND` in the LEF library, can be converted to `pg_pins` if

  - The `use_power_ground_pin_from_lef` attribute is set to `true` (default)
  - The logical pin definition is not too complex

  A logical pin definition is considered too complex in the following cases:

  - The pin direction in .lib does not match the pin direction in LEF
  - The pin has timing arcs
  - The pin has a function or is used in a function

- ❑ The pin is a member of a bus or bundle
- ❑ The pin is a state retention pin
- ❑ The pin has other attributes than the following: `direction`, `input_signal_level`, `output_signal_level`, `capacitance`, `is_pad`, `max_fanout`, `max_transition`, `max_capacitance`, `connection_class`, and `internal_power` group.

Note: If you also read in a LEF library, you must do so before elaborating the design.

- `hpin_busses` is a bussed connection point.

  Similar to the pin example above, the following `pin_busses` would be listed in the design hierarchy:

```
/designs/my_chip/hinsts/s/hpin_busses/in
/designs/my_chip/hinsts/s/hpin_busses/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.

- `power` contains all power and CPF-related information.
- `ports` is a single 1-bit connection point on a design.

  Assume you have the following Verilog design:

```
module my_chip(a, b, c, d);
    input [2:0] a;
    input b
    ...
```

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

```
a[0]            (port)
a[1]            (port)
a[2]            (port)
b               (port)
```

- `port_busses` represents all bussed input and output ports of a top-level design.

  For example, Genus displays the port and bus inputs in the `alu` design:

```
vls -long /designs/alu/port_busses/
```

```
a[0]            (port_bus)
a[1]            (port_bus)
a[2]            (port_bus)
b               (port_bus)
```

  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.

- `hports` is a single bit-wise connection point within a module that has been instantiated.

  Assume module `sub` is defined in Verilog as follows:

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

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

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

  Then the following hports are listed in the design hierarchy as follows:

  ```
  in[0]           (hport)
  in[1]           (hport)
  out             (hport)
  ```

  See `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. Difference between hport and hpin on page 65 for more information.

- `hport_bus` are bussed connection points within a module.

  Similar to the above `hport` example, `hport_bus` objects are listed in the design hierarchy as follows:

  ```
  in              (hport_bus)
  out             (hport_bus)
  ```

  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 `hport` and the `hport_bus` example above, object `out` would appear in both directories because there is both a `hport` called `out` and a `hport_bus` called `out`.

- `modules` 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 module object is listed in the design hierarchy as:

  ```
  /designs/my_chip/modules/sub
  ```

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

  ```
  get_db design:my_chip .hinsts
  ```

  ```
  hinst:my_chip/s
  ```

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

See Modules on page 96 for information on how to find `modules` in the `design` data structure.

- `timing` contains the following timing and environment constraint subdirectories.
    - `clocks` refers to a defined clock waveform. Clock objects are created using the `create_clock` command. It is an SDC command.

      ```
      create_clock -domain clock_domain
      ```

    - `clock_domains` refers to clocks that are grouped together because they are synchronous to each other, letting you perform timing analysis between these clocks. Genus only computes constraints among clocks in the same clock domain. By default, Genus assumes that every clock object belongs to a single clock domain. Create a `clock_domain` using the `create_clock` command.
    - `cost_groups` refers to a group of timing paths with a single timing optimization objective. During optimization, Genus 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, Genus creates a separate cost group for each clock created using the `create_clock` command.
    - `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:
        - `set_multicycle_path`
        - `set_path_adjust`
        - `set_max_delay`
        - `set_false_path`
        - `group_path`
        - `define_cost_group`
        - `specify_paths`

`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. **Difference between hport and hpin**

It is important to understand the difference between a `hport` and a `hpin` to manipulate the design hierarchy. A `hport` 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 `hports` are similar to ports through which it will pass and receive data.

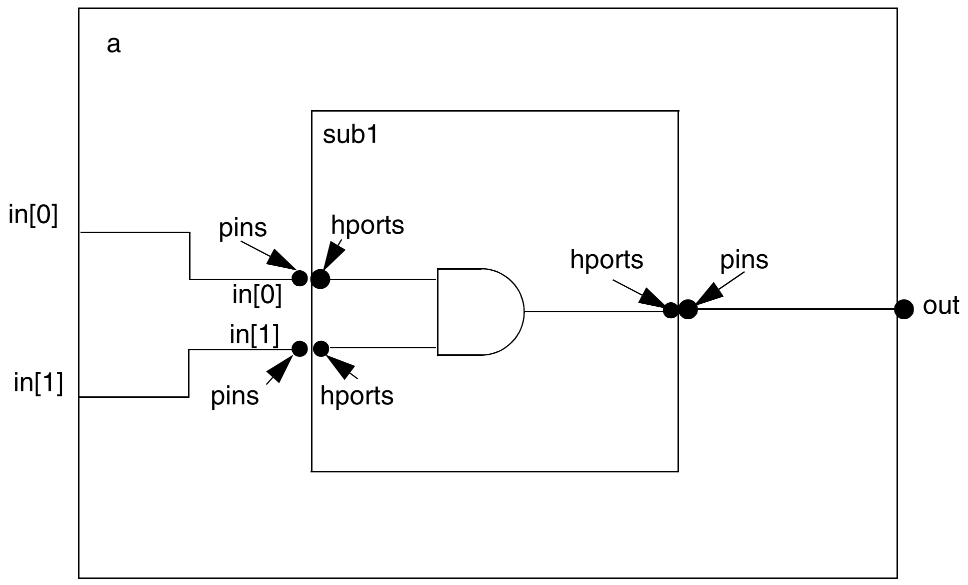
The following example shows the difference between a hport and a pin. Example 2-1 describes a Verilog design.

Example 2-1 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 2-5 shows a schematic representation of this example:

Figure 2-5 Schematic of hports and hpins



During elaboration, Genus generates hports and pins in the design information hierarchy. As shown in Example 2-2, if you check the attributes for `hports` and `hpins` for the `sub` module, you will see the following information:

Example 2-2 hport and hpin Attributes in the Design Hierarchy

```
genus:hinst:a/sub1.hports 16> vls -a
Total: 3 items
./
in1[0]      (hport)
  Attributes:
    base_name = in1[0]
    bus = hport_bus:a/sub1/in1
    direction = in
    escaped_name = sub1/in1[0]
    hinst = hinst:a/sub1
    hnet = hnet:a/sub1/in1[0]
    hpin = hpin:a/sub1/in1[0]
    name = sub1/in1[0]
    net = net:a/in1[0]
    obj_type = hport
in1[1]      (hport)
  Attributes:
    base_name = in1[1]
    bus = hport_bus:a/sub1/in1
    direction = in
    escaped_name = sub1/in1[1]
    hinst = hinst:a/sub1
    hnet = hnet:a/sub1/in1[1]
    hpin = hpin:a/sub1/in1[1]
    name = sub1/in1[1]
    net = net:a/in1[1]
    obj_type = hport

genus:hinst:a/sub1.hpins 19> vls -a
Total: 3 items
./
in1[0]      (hpin)
  Attributes:
    base_name = in1[0]
    direction = in
    escaped_name = sub1/in1[0]
    hinst = hinst:a/sub1
    hnet = hnet:a/sub1/in1[0]
    hport = hport:a/sub1/in1[0]
    name = sub1/in1[0]
    net = net:a/in1[0]
    obj_type = hpin

in1[1]      (pin)
    Attributes:
    base_name = in1[1]
    direction = in
    escaped_name = sub1/in1[1]
    hinst = hinst:a/sub1
    hnet = hnet:a/sub1/in1[1]
    hport = hport:a/sub1/in1[1]
    name = sub1/in1[1]
    net = net:a/in1[1]
    obj_type = hpin
```

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

Working in the Library Directory

The `libraries` directory contains all the libraries read in during a Genus session.

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") {}
```

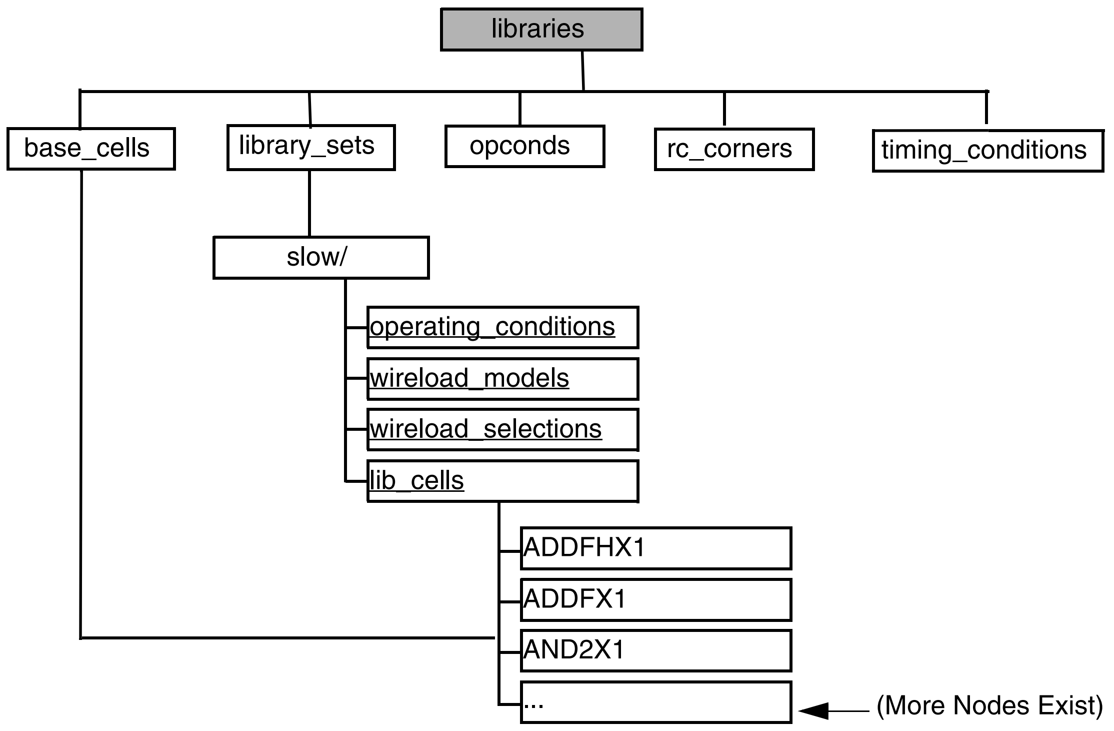
The technology library is listed in the library directory as follows:

```
/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 2-6, if you look into the libraries directory using the `vcd` command and list the contents using the `vls -long` command, there is only one library (`slow`) in the /`libraries/library_sets` directory:

```
genus:root: 5> vcd libraries/library_sets
genus:libraries.library_set 6> vls -l
Total: 2 items
./
slow/    (library_set)
genus:libraries.library_set 6> vcd slow/
genus:library_set:slow 6> vls -l
./       (library_set)
slow1/   (library)
```

Figure 2-6 Library Directory Example



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

```
Total: 5 items

./    (library)
lib_cells/
operating_conditions/
wireload_models/
wireload_selections/
```

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

- `lib_cells/`

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

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

    ```
    timing() {}
    ```

- ❑ `lib_pins` 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:

  ```
  lib_pin:my_technology/INVX1/invx/A
  ```

- ❑ `pg_lib_pins` corresponds to the power-ground pins within a library cell.

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

```
genus:root: 7> vls -l -a lib_pin:design_name/cell_name/pin_name
```

Genus 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`.

```
genus:root:lib_cell:slow/NAND4BBX1 12> vls -l -a Y
```

This displays the following information:

```
Total: 1 item
./          (lib_pin)
    All attributes:
        alive_during_partial_power_down = false
        alive_during_power_up = false
        all_q_pin_of_d_pin =
        async_clear_polarity = none
        async_preset_polarity = none
......
......
        user_function =
        voltage_value =
        x_offset = no_value
        y_offset = no_value
```

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

For a library cell, Genus 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
./          (lib_cell)
    All attributes:
        area = 26.611
        area_multiplier = 1.0
        async_clear_pins =
        async_preset_pins =
        avoid = 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), `vcd` into the directory and list its contents:

```
genus:library:slow.lib_cells 18> vcd NAND4BBXL
genus:lib_cell:slow/NAND4BBXL 19> vls -l
```

This displays information similar to the following:

```
Total: 5items
./    (lib_cell)
lib_arcs/
lib_pins/
pg_lib_pins/
seq_functions/
```

- `operating_conditions/`

  Operating conditions for which the technology library is characterized.

- `wireload_models/`

  The available wire-load models in the technology library. These models are used to calculate the loading effect of interconnect delays in the design

  The information for the wire-load models is stored in associated `wireload` attributes.

  See Finding and Listing Wire-Load Models on page 91 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 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 four directories under each `hdl_libraries` subdirectory. The four directories are: `architectures`, `components`, `configurations` 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 Genus. Component information such as bindings, implementations, parameters, and pins can be found under this directory.

- `.../configurations`

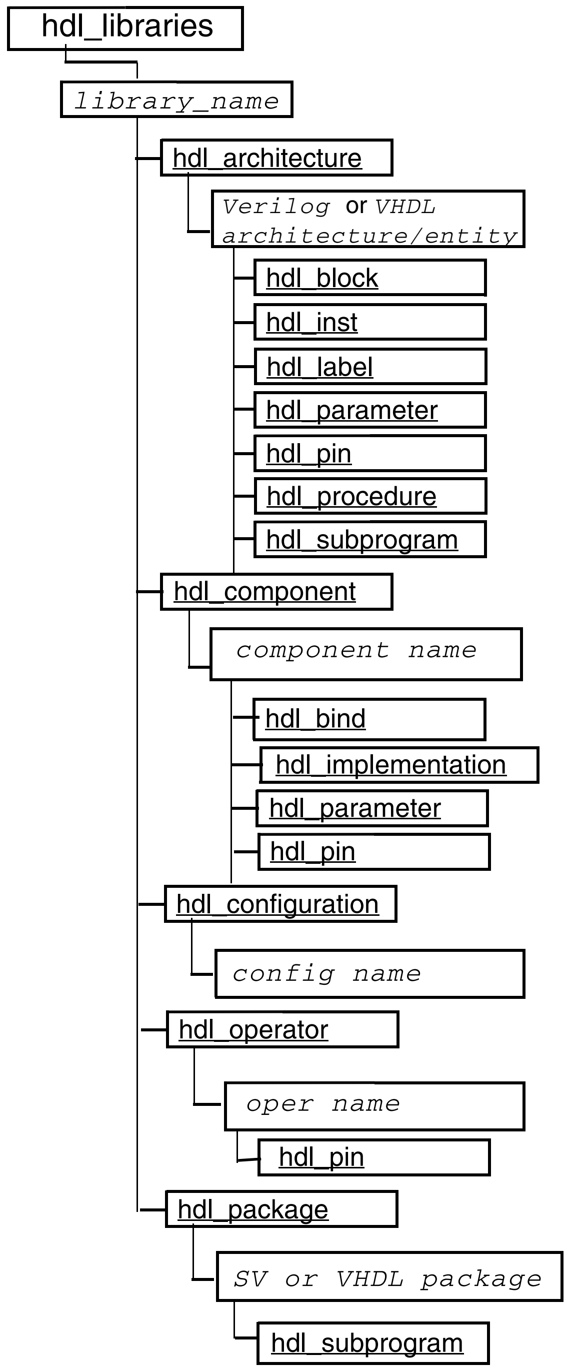
  Contains all configuration names present in the RTL.

- `.../packages`

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

The `hdl_libraries` directory contains the following object types, as shown in Figure 2-7.

Figure 2-7 hdl_libraries Directory



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

```
genus:hdl_libraries 21> vls -l
./
CADENCE/        (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 object 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_architecture`

  Refers to the VHDL architecture/entity or 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. To get a list of the `hdl_architecture` subdirectories, type the `vls` command. For example:

  ```
  genus:root: 4> read_hdl -language vhdl test.vhd
  genus:root: 5> vls /hdl_libraries/default/architectures/test(rtl)/

  blocks/         labels/         hdl_pins/          subprograms/
  instances/      parameters/     processes/
  ```

  - ❑ `hdl_block`

    Refers to a VHDL block VHDL `generate`, or Verilog `generate`, as shown in Example 2-3, in the directory named:

    `/hdl_libraries/library_name/architectures/`*`module_or_architecture_name`*`/blocks`

    To get a list of the blocks, type the `vls` command. For example:

    ```
    genus:root: 9> read_hdl -language vhdl test.vhd

    genus:root: 10> vls /hdl_libraries/default/architectures/test(rtl)/
    blocks/         labels/         pins/         subprograms/
    instances/      parameters/     processes/

    genus:root: 11> vls -l
    /hdl_libraries/default/architectures/test(rtl)/blocks/
    blok      (hdl_block)
    ```

Example 2-3 VHDL Block

```
library ieee;
use ieee.std_logic_1164.all;
entity test is
    port (y : out std_logic_vector (3 downto 0);
        a, b, c :in std_logic_vector (3 downto 0);
        clk : in std_logic);
end;
architecture rtl of test is
    signal p : std_logic_vector (3 downto 0);
begin
    blok : block
    begin
        p <= a and b;
    end block;
    y <= p or c;
end;
```

- `hdl_inst`

  Refers to the HDL instance in the directory named:

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

- `hdl_label`

  Refers to the Verilog or VHDL label in the directory named:

  `/hdl_libraries/library_name/architectures/`*`module_or_architecture name`*`/labels`

- `hdl_parameter`

  Refers to a generic of a VHDL entity, a parameter of a Verilog module, or a parameter of a ChipWare component in the directory named:

  `/hdl_libraries/library_name/architectures/`*`module_or_architecture name`*`/parameters`

  or

  `/hdl_libraries/library_name/architectures/`*`module_or_architecture name`*`/components/parameters`

- `hdl_pin`

  Refers to an input/output port of a VHDL entity, a Verilog module, or a ChipWare component in the directory named:

`/hdl_libraries/default/architectures/`*`module_or_architecture name`*`/pins`

- ❑ `hdl_procedure`

  Refers to the VHDL process or a Verilog `begin` and `end` block, as shown in Example 2-4 on page 76, in the directory named:

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

  Unnamed processes are named using a *noname@linesourcelinenumber* naming convention. To get a list of the hdl_procedure processes, type the `vls` command. For example:

```
genus:root: 18> read_hdl -language vhdl test.vhd
genus:root: 19> vls /hdl_libraries/default/architectures/test(rtl)/
blocks/         labels/         hdl_pins/         subprograms/
instances/      parameters/     processes/
genus:root: 20> vls -l
/hdl_libraries/default/architectures/test(rtl)/processes/
blok      (hdl_procedure)
```

Example 2-4 VHDL Process Begin and End Block

```
library ieee;
use ieee.std_logic_1164.all;
entity test is
    port (y : out std_logic_vector (3 downto 0);
        a, b, c : in  std_logic_vector (3 downto 0);
        clk : in std_logic);
end;
architecture rtl of test is
    signal p : std_logic_vector (3 downto 0);
begin
    blok: process (clk)
    begin
        if clk'event and clk = '1' then
           p <= a and b;
        end if;
    end process;
    y <= p or c;
end;
```

- ❑ `hdl_subprogram`

Refers to the VHDL function/procedure or a Verilog function/task 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 2-5.

Example 2-5 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_component`

  Refers to the ChipWare component in the directory named:

  /hdl_libraries/*library_name*/components/

  - `hdl_bind`

    Refers to the ChipWare binding in the directory named:

    /hdl_libraries/*library_name*/components/*component_name*/bindings

  - `hdl_implementation`

    Refers to the ChipWare implementations in the directory named:

    /hdl_libraries/*library_name*/components/*component_name*/
    implementations

  - `hdl_parameter`

    Lists the HDL parameters used inside the component.

  - `hdl_pin`

    Lists the pins of the Chipware component.

- `hdl_configuration`

  Refers to the Verilog or VHDL configuration.

  VHDL configuration is shown in Example 2-6, in the directory named:

  /hdl_libraries/*library_name*/components/*component_name*/
  configurations

  To get a list of the configurations, type the `vls` command. For example:

```
genus:root: 16> read_hdl -language vhdl test.vhd
genus:root: 17> vls /hdl_libraries/default/
architectures/          configurations/
components/             packages/
genus:root: 18> vls -l /hdl_libraries/default/architectures/
one_gate(my_and)/    (hdl_architecture)
one_gate(my_or)/     (hdl_architecture)
one_gate(my_xor)/    (hdl_architecture)
top(xarch)/          (hdl_architecture)
genus:root: 19> vls -l /hdl_libraries/default/configurations/
xconf       (hdl_configuration)
```

  An example of Verilog configuration is shown in Example 2-7.

Example 2-6 VHDL Configuration

```
entity one_gate is port (q: out bit; j,k: in bit); end;
architecture my_and of one_gate is begin q <= j and k;
end;
architecture my_or  of one_gate is begin q <= j or  k;
end;
architecture my_xor of one_gate is begin q <= j xor k;
end;

entity top is port (a,b,c: in bit; y,z: out bit); end top;
architecture xarch of top is
    component use_cnfg port (q: out bit; j,k: in bit);
end component;
    component one_gate port (q: out bit; j,k: in bit);
end component;
    begin
        u1: use_cnfg port map (q => y, j => a, k => b);
        u2: one_gate port map (q => z, j => a, k => c);
    end xarch;
    configuration xconf of top is
        for xarch
        for u1: use_cnfg use entity work.one_gate(my_and);
    end for;
        for u2: one_gate use entity work.one_gate(my_or );
    end for;
end for;
end configuration;
```

Example 2-7 Verilog Configuration

```
module test(a, b, c);
    input [2:0] a;
    input [2:0] b;
    output [2:0] c;
    foo u0(a[0], b[0], c[0]);
    foo u1(a[1], b[1], c[1]);
    foo u2(a[2], b[2], c[2]);
endmodule

config cfg1;
    design work.test;
    instance test.u0 use lib2.foo;
    default liblist lib1 lib2;
endconfig

config cfg2;
    design work.test;
    default liblist lib1 lib2;
endconfig

config cfg3;
    design work.test;
    instance test.u0 liblist lib1 lib2;
    instance test.u1 use lib2.foo;
    instance test.u2 use foo;
endconfig
```

After reading in the design, use a particular configuration from the above example:

```
genus:root: 23> elab cfg2
```

- `hdl_operator`

  Refers to the ChipWare Developer synthetic operator in the directory named:

  `/hdl_libraries/`*`library_name`*`/components/`*`component_name`*`/operators`

  - ❑ `hdl_pin`

- `hdl_package`

  Refers to the VHDL package in the directory named:

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

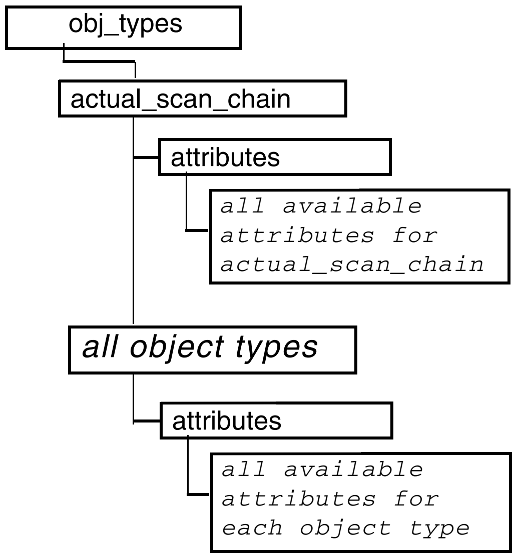
  - ❑ `hdl_subprogram`

To learn how to find information about an HDL object, see Finding Specific Objects and Attribute Values on page 91.

Working in the obj_types Directory

The `obj_types` directory contains the following objects shown in Figure 2-8.

Figure 2-8 obj_types Directory


obj_types
actual_scan_chain
attributes
all available attributes for actual_scan_chain
all object types
attributes
all available attributes for each object type

The `obj_types` directory contains all the object types in the Design Information Hierarchy. To get a list of all the object types, type `vls` on the `obj_type` directory:

```
genus:root: 17> vls /obj_types/
/obj_types:
./                      hdl_parameter/                  pnet/
actual_scan_chain/      hdl_pin/                        port/
actual_scan_segment/    hdl_procedure/                  port_bus/
...
```

Under each object type, there is a subdirectory called `attributes`. Typing `vls -attribute` on this directory not only shows you what attributes are valid for this particular object type, but also default values, help, and other information. For example,

```
genus:root: 19> vls -attribute  /obj_types/actual_scan_chain/attributes/
obj_type:actual_scan_chain.attributes:

Total: 30 items
./
analyzed                          (attribute)
  Attributes:
    base_name = analyzed
    category = dft
    data_type = bool
    default_value = false
    escaped_name = actual_scan_chain/analyzed
    help = Whether this is an analyzed chain.
    is_saved = true
    name = actual_scan_chain/analyzed
    obj_type = attribute
    parent = obj_type:actual_scan_chain
    possible_values = 1 0 true false

    ....
...
```

When you use the `define_attribute` command, you will get the path to your newly created attribute:

```
genus:root: 20> define_attribute -category tui -data_type string -obj_type \
    inst bree_olson

attribute:inst/bree_olson
```

To delete your newly created attribute, use the `delete_obj` command:

```
genus:root: 21> delete_obj attribute:inst/bree_olson

Info    : Removed object. [TUI-58]

        : Removed attribute 'attribute:inst/bree_olson'.
```

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 *Genus 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.

Note: In the low-power flows, it is not recommended to perform any ungrouping before you have read in the CPF file and activity files (TCF, VCD).

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_component` — An Genus or other tool defined component
- `hdl_implementation` — An architecture of a Genus or other tool defined component
- `hdl_architecture` — A user defined module
- `hdl_inst` — An instance of a user defined module, a Genus or other tool defined component

By default, Genus 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_architecture` 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:

  ```
  genus:root: 16> set_db hdl_architecture:default/foo .ungroup true
  ```

- 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:

```
genus:root: 19> set_db inst:default/foo/inst1 .ungroup true
```

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

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

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

  ```
  genus:root: 21> set_db [get_db hdl_components $component_name] .ungroup true
  ```

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

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

  ```
  genus:root: 23> set_db [get_db hdl_architectures $module_name] .ungroup true
  ```

- 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:

  ```
  genus:root: 28> set_db [get_db hdl_insts $instance_name] .ungroup true
  ```

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:

```
genus:root: 31> set_db [get_db hdl_components *CW_add] .ungroup true
```

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 Genus created modules by typing:
   ```
   genus:root: 32> set_db gen_module_prefix CDN_DP_
   ```
2. Read in the Verilog files by typing:
   ```
   genus:root: 34> read_hdl files
   ```
3. Elaborate (build) the design by typing:
   ```
   genus:root: 38> elaborate
   ```
4. Specify your constraints.
5. Synthesize the design by typing:
   ```
   genus:root: 41> syn_generic
   genus:root: 42> syn_map
   ```
6. Ungroup the generated modules before writing out the design by typing:
   ```
   genus:root: 45> set all_subdes [get_db modules CDN_DP*]
         foreach sub_des $all_subdes { \
         set inst [get_db insts $sub_des] \
         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 vcd Command to Navigate the Design Information Hierarchy on page 87
- Using the vls Command to List Directory Objects and Attributes on page 88
- Using the get_db Command to Search for Information on page 90

Using the vcd Command to Navigate the Design Information Hierarchy

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

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:

```
genus:root: 46> vcd /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 `module` directory:

```
genus:root: 53> vcd /obj_types/module/
```

Using the vls Command to List Directory Objects and Attributes

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

See Using the vls Command versus the get_db Command on page 92 to learn the difference in using these two commands.

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

```
genus:root: 10> vls
```

➤ To list all the contents in the long format:

```
genus:root: 11> vls -long
```

or, use the equivalent shortcut command:

```
genus:root: 12> vls -l
```

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

```
genus:root: 13> vls -attribute
```

or, use the equivalent shortcut command:

```
genus:root: 14> vls -a
```

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

```
genus:root: 15> vls -attribute /designs/alu/modules/
design:alu.modules
Total: 2 items
./
addinc65/    (module)
  Attributes:
    arch_filename = /home/example1/Data/alu_mod.v
    arch_name = addinc65
    base_name = addinc65
    design = design:alu
    entity_filename = /home/example1/Data/alu_mod.v
    entity_name = araddinc65b
    escaped_name = addinc65
    fplan_height = 1350.040 microns
    fplan_width = 1375.660 microns
    hdl_all_filelist = {default -v2001 {SYNTHESIS} {/home/example1/Data/
alu_mod.v} {} {}}
    hdl_filelist = {default -v2001 {SYNTHESIS} {//home/example1/Data/
alu_mod.v} {} {}}
    hinsts = hinst:alu/sub1
    language = Verilog
    library_name = default
    lp_clock_gating_max_flops = inf
    lp_clock_gating_min_flops = 3
    name = addinc65
    obj_type = module
```

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

```
vls -a -l
```

or

```
vls -la
```

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

```
genus:root:.designs 18> vls -long
```

Genus displays information similar to the following:

```
genus:root: 19> vls -long /designs/alu/port_busses/
```

```
design:alu.port_busses:
```

```
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):

```
genus:root:.designs 21> vls -computed
```

Genus displays information similar to the following:

```
genus:root:.designs 22> vls -computed
```

```
Total: 2 items
./
MOD69/    (design)
    Attributes:
       analysis_views = analysis_view:MOD69/default_emulate_view
       arch_filename = /home/abc/test1/Data/abc.v
       arch_name = test1
       aspect_ratio = 0.9814
       average_net_length = 59.605301426059775 microns
       base_name = test1
       bbox = {0.0 0.0 1375.66 1350.04} microns
       constant_0_loads =
       constant_0_nets =
       constant_1_loads =
       constant_1_nets =
       constants = /designs/test1/constants/0 /designs/test1/constants/1
       constraint_modes =
       cost_groups = /designs/test1/timing/cost_groups/default
       entity_filename = /home/abc/test1/Data/abc.v
       entity_name = test1
       .......
```

It is a list of all the computed attributes. Values are left blank for attributes whose values cannot be computed.

Using the get_db Command to Search for Information

The `get_db` command in Genus can be used 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 `get_db` 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:

```
genus:root: 8> get_db / <...>
```

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

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

```
genus:root: 9> get_db . <...>
```

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 Genus 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.

```
genus:root: 14> get_db inst:woodward/inst1 .pin out*
pin:woodward/inst1/out1[3]
genus:root: 15> get_db pins out*
pin:woodward/inst1/out1[3]
pin:MOD69/inst2/out1[3]
```

Finding Top-Level Designs, modules, and Libraries

➤ The `get_db` command can also search for the top-level design names:

```
genus:root: 25> get_db designs *
design:SEQ_MULT
```

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

```
genus:root: 26> get_db modules *
```

In this example `SEQ_MULT` has four modules:

```
module:SEQ_MULT/cal
module:SEQ_MULT/chk_reg
module:SEQ_MULT/FSM
module:SEQ_MULT/reg_sft
```

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

```
genus:root: 27> get_db hdl_libraries *GTECH*
hdl_lib:GTECH
```

Finding Specific Objects and Attribute Values

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

```
get_db hdl_libraries *CW
```

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

```
get_db hdl_components *CW_add
```

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

```
get_db hdl_implementations *CW_add/*
```

or:

```
get_db [get_db hdl_components *CW_add] hdl_implementations *
```

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

```
genus:root: 36> vls -long -attribute /hdl_libraries/CW/components/CW_add
hdl_component:CW/CW_add:
Total: 5 items
./                                (hdl_component)
     All attributes:
        avoid = false
        base_name = CW_add
        designware_compatibility = false
        escaped_name = CW/CW_add
        hdl_lib = /hdl_lib:CW
        location =
        name = CW/CW_add
        obj_type = hdl_component
        obsolete = false/
        parameters = wA
..........
..........
```

Finding and Listing Wire-Load Models

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

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

```
genus:root: 21> get_db wireloads *
```

The command displays information similar to the following:

```
wireload:slow/ForQA wireload:slow/CSM18_Conservative
wireload: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:

```
genus:root: 22> get_db library:slow .wireload_models *
```

Finding design attributes

Use the `get_db` 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_db` command.

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

  ```
  genus:root: 11> get_db [get_db modules FSH] .instances
  hinst:SEQ_MULT/I1
  ```

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

  ```
  genus:design:design1.modules 12> get_db module:design1/counter .instances
  hinst:design1/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_db` command.

  ```
  genus:root: 15> get_db module:top/dasein .arch_filename
  ```

  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 vls Command versus the get_db Command

The following examples show the difference between using the `vls` command and the `get_db` command to return the wire-load model.

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

  ```
  genus:root:.designs 12> vls -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_db wireload` command:

  ```
  genus:root:.designs 15> get_db design:async_set_reset_flop_n .wireload
  ```

  and returns the following wire-load model:

  ```
  wireload:slow/sartre18_Conservative
  ```

  The `vls -attribute` command lists all user modified attributes and their values. The `get_db` command lists only the value of the specified attribute. The `get_db` command is especially useful in scripts where the 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.

  ```
  genus:root:.designs 23> vls -computed
  ```

  ```
  Total: 2 items
  ./
  MOD69/    (design)
      Attributes:
         arch_filename = /home/abc/test1/Data/abc.v
         arch_name = test1
         base_name = test1
         constant_0_loads =
         constant_0_nets =
         constant_1_loads =
         constant_1_nets =
         constants = /designs/test1/constants/0 /designs/test1/constants/1
         cost_groups = /designs/test1/timing/cost_groups/default
         entity_filename = /home/abc/test1/Data/abc.v
         entity_name = test1
         .......
  ```

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

  ```
  genus:root:.designs 34> get_db design:stormy .total_area
  ```

  ```
  106.444
  ```

Navigating a Sample Design

Figure 2-9 shows a sequential multiplier, `SEQ_MULT`. Figure 2-10 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 54 for detailed descriptions.

Figure 2-9 SEQ_MULT Design

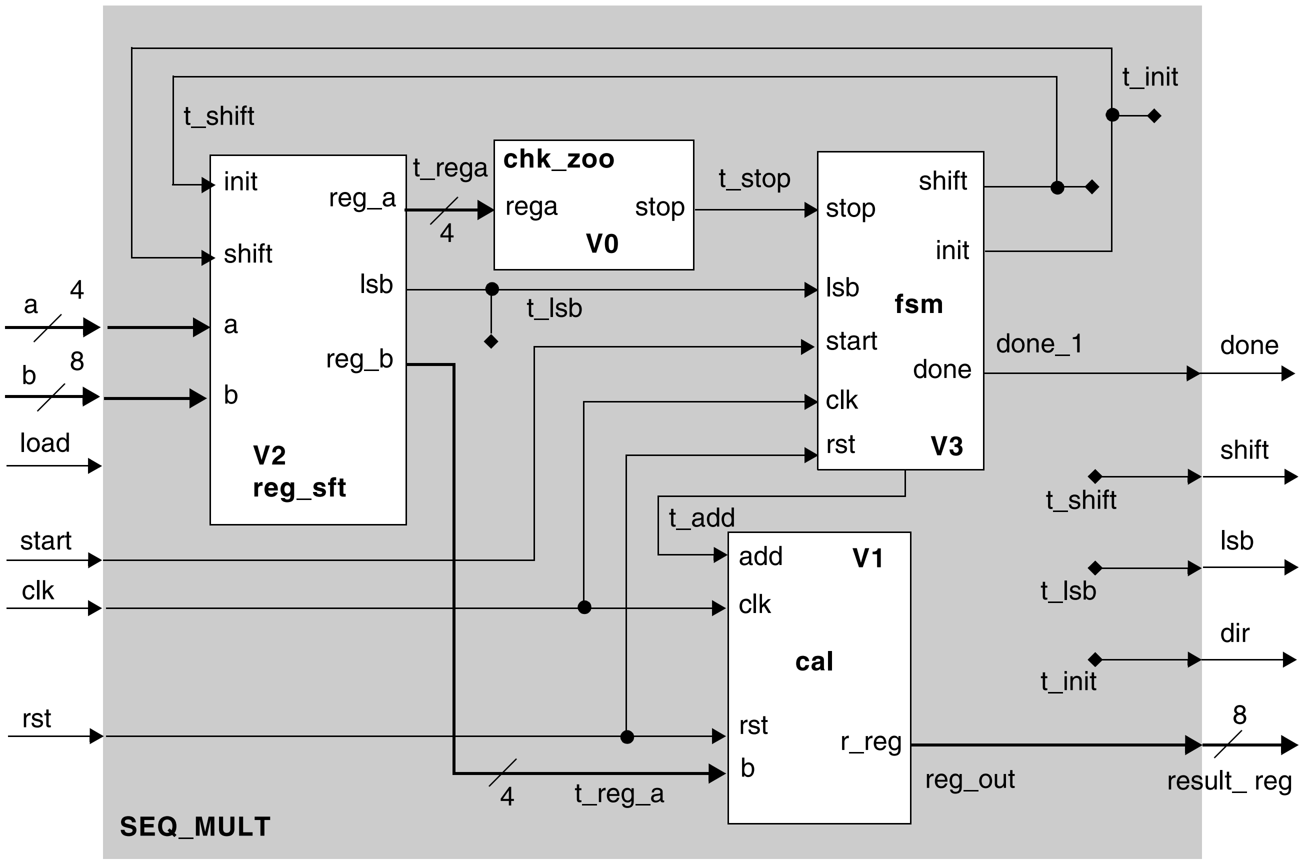
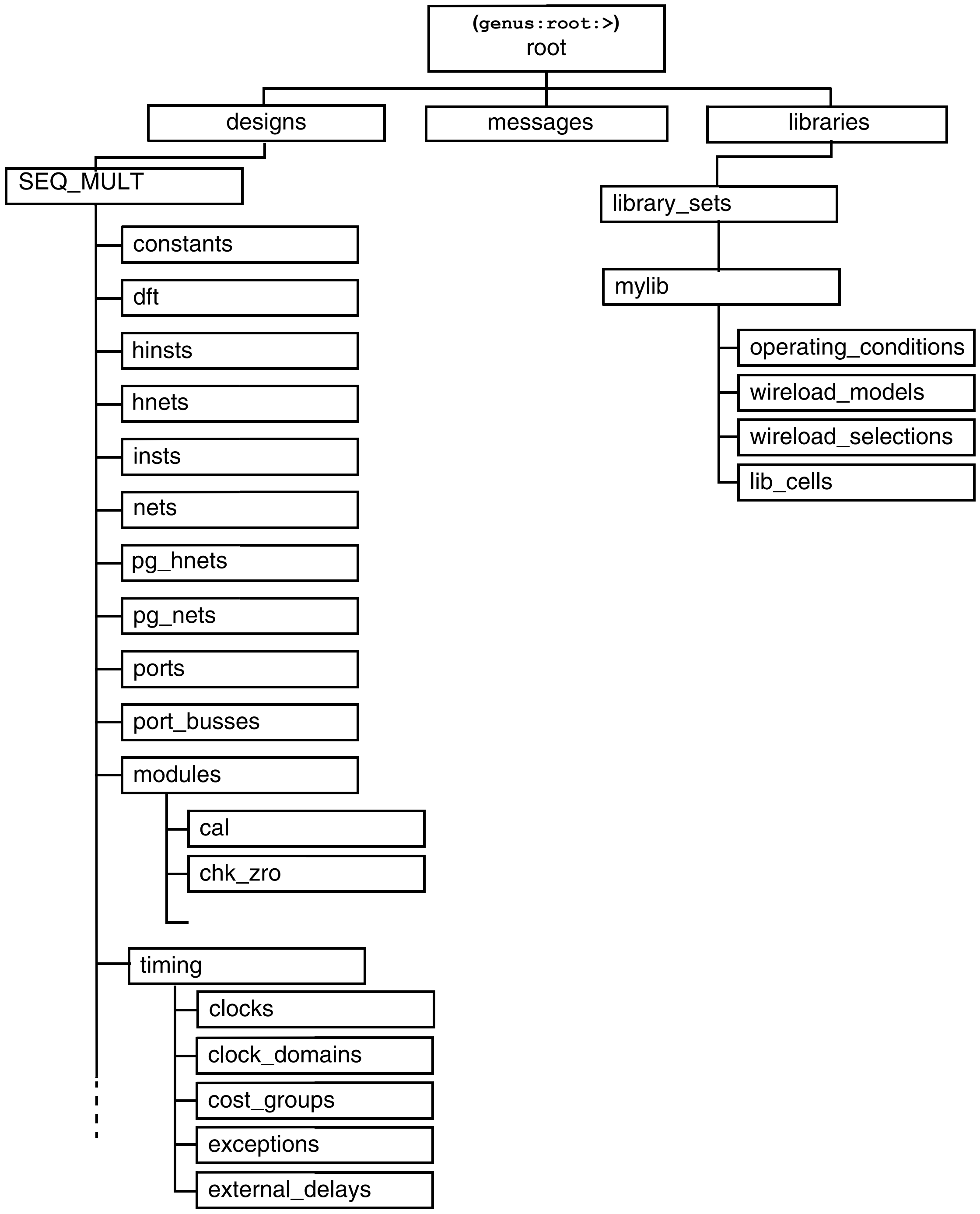


Figure 2-10 Top-Level View for SEQ_MULT Design



Modules

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

```
genus:root: 12> vcd des*/*/modules/*
genus:design:SEQ_MULT.modules 13> vls -l
```

And returns the following:

```
Total: 3 items

./
cal/          (module)
chk_zro/      (module)
fsm           (module)
reg_sft/      (module)
```

Input and Output Ports

To see the top level input and output ports, go to the `ports` directories (Figure 2-10 on page 95).

➤ The following command finds the input and output ports with the `get_db` command:

```
genus:root: 16> get_db [get_db designs SEQ_MULT] .ports *
```

The command displays information similar to the following:

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

See *port* in the Working in the designs Directory on page 57 for a detailed description of the `ports` directory.

Hierarchical Instances

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

```
/designs/SEQ_MULT/hinsts/
```

The `/designs/SEQ_MULT/hinsts` directory contains all the hierarchical instances in the `SEQ_MULT` top level design (see Figure 2-11).

Sequential Instances

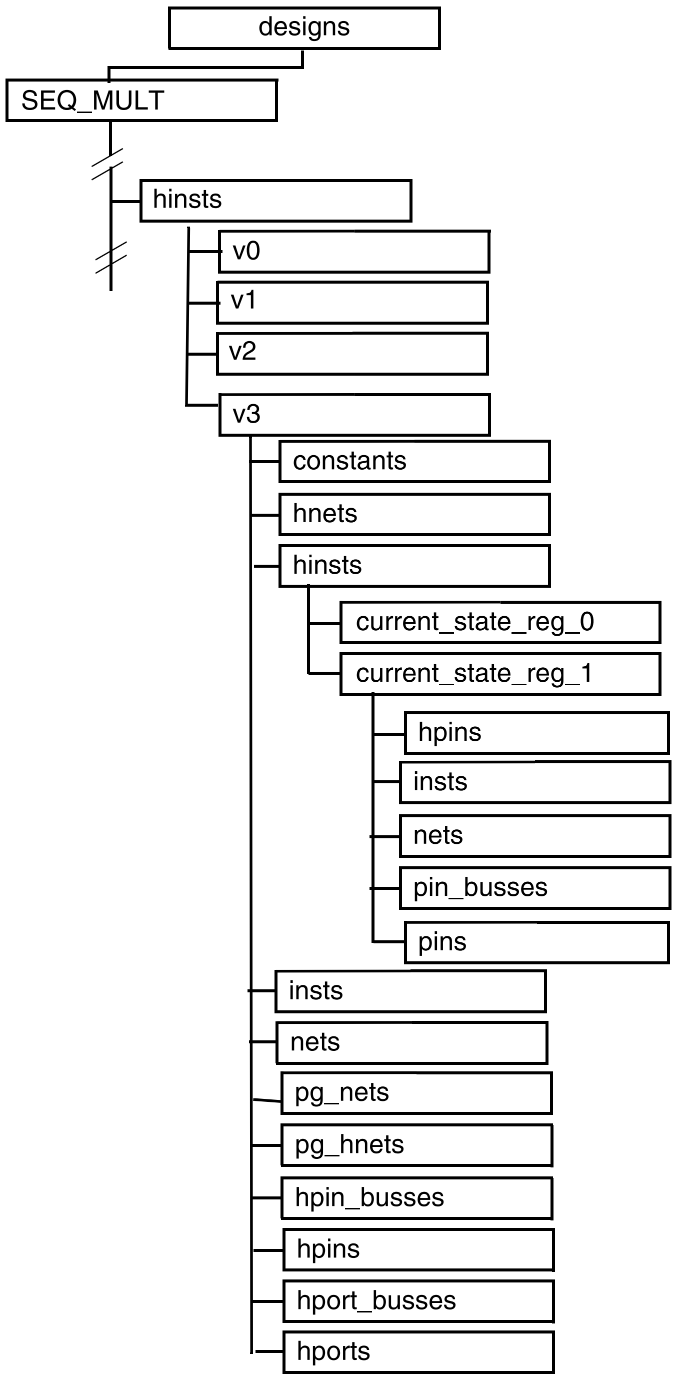
Any sequential instances in the top-level design are listed in the `/designs/SEQ_MULT/instances_seq` directory shown in Figure 2-10. The `SEQ_MULT` design does not have any sequential instances at this level. However, it does have some at a lower level in the hierarchy, as shown in Figure 2-11.

Lower-level Hierarchies

Figure 2-11 on page 98 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.

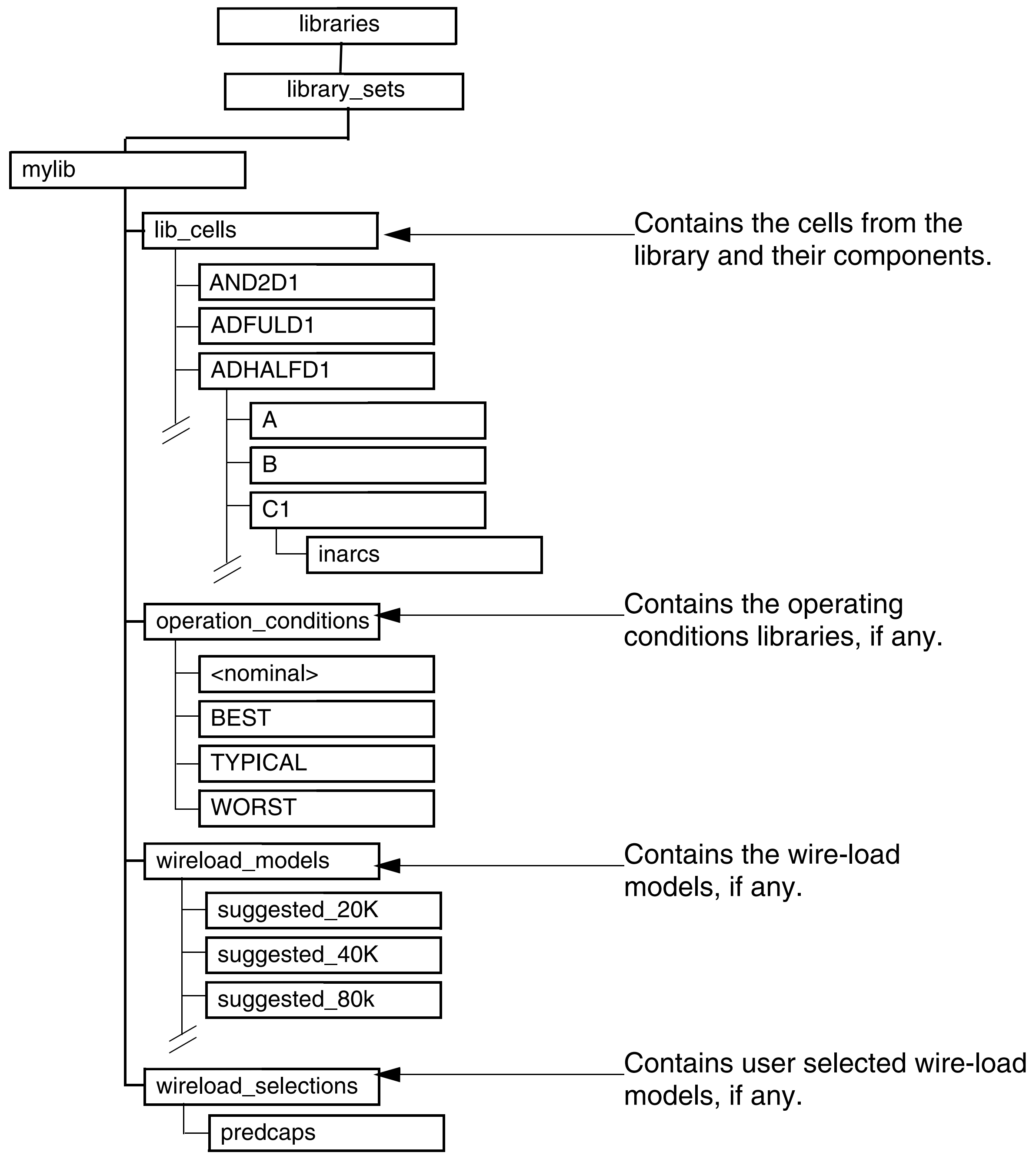
Figure 2-11 Low-Level for SEQ_MULT Design



Library Information

Figure 2-12 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 2-12 Libraries Directory Structure


libraries
library_sets
mylib
lib_cells
AND2D1
ADFULD1
ADHALFD1
A
B
C1
inarcs
operation_conditions
<nominal>
BEST
TYPICAL
WORST
wireload_models
suggested_20K
suggested_40K
suggested_80k
wireload_selections
predcaps
Contains the cells from the library and their components.
Contains the operating conditions libraries, if any.
Contains the wire-load models, if any.
Contains user selected wire-load models, if any.

See Working in the Library Directory on page 68 for a detailed description of this directory.

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 Genus commands:

Example 2-8 Saving the Design Information Hierarchy

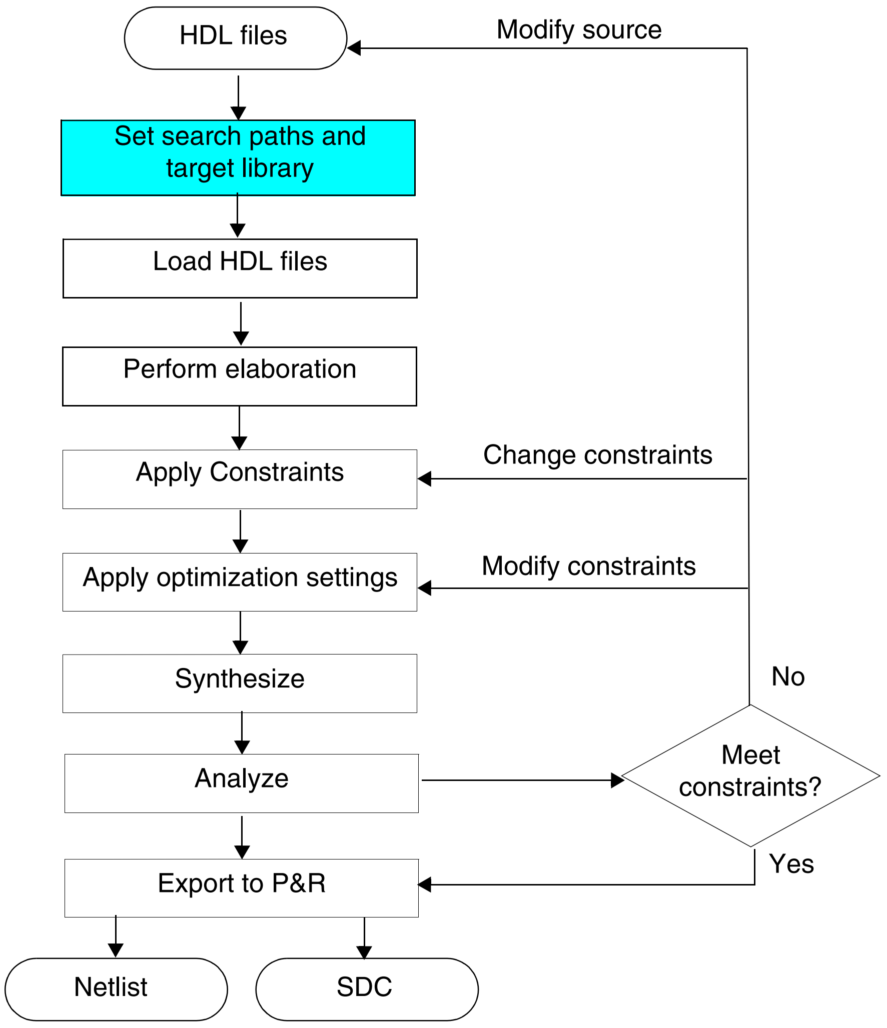
```
proc vdir_save {args} {
  set pov [parse_options [calling_proc] fil $args \
    "-detail bos include detailed info" detail \
    "drs root vdir from which to start saving data" vdir]

  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
  }
}
```

3

Using the Libraries

- Overview on page 102
- Tasks on page 103
    - Specifying Explicit Search Paths on page 103
    - Specifying Implicit Search Paths on page 104
    - Specifying Settings that Influence Handling of Library Cells on page 105
    - Setting the Target Technology Library on page 105
    - Preventing the Use of Specific Library Cells on page 107
    - Forcing the Use of Specific Library Cells on page 107
    - Working with Liberty Format Technology Libraries on page 108
    - Using Voltage Scaling on page 109
    - Library Scaling in Multi-Mode Multi-Corner (MMMC) Flow on page 111
- Troubleshooting on page 112
    - Cells Identified as Unusable on page 112
        - Possible Reasons for a Cell to be Marked Unusable on page 112
        - Effect of Unusable Cells on the Flow on page 114
    - Cells Identified as Timing Models on page 117
        - Possible Reasons for a Cell to be Marked as Timing Model on page 117

Overview



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

Note: In the physical flows, you also need to load LEF libraries and parasitic information. For more information on these tasks and on the physical flows, refer to *Genus Physical Guide*.

Tasks

- Specifying Explicit Search Paths on page 103
- Specifying Implicit Search Paths on page 104
- Specifying Settings that Influence Handling of Library Cells on page 105
- Setting the Target Technology Library on page 105
- Preventing the Use of Specific Library Cells on page 107
- Forcing the Use of Specific Library Cells on page 107
- Working with Liberty Format Technology Libraries on page 108

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 Genus 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:

- `init_lib_search_path`

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

- `script_search_path`

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

- `init_hdl_search_path`

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

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

```
set_db init_lib_search_path path
set_db script_search_path path
set_db init_hdl_search_path path
```

where *`path`* is the full path of your target library, script, or HDL file locations. These need to be set before reading any libraries.

The slash ("/") in these commands refers to the root-level Genus object that contains all global Genus 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 Genus to search for library files both in the current directory (.) and in the path `/home/customers/libs`:

```
set_db init_lib_search_path {. /home/customers/libs}
```

To see all of the current settings, type:

```
vls -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 Genus 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, Genus recursively searches the specified paths and sets a false path between all clock objects named `clk1` and `clk2`.

```
set_db path ". / /libraries/* /designs/*"
set_false_path -from clk1 -to clk2
```

Genus 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`, Genus 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/*
```

Specifying Settings that Influence Handling of Library Cells

Some attributes influence the handling of some library cells during mapping and incremental optimization. These attributes must be set before the library is read in. Following is a list of such attributes:

- `exact_match_seq_async_ctrls`
- `exact_match_seq_sync_ctrls`
- `lbr_respect_async_controls_priority`
- `lbr_seq_in_out_phase_opto`
- `map_to_master_slave_lssd`

Setting the Target Technology Library

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

➤ To specify a single library:

```
genus:root: 1> set_db library lib_name.lib
```

Genus will use the library named `lib_name.lib` for synthesis. Genus 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:

```
set_db library /usr/local/files/lib_name.lib
```

➤ To specify a single library compressed with gzip:

```
set_db library lib_name.lib.gz
```

➤ To append libraries:

```
set_db 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. Genus uses the operating and nominal conditions, thresholds, and units from the first library specified. If you specify libraries sequentially, Genus uses only the last one loaded.

In the following example Genus uses only `lib_name2.lib` as the target library:

```
set_db library lib_name.lib
set_db library lib_name2.lib
```

To specify multiple libraries using the `library` variable:

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

   ```
   set library {lib_name1.lib lib_name2.lib}
   ```

   When listing files, use the Tcl list syntax: `{entry1 entry2 ...}`.

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

   ```
   set_db library $library
   ```

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

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

```
set_db library { lib_name.lib lib_name2.lib }
```

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_db cell_name(s) .avoid {true | false}
```

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

```
set_db { lib_cell:nlc18_custom/snl_mux21_prx* } .avoid true
set_db { lib_cell:nlc18/*nsdel } .avoid true
```

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

```
set_db hdl_component:CW/CW_ashiftr .avoid true
```

Forcing the Use of Specific Library Cells

You can instruct Genus 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:

```
set_db libcell_name .preserve false
```

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

```
set_db libcell_name .avoid false
```

Working with Liberty Format Technology Libraries

Source code for technology libraries is written in the `.lib` (Liberty) 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`
- `lib_cell`
- `lib_pin`
- `lib_arc`
- `wireload`
- `operating_condition`

The following examples demonstrate the uses of the `liberty_attributes` string:

```
get_liberty_attribute “current_unit” [get_db libraries *lib1*]
    1ma
get_liberty_attribute “area” [get_db lib_cells *nr23d4]
    4
get_liberty_attribute “cell_footprint” [get_db lib_cells *nr23d4]
    aoi_3_5
get_liberty_attribute “function” [get_db lib_pins *nr23d4/zn]
    (a1'+a2'+a3')
get_liberty_attribute “timing_type” [get_db lib_arcs *invtd1/zn/en_d50]
    three_state_disable
```

Using Custom Pad Cells

Genus 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).

Genus identifies pad cells through the Liberty attributes `is_pad` (for `lib_pins`) and `pad_cell` (for `lib_cells`). 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 `lib_pin` description and the `pad_cell` construct on the `lib_cell` description.

Using Voltage Scaling

Genus supports voltage based delay scaling when two libraries that are characterized at two different voltages but at constant temperature and process, are provided. If you load two libraries characterized at P1-V1-T1 (process P1, voltage V1 and temperature T1) and P1-V2-T1 (process P1, voltage V2 and temperature T1), you can synthesize the design at a different operating voltage V3 (where V1 < V3 < V2). Currently Genus supports linear interpolation for voltage based NLDM delay scaling.

Use model

The `library` attribute will be used with an extra level of braces for using this feature.
For example:

```
set_db / .library {library1_PV1T.lib library1_PV2T.lib}
```

where, both library1_PV1T.lib and library1_PV2T.lib are expected have the same set of lib-cells and are characterized at P-V1-T and P-V2-T respectively.

If there are some libraries that are available for only one characterized voltage, you need to specify them as follows:

```
set_db / .library { {library1_PV1T.lib library1_PV2T.lib} {single_lib1.lib} \
{library2_PV1T.lib library2_PV2T.lib} {single_lib2.lib} …..…}
```

Flow when using a single library

If all libraries are characterized at P1-V1-T1, use:

```
set_db library {<list of libraries>}
```

Now we can change the voltage of active operating condition:

```
set_db [get_db / .active_operating_conditions] .voltage V2
........
..........
```

This `voltage` attribute needs to be set to the appropriate voltage value during voltage scaling to select the operating voltage.

Rest of the synthesis flow would be using delay numbers corresponding to P1-V2-T1

Currently Genus can scale the delay or power numbers for expected PVT parameters, given that the given libraries have all necessary `k_factors`.

Flow when using multiple libraries

In this flow, while setting the `library` attribute, the libraries that are characterized at two different voltages, which would be used for interpolation, must be grouped within an additional level of Tcl list structure to indicate the intended use to the library subsystem.

The libraries in this set can differ only in the characterization voltage, they must be identical to each other in all other aspects.

`Library*1.lib` are characterized at P1-V1-T1 and *`Library*2.lib`* are characterized at P1-V2-T1

```
set_db / .library { {library11.lib library12.lib} {library21.lib
library22.lib} ..... }
```

Now we can change the voltage of active operating condition:

```
set_db [get_db / .active_operating_conditions] .voltage V3
........
........
```

This `voltage` attribute needs to be set to the appropriate voltage value (V3 in this case) during voltage scaling to select the operating voltage.

Rest of the synthesis flow would be using delay numbers corresponding to P1-V3-T1.

In multiple library flow, you have to specify libraries in pairs within an additional level of Tcl list structure to trigger library scaling feature. Now Genus will determine that the PVT parameters among the libraries in one set differ only in voltage. Refer to Create Library Domains in *Genus Synthesis Flows Guide*, to understand how to create a library domain.

Hence, for multiple library domains, the flow is:

```
set_db library_domain:Lib-Domain1 .library { {library11.lib library12.lib} \
{library21.lib library22.lib} }
set_db [get_db library_domain:Lib-Domain1 .active_operating_conditions] \
.voltage V3
set_db library_domain:Lib-Domain2 .library { {library11.lib library12.lib} \
{library21.lib library22.lib}
set_db [get_db library_domain:Lib-Domain2 .active_operating_conditions] \
.voltage V4
```

For this, Genus will do some sanity checks:

- ❑ The library cells in those two libraries should be identical.
- ❑ The libraries in each of the sets should be characterized at the same PVT.
- ❑ The two libraries in all the sets should differ in one and only one PVT parameter (that is, voltage).

Library Scaling in Multi-Mode Multi-Corner (MMMC) Flow

In MMMC flow, Genus allows the MMMC view definition file to specify operating conditions (opconds) with a voltage value that is different from the existing PVT (Process -P Voltage-V Temperature -T) values specified in the library files. This leads to three use-cases in Genus.

Flow when using multiple libraries

The MMMC view definition file should be read using the `read_mmmc` command in Genus and the libraries in this set differ only in the characterization voltage (that is, P1-V1-T1 and P1-V2-T1).

Currently, Genus supports maximum two libraries for scaling.

For example, if *`Library*1.lib`* are characterized at P1-V1-T1 and *`Library*2.lib`* are characterized at P1-V2-T1, the flow is:

```
read_mmmc <viewDefinition>.tcl
create_library_set -timing  {  {library11.lib library12.lib} {library21.lib
library22.lib} .... }
```

Now, we change the voltage of active operating conditions:

```
create_opcond -name <oc_name> -process P1 -voltage V3 -temperature T1
```

Important

The library-pairs used in the `create_library_set` command should be specified in a way that within the pair, the library which is characterized at lower voltage should be specified first. For example, { {lib_0.6 lib_0.7} {lib_0.55 lib_0.7} } is correct. But specifying { {lib_0.6 lib_0.7} {lib_0.7 lib_0.55} } can lead to errors.

Rest of the synthesis flow will use the delay numbers corresponding to P1-V3-T1.

For this, Genus will do some sanity checks:

- ❑ The two libraries in all the sets should differ in one and only one PVT parameter (that is, voltage).
- ❑ The library pairs should be specified with ascending value for voltages.
- ❑ Only two library pairs are used for scaling.

Troubleshooting

Cells Identified as Unusable

The tool identifies cells as usable or unusable when it parses the libraries. Cells that are marked *unusable* will not be used during global mapping nor during incremental optimization, but can still be used for clock-gating, SRPG replacement, and scan mapping (except for those cells attributed with `dont_use`).

The tool reports the total number of cells and the number of unusable cells in the log file. If the `information_level` root attribute is set to 6 or higher, all unusable cells are listed, otherwise only 10 cells are listed. This is stated in the LBR-415 message. For example,

```
Library: 'coreseq_hvt_c35Fsc12mc_cln28hpmPtyp30V0900T125.lib', Total cells: '35',
 Unusable cells: '18'.
       List of unusable cells: 'L1L2SRPG_X1M_F12TH_C35 L1L2SRPG_X2M_F12TH_C35
L1L2SRPG_X4M_F12TH_C35 L1L2SRPG_X8M_F12TH_C35 L1SSRPG_X1M_F12TH_C35
L1SSRPG_X2M_F12TH_C35 L1SSRPG_X4M_F12TH_C35 L1SSRPG_X8M_F12TH_C35
MSFFSRPG_X0P99M_F12TH_C35 MSFFSRPG_X1M_F12TH_C35 ... and others.'
```

Possible Reasons for a Cell to be Marked Unusable

Note: The reason why a cell is considered unusable is stored in the `unusable_reason` `lib_cell` attribute.

The `lib_cell` can be marked unusable if:

1. One of the following attributes is set to `true:`
   - ❑ the `avoid` `lib_cell` attribute
   - ❑ the `preserve` `lib_cell` attribute
   - ❑ the `is_always_on` `lib_cell` attribute
2. The `dont_touch` Liberty cell attribute set to `true`.
3. The `dont_use` Liberty cell attribute set to `true`.
4. Has a negative area.
5. Has pins with direction `inout`.
6. Has no logical input pins.
7. Contains the `pin_opposite` attribute.

8. The `delay_model` Liberty attribute was set to `table_lookup`, but the `lib_cell` has no timing defined.
9. Is a sequential cell but:
   - ❑ has either no clock pin or no output pin defined as sequential output pin.
   - ❑ has an asynchronous preset or asynchronous clear pin referenced in the `data_in` or `next_state` attribute.
   - ❑ has more than one clock pin while the `lib_cell` is not a master slave flip-flop nor a state retention cell.
   - ❑ is missing either the setup arc from the clock to the data pin or the `clock_edge` arc from the clock to the output pin.
   - ❑ does not have a complete timing specification for the clock_edge arc.
   - ❑ has combinational arcs from or to a scan-only pin.
10. Is a master slave flip-flop that has both the master clock and slave clock triggered by the same clock edge of the same clock pin.
11. Is a clock-gating cell (combinational or integrated).
12. Is a multibit flop or a multibit latch.

    For more information refer to Clock-Gating Cell Specification in the *Genus Library Guide*.
13. Is a multibit cell with bussed pins.
14. Is a clocked LSSD scan flip-flop but the `cell` group does not have a `statetable` group that describes the cell function.

    For more information, refer to Scan Cell Requirements in the *Genus Library Guide*.
15. Is present in the LEF library but not in the Liberty library. In this case, the `lib_cell` is called a physical-only cell.
16. Is not present in the LEF library or its area in the LEF library is zero.
17. Is a tristate cell and has a constant value for the `three_state` attribute.
18. Is a tristate cell and has a constant value for the output pin function.
19. Is a timing model.

    Refer to Cells Identified as Timing Models on page 117 for more information on when a `lib_cell` is considered a timing model.

Effect of Unusable Cells on the Flow

Cells that are marked unusable cannot be inferred during global mapping or incremental optimization. To map a design the libraries must have at least one usable inverter, basic gate, latch and flip-flop. Look for the following messages in the log file:

Message-ID	Title	Help
LBR-171	Cannot perform synthesis because libraries do not have usable inverters.	Inverters are required for mapping. Ensure that the loaded libraries contain at least one usable inverter.
LBR-172	Cannot perform synthesis because libraries do not have usable basic gates.	At least one usable two-input and/or/nand/nor gate (modulo inversion at inputs) is required for mapping. Ensure that the loaded libraries contain at least one such cell.
MAP-1	Unable to map design without a tristate buffer or inverter.	Check the libraries for necessary tristate cell. If the tristate cell exists in the library, query using the 'unusable_reason' attribute on the libcell to know why the tool marked it as unusable.
MAP-2	Unable to map design without a suitable flip-flop.	Check the libraries for necessary flop cell. If the flop cell exists in the library, query using the 'unusable_reason' attribute on the libcell to know why the tool marked it as unusable
MAP-3	Unable to map design without a suitable latch.	Check the libraries for necessary latch cell. If the latch cell exists in the library, query using the 'unusable_reason' attribute on the libcell to know why the tool marked it as unusable.
MAP-19	Specified libcell is either avoided or not usable.	Check if the 'avoid' libcell attribute is set to 'true'. If so, change the attribute value to 'false'. Check if the 'usable' libcell attribute is set to 'false'. If so, remove the cell from the 'map_to_register' attribute value.
MAP-20	Specified libcell is avoided.	Check if the 'avoid' libcell attribute is set to 'true'. If so, change the attribute value to 'false'.

Unusable cells can affect the **DFT flow** including mapping to scan, scan connection, scan compression.

Message-ID	**Title**	**Help**
DFT-112	Failed to connect scan chains.	The library has no flop or latch that is considered usable. A library cell is considered not usable if it has a 'dont_use' or a 'dont_touch' attribute set to 'true' in the .lib files. Set the attribute 'preserve' to false on the library cell and set the attribute 'avoid' to false on the library cell to make a flop or latch usable for lockup insertion.
DFT-227	Failed to compress scan chains.	The library has no latch that is considered usable. A library cell is considered not usable if it has a 'dont_use' or a 'dont_touch' attribute set to 'true' in the .lib files. Set the attribute 'preserve' to false on the library cell and set the attribute 'avoid' to false on the library cell to make a latch usable for lockup insertion.
DFT-510	Could not find a scan-equivalent cell.	A scan-equivalent cell was not found. A potential scan-equivalent library cell is considered not usable if it has a 'dont_use' or a 'dont_touch' attribute set to true in the .lib files. In this case, set the attribute 'preserve' to false on the scan library cell and set the attribute 'avoid' to false on the scan library cell to make the cell usable. A potential scan-equivalent library cell is excluded if it does not follow the Scan Cell Requirements described in the 'Library Guide'. This requires fixing the library.

Unusable cells can affect the **Low Power flow** including clock-gating, power analysis, level-shifter insertion, isolation insertion.

Message-ID	**Title**	**Help**
LBR-100	Unusable clock gating integrated cell.	Check to make sure that clock gating cell has all its pin attributes set correctly.
LBR-101	Unusable clock gating integrated cell.	To use the cell in clock gating, Set cell attribute 'dont_use' false in the library.

LBR-201	Invalid level shifter pin. The level shifter is not usable.	Make sure the signal level attribute for the pin is properly set.
LBR-301	Unusable isolation cell.	To use the cell for isolation cell insertion, set cell attribute 'dont_use' and 'dont_touch' to 'false' in the library.
PA-9	Could not perform a meaningful RTL power analysis.	Make sure that you have a library that contains the above specified cell or cells to create power models for unmapped gates in the netlist. A library cell is considered not usable if it has a 'dont_use' or a 'dont_touch' attribute set to 'true' in the .lib files. In this case, use 'set_attribute preserve false <libcell>' and 'set_attribute avoid false <libcell>' to make the cell usable.

Unusable cells can affect timing analysis.

Message-ID	**Title**	**Help**
TIM-30	Could not perform a meaningful RTL delay analysis.	Make sure that your library contains at least one inverter and one 2-input library cell to create timing models for unmapped gates in the netlist. A library cell is considered not usable if it has a 'dont_use' or a 'dont_touch' attribute set to 'true' in the .lib files. In this case, use 'set_attribute preserve false <libcell>' and 'set_attribute avoid false <libcell>' to make the cell usable.

Cells Identified as Timing Models

Some cells (such as Liberty models of RAMs and complex IP) are timing models by design. In other cases, modeling inconsistencies mentioned below could make standard cells to be treated as timing models.

Possible Reasons for a Cell to be Marked as Timing Model

1. The cell function or the function of one of its output pins is either missing, too complex, or has an invalid pin name.

Message-ID	Title	Help
LBR-41	An output library pin lacks a function attribute.	If the remainder of this library cell's semantic checks are successful, it will be considered as a timing-model (because one of its outputs does not have a valid function.
LBR-42	Could not parse a library pin's function statement.	Check the pin's function statement in the library source.
LBR-140	Sequential cell function definition makes cell unusable.	The sequential cell cannot be inferred because its function is unknown.
LBR-146	Invalid pin name used.	

2. The sequential `lib_cell` contains an invalid pin name in the `ff` or `latch` group, or is missing some of the group attributes, such as `data_in`, `enable`, `clocked_on`, `next_state` and so on.

Message-ID	Title	Help
LBR-41	An output library pin lacks a function attribute.	If the remainder of this library cell's semantic checks are successful, it will be considered as a timing-model (because one of its outputs does not have a valid function.
LBR-42	Could not parse a library pin's function statement.	Check the pin's function statement in the library source.
LBR-146	Invalid pin name used.	

3. The `lib_cell` is a sequential cell that has more than one setup arc while the `lib_cell` is not a master slave flip-flop nor a state retention cell.

Message-ID	Title	Help
LBR-152	Pin has more than one setup arc.	Pin should not have more than one setup arc. Otherwise, the library cell will be treated as a timing-model.

4. Pins in the `next_state` function have no incoming setup arc defined, or have an outgoing setup or clock edge arc defined.

Message-ID	Title	Help
LBR-8	Found an outgoing setup or clock edge timing arc for next_state library pin.	Pin used in a next_state function should not have an outgoing setup or clock edge arc. Otherwise, the library cell will be treated as a timing model.
LBR-34	Missing an incoming setup timing arc for next_state library pin.	Pin used in a next_state function must have an incoming setup timing arc. Otherwise, the library cell will be treated as a timing-model.

5. A sequential cell has a setup arc for a pin that does not appear in the `next_state` function.

Message-ID	Title	Help
LBR-151	Pin with a setup timing arc is not in the support set of the next-state function.	Pin with a setup timing arc must be in the support set of the next-state function. Otherwise, the library cell will be treated as a timing-model.

6. A combinational cell has also sequential arcs or a sequential cell has also combinational arcs:

Message-ID	Title	Help
LBR-75	Detected both combinational and sequential timing arcs in a library cell.	The library cell will be treated as a timing-model. Make sure that the timing arcs and output function were described correctly. If the cell was intended to have dual-functionality this may be ok, but this cell cannot be unmapped or automatically inferred.
LBR-76	Detected both combinational and sequential timing arcs in a library cell.	The library cell will be treated as a timing-model. Make sure that the timing arcs and output function were described correctly. If the cell was intended to have dual-functionality this may be ok, but this cell cannot be unmapped or automatically inferred.

7. The output pin of the `lib_cell` has no incoming timing arcs.

Message-ID	Title	Help
LBR-158	Libcell will be treated as a timing model.	Ensure that the relevant timing arcs are defined in the Liberty model of the libcell.

8. The `lib_cell` is a master slave flip-flop that has both the master clock and slave clock triggered by the same clock edge of the same clock pin.

Message-ID	Title	Help
LBR-414	Sequential cell cannot be treated as MSFF.	The libcell will be marked as timing model. To make sure that the sequential cell is treated as a master-slave flip-flop, use either different clocks or different clock edges of the same clock for the master and slave clocks.

9. A pin in the `next_state` of the master slave flop is missing the setup arc to the clock pin of the slave flop.

Message-ID	**Title**	**Help**
LBR-34	Missing an incoming setup timing arc for next_state library pin.	Pin used in a next_state function must have an incoming setup timing arc. Otherwise, the library cell will be treated as a timing-model.

10. The `lib_cell` is a combinational cell with a clock pin.
11. The `lib_cell` has disabled arcs.

    Note: The tool will not mark the `lib_cell` as a timing model if

    - The `lib_cell` is a state-retention cell and the only disabled arc is from the retention pin to the output pin
    - The `lib_cell` is a master slave flip-flop, and the only disabled arc is to the test_scan_in pin.

    Note: An arc becomes disabled when you set the `enable` attribute on the lib_arc to `false`.
12. The `lib_cell` is a latch with an incorrect clock pin specification.

Message-ID	**Title**	**Help**
LBR-411	Found incorrect pin specification.	If the pin name is specified within double quotes, extra blanks and parentheses are not allowed inside the double quotes.

13. The lib_cell has the `pad_cell` Liberty attribute set to `true`
14. The lib_cell is a basic gate that is missing the `cell_rise` and `cell_fall` groups, or `rise_propagation` and `fall_propagation` groups in the `timing` group of the output pin.
15. The lib_cell is a sequential cell with a three-state output.

16. The `lib_cell` is a sequential cell and it has either a clock pin, asynchronous preset or asynchronous clear pin with a complex function.

Message-ID	Title	Help
LBR-141	Clock function definition makes cell unusable.	The sequential cell cannot be inferred because its clock function is unknown.
LBR-142	Async-clear function definition makes cell unusable.	The sequential cell cannot be inferred because its async-clear function is unknown.
LBR-143	Async-preset function definition makes cell unusable.	The sequential cell cannot be inferred because its async-preset function is unknown.

17. The `lib_cell` is a sequential cell that has an asynchronous preset or asynchronous clear pin referenced in the `data_in` or `next_state` attribute.

Message-ID	Title	Help
LBR-413	Improperly defined sequential function.	

18. The output pin of the lib_cell is missing timing arcs for the input pins listed in its `function` attribute.

19. The lib_cell has a timing arc from an internal pin to an inout pin or an output pin.

20. The `lib_cell` is a scan cell, with the cell function defined using a state table and the `test_cell` function defined using an ff group, and has an arc from the scan enable pin to an output pin other than the scan output pin with the value of the `timing_sense` attribute set to `non_unate`.

21. The pins of the `lib_cell` have the `clock_gate_clock_pin`, `clock_gate_enable_pin`, and `clock_gate_out_pin` attributes, but either the cell is missing the `clock_gating_integrated_cell` attribute, or has an invalid function or an invalid statetable.

Message-ID	Title	Help
LBR-98	Incorrect gating function for combinational clock-gating integrated cell.	The combinational clock-gating integrated cell must be either an AND or OR type gate.
LBR-99	Cannot process state table for clock-gating integrated cell.	The input node names in the state table must match the cell input pin names.

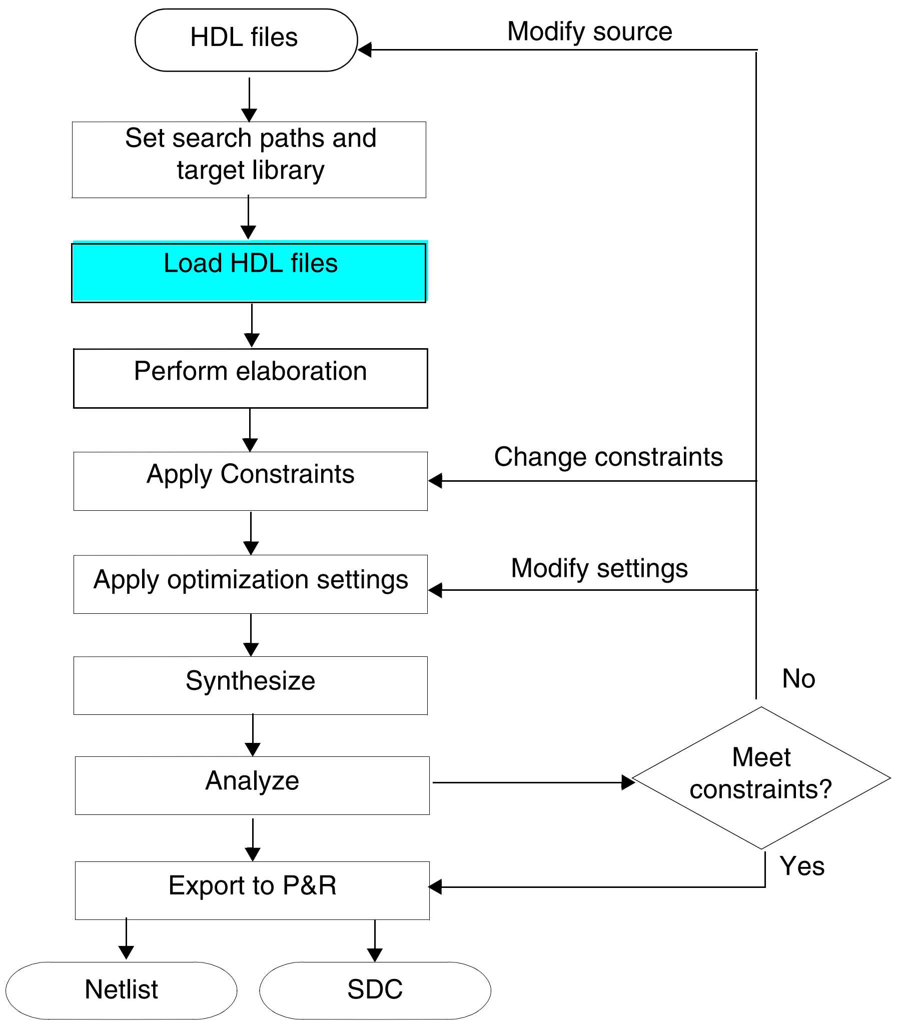
4

Loading Files

- Overview on page 124
- Tasks on page 125
    - Updating Scripts through Patching on page 125
    - Running Scripts on page 126
    - Reading HDL Files on page 127
    - Reading Verilog Files on page 133
    - Reading VHDL Files on page 141
    - Reading Designs with Mixed Verilog and VHDL Files on page 145
    - Reading and Elaborating a Structural Netlist Design on page 147
    - Reading a Partially Structural Design on page 148
    - Keeping Track of Loaded HDL Files on page 150
    - Importing the Floorplan on page 150

Overview

This chapter describes how to load HDL files into Genus.



Tasks

- Updating Scripts through Patching on page 125
- Running Scripts on page 126
- Reading HDL Files on page 127
- Reading Verilog Files on page 133
- Reading VHDL Files on page 141
- Reading Designs with Mixed Verilog and VHDL Files on page 145
- Keeping Track of Loaded HDL Files on page 150

Updating Scripts through Patching

The patch mechanism in Genus allows you to potentially fix a problem, in Tcl, without waiting for the next official release. Also, in the last stages of a tapeout, it can address targeted issues without absorbing an entire new feature set that accompanies a new release. Specifically, this mechanism is 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 Genus banner will show the patch ID as part of the version. For example, if patches 1 and 3 are applied to version `15.20`, the banner would show the version as being 15`.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

Genus 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 `genus` command to start Genus and run your scripts immediately:

  ```
  unix> genus -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 Genus as a background process and execute a script by typing the following command from the UNIX command line:

  ```
  unix> genus < script_file_name &
  ```

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

  ```
  genus:root: 1> include script_file1 script_file2 ...
  ```

  or:

  ```
  genus:root: 2> source script_file1 script_file2 ...
  ```

For a sample script file, see “Simple Synthesis Template” on page 329.

For information on using interactive GUI commands so that you can write your own scripts to interact with the GUI and to add features that are not part of the normal installation, see *Genus GUI Guide* for detailed information.

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 Genus. When you issue a `read_hdl` command, Genus reads the files and performs syntax checks.

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

```
read_hdl [-langauage {v2001 | v1995 | sv | vhdl} ]
         [-library library_name][-f]
         | -netlist] [-mixvlog]
         [-define macro=value] .. file_list
```

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

- List 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
  ```

  or

  ```
  read_hdl -f optionfile
  ```

  Note: The host directory where the HDL files are searched for is specified using the `init_hdl_search_path` root attribute.

See Specifying HDL Search Paths on page 132 for more information.

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

  ```
  read_hdl -language 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 `init_hdl_search_path` attribute to make the TCL scripting more concise. See Specifying HDL Search Paths on page 132 for an example.

- Use the -language option to specify the language for parsing the design files.
  - Use the `v1995` option to specify the Verilog IEEE Std 1364-1995 compliance. 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`, `2008` or the `1993` version, but do not read in a design that has a mixture of these 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 `-f` option to specify the name of the list file from the simulation environment. For details, refer Reading Designs in Simulation Environment in *Genus Synthesis Flows Guide*.
- Use the `-mixvlog` option to ask Genus to automatically read Verilog source file in Verilog1995, Verilog2001, or SystemVerilog standard languages, according to the file extension. It will also consider any other extension as specified by the `hdl_set_vlog_file_extension` command.

  The default file extensions for Verilog standards are as follows:
  Verilog 1995: `.v95`, `.v95p`
  Verilog 2001: `.v`, `.v2k`, `.vp`
  System Verilog: `.sv`, `.svp`

  Examples:

  ```
  read_hdl -mixvlog bot.v test.sv
  ```

  The above command will automatically parse `bot.v` in `v2001` format and `test.sv` in System Verilog format.

  ```
  read_hdl -mixvlog -f optionfile
  ```

  Another example with `hdl_set_vlog_file_extension`

  ```
  hdl_set_vlog_file_extension -v2001 .v1
  hdl_set_vlog_file_extension -sv .v2 .v3
  read_hdl -mixvlog bot.v1 test.v2
  ```

The above command will automatically parse `bot.v1` in v2001 language and `test.v2` in System Verilog.

- Use the `-library` option to specify the name of the Verilog or VHDL library in which the definitions will be stored.

  A virtual directory with the library name will be created in the `hdl_libraries` directory of the design hierarchy if it does not already exist. The library definitions remain in effect until elaboration, after which all library definitions are deleted. By specifying Verilog and VHDL library names, you can read in multiple Verilog modules and VHDL entities (and VHDL packages) with the same name without overwriting each other.

  The following example loads a single VHDL file and specifies a single VHDL library:

  ```
  read_hdl -language vhdl -library lib1 test1.vhdl
  ```

  The following commands read in two Verilog files that each contain a Verilog module with the same name (`compute`) but with different functionality. To store both definitions, the `-library` option indicates in which library to store the definition.

  ```
  read_hdl -language v2001 -library lib1 test_01_1.v
  read_hdl -language v2001 -library lib2 test_01_2.v

  vls /hdl_libraries/lib1/architectures/

  /hdl_libraries/lib1/architectures:
  ./ compute/

  vls /hdl_libraries/lib2/architectures/

  /hdl_libraries/lib2/architectures:
  ./ compute/
  ```

- Use the `-netlist` option to read structural Verilog 1995 files.

  In the following example, the `-v1995` option is ignored. Both `rtl.v` and `struct.v` are parsed in the structural mode.

  ```
  read_hdl -language v1995 rtl.v -netlist struct.v
  ```

Follow these guidelines when reading HDL files:

- Read files containing macro definitions before the macros are used.
- Using the `-language` option 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 compressed gzip file. For example:

  ```
  read_hdl -language vhdl sample.vhdl.gz.
  ```

  Genus detects the `.gz` file suffix and automatically unzips the input file.

Specifying the HDL Language Mode

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

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

Default: `v2001`

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

Note: Using the `-language` option with the `read_hdl` command will override the setting of the `hdl_language` attribute.

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

Table 4-1 lists the language modes and the various ways you can use the commands and attributes to set these modes.

Table 4-1 Specifying the Language Mode

Language Mode	**Command**

`Verilog-1995`	`read_hdl -language v1995 design.v`
	or
	`set_db hdl_language v1995`
	`read_hdl design.v`

`Verilog-2001`	`read_hdl -language v2001 design.v`
	or
	`set_db hdl_language v2001`
	`read_hdl design.v`

`SystemVerilog`	`read_hdl -language sv design.v`
	or
	`set_db hdl_language sv`
	`read_hdl design.v`

Language Mode	Command

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

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

Specifying HDL Search Paths

HDL files may not be located in the current working directory. Use the `init_hdl_search_path` attribute to tell Genus 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, Genus goes to these directories to look for it.

➤ Specify a list of UNIX directories where Genus 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_db init_hdl_search_path {../location_of_top ../location_of_sub}
read_hdl top.v sub.v
```

Default: `set_db init_hdl_search_path .`

If this attribute carries multiple UNIX directories, the way Genus 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 `init_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 `init_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_db init_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.
- 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 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 4-1.

Example 4-1 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 4-2.

Example 4-2 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, the following command reads the Verilog file shown in Example 4-3:

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

Example 4-3 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 4-4.

Example 4-4 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 is 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 remodel Example 4-4 using the modeling style, shown in Example 4-5.

Example 4-5 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 4-6.

Example 4-6 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, then 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.

Define statements are persistent across all the files read in by a single `read_hdl` command. If the `` `define `` statements are contained in a separate "header" file, then read that header file first to apply it to all the subsequent Verilog files.

For example, the following command apply the `` `define `` statements in `header.h` to `file1.v`, `file2.v`, and `file3.v`:

```
read_hdl "header.h file1.v file2.v file3.v"
read_hdl "file4.v"
```

Since `file4.v` is read with a separate `read_hdl` command, the `` `define `` statements in the `header.h` file are not applied to `file4.v`.

If multiple `read_hdl` commands are used to load the HDL files, then 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 with the `` `define FUNC 2`` statement
- A `test0.v` file, as shown in Example 4-7

Example 4-7 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 `test1.v` file, as shown in Example 4-8.

Example 4-8 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 `test2.v` file, as shown in Example 4-9.

Example 4-9 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
```

Using the following sequence of commands, with multiple `read_hdl` commands:

```
set_db library tutorial.lib
set_db init_hdl_search_path .
read_hdl -language sv test0.v test1.v
read_hdl -language sv test2.v
elaborate
write_hdl -g
```

If the `test1.v` file is affected by the macro definition in the `test.h` file, but the `test2.v` file is not, then Example 4-10 shows the generated netlist:

Example 4-10 Generated Netlist for Verilog Macros Using Multiple read_hdl Commands

```
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
```

Using the following sequence of commands, with only one `read_hdl` command:

```
set_db library tutorial.lib
set_db init_hdl_search_path .
read_hdl -language sv test1.v test0.v test2.v
elaborate
write_hdl -g
```

If the `test1.v` file is not affected by the macro definition in `test.h`, but the `test2.v` file is, then Example 4-11 shows the generated netlist.

Example 4-11 Generated Netlist for Verilog Macros Using One read_hdl Command

```
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_db 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.
- Genus provides a set of pre-defined packages for VHDL designs that use standard arithmetic packages defined by IEEE, Cadence, or Synopsys. The Genus-provided version of these pre-defined packages are tagged with special directives that let Genus implement the arithmetic operators efficiently. Each VHDL environment is associated with a unique set of pre-defined packages.
- In each Genus session, based on the setting of the VHDL environment (`common` or `synergy`) and the VHDL version (`1987`, `1993`, `2003`), Genus pre-loads a set of pre-defined packages from the following directory:

  ```
  $CDN_SYNTH_ROOT/lib/vhdl/
  ```
- Refer to Table 4-2 for a description of the predefined VHDL environments and to Table 4-3 for descriptions of all the predefined libraries for each of the VHDL environments.

See Using Arithmetic Packages from Other Vendors on page 143 for more information.

Table 4-2 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 4-3 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`
CADENCE	`attributes`
STD	`standard`
	`textio`
SYNOPSYS	`attributes`
	`bv_arithmetic`
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

➤ To enforce a strict interpretation of the *VHDL Language Reference Manual* (LRM) to guarantee portability to other VHDL tools, use the following attribute:

```
set_db 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 `vls /hdl_libraries/*` command. For example:

```
vls /hdl_libraries
```

For detailed information, see Chapter 2, "Genus Design Information Hierarchy."

Using Arithmetic Packages from Other Vendors

See Specifying the VHDL Environment on page 141 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_db hdl_vhdl_environment {common | synergy}
   set_db hdl_vhdl_read_version { 1993 | 1987 | 2008 }
   ```

   Genus 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 -language vhdl -library 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;
```

Genus 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 -language vhdl -library 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_db hdl_vhdl_case { lower | upper | original }
```

For example:

```
set_db 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

See "Reading Designs with Mixed Verilog-2001 and SystemVerilog Files" in *Genus HDL Modeling Guide* if your design contains a mix of Verilog-2001 and SystemVerilog files.

Reading in Verilog Modules and VHDL Entities with Same Names

Genus only supports one module or entity with a given name. Any definition, either a module or entity, overwrites a previous definition. Genus 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 newly read VHDL entity replaces any previously read Verilog module or
          VHDL entity in the same library if its name matches (case-insensitively)
          the existing module or entity.
              For instance:
                 VHDL 'foo' replaces VHDL {'FOO' or 'foo' or 'Foo' or ...} in the
                 same library.
                 VHDL 'foo' (in any library) replaces Verilog {'FOO' or 'foo' or
                 'Foo' or ...} in the same library.

          A newly read Verilog module replaces any previously read Verilog module
          if its name matches (case-sensitively) that module. Further, it replaces
          any previously read VHDL entity in the same library if its name matches
          case -insensitively) that entity.
              For instance:
                 Verilog 'foo' replaces VHDL {'FOO' or 'foo' or 'Foo' or ...} in
                 the same library
                 Verilog 'foo' replaces Verilog 'foo' only.

              In addition:
                 Verilog 'foo' does not replace Verilog 'FOO' and the two remain
     as distinct modules.
```

Using Case Sensitivity in Verilog/VHDL Mixed-Language Designs

Genus supports a mixed-language design description, which means that the files that make up the design can be written in VHDL, Verilog, and System Verilog. Verilog and System Verilog are case-sensitive languages, while VHDL is case-insensitive. Care must be taken when the HDL code refers to an object defined in another language.

Use the following attributes if your design has objects (such as modules, pins, and parameters) that are defined in one language but referenced in a different language:

➤ To specify how names defined in VHDL are referenced in Verilog or System Verilog, use the `hdl_vhdl_case` attribute.

When the `hdl_vhdl_case` attribute is set to `original`, a VHDL entity `SuB` must be instantiated as `SuB` in a Verilog file. However, if the `hdl_vhdl_case` attribute is set to `upper` (`lower`), the entity must be instantiated as `SUB` (`sub`).

➤ To specify how Verilog or System Verilog instantiations are interpreted, use the `hdl_case_sensitive_instances` root attribute.

When set to `false`, a VHDL entity `SUB` can be instantiated as `sub`, `SUB`, or `SuB` in a Verilog file. When set to `none`, it must be instantiated as `SUB`.

Reading and Elaborating a Structural Netlist Design

If the entire design is described by a Verilog-1995 or Verilog-2001structural netlist, use the `read_netlist` command to read and elaborate a structural netlist. This command creates a generic netlist that is ready to be synthesized. You do *not* need to use the `elaborate` command.

The `read_hdl -netlis`t and the `read_netlist` commands support the following attributes in the structural flow:

- Root attributes:

```
init_blackbox_for_undefined
hdl_preserve_dangling_output_nets (only supported by read_hdl
        -netlist)
init_hdl_search_path
hdl_resolve_instance_with_libcell
input_pragma_keyword
synthesis_off_command
synthesis_on_command
uniquify_naming_style
Design attribute:
hdl_filelist
```

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

If the netlist loaded with a single `read_netlist` command has multiple top-level modules, Genus randomly selects one of them and deletes the remaining top-level modules.

Each time you use the `read_netlist` command, a new design object is created in the `/designs/...` directory of the information hierarchy. As a result, the linking of structural modules that were read using multiple `read_netlist` commands did not happen explicitly because the modules resided under multiple design objects.

Note: To specify a top-level module, which should be preserved as a design object, use the `-top` *`modulename`* option with the `read_netlist` command.

Reading a Partially Structural Design

If parts of the input design is in the form of a structural netlist, then the design is a partially structural design. You can read and elaborate partially structural files provided the structural part of the input design is in the form of structural Verilog-1995 constructs and is contained in files separate from the non-structural (RTL) input.

Example 4-12 shows a typical read and elaborate session for a partially structural design.

- `read_hdl -netlist` is used to load the structural input files
- `read_hdl` without the `-netlist` option is used to load RTL files

After using the `read_hdl` command these modules are visible in the design hierarchy in the `/hdl_libraries/default/architectures` directory as `hdl_architecture` object types, such as regular RTL input modules. You can then use these paths to get and set attributes on the architecture objects for the structural modules before using the `elaborate` command.

After the partially structural design has been read using one or more `read_hdl` and `read_hdl -netlist` commands, use the `elaborate` command to elaborate the top modules (including those that may be among the structural input), which will represent them as separate design objects in the `/designs` directory. If you want to elaborate a specific module or set of modules (whether RTL or structural) as the top module(s), then specify this list of modules as an argument to the `elaborate` command.

Even though you can read structural files using the `read_hdl` command without the `-netlist` option, using the `-netlist` option lets you read structural files much more efficiently that results in less runtime and memory than using the `read_hdl` command without the `-netlist` option. This efficiency in runtime and memory also applies when you elaborate a structural module that has been read using the `read_hdl -netlist` command

Example 4-12 Reading a Partially Structural Design

```
## Commands for reading a technology library, and so on.
...
## Commands for reading RTL and structural input.
read_hdl rtl1.v rtl2.v
read_hdl -language vhdl rtl3.vhdl rtl4.vhdl
read_hdl -netlist struct1.v struct2.v struct3.v
read_hdl rtl5.v ...
read_hdl -netlist struct4.v ...
...
## Command for getting/setting attributes on hdl_architecture objects
## (including the structural modules read in) under hdl_libraries vdir.
...
## Commands for elaboration
elaborate <optional list of top modules RTL/structural/both>
## Commands for optimization and so on.
read_sdc
...
syn_gen
syn_map
...
```

Keeping Track of Loaded HDL Files

➤ Use the `hdl_filelist` attribute to keep track of the HDL files that have been read into Genus. Each time you use elaborate command, the library, filename, and language format are appended to this attribute in a TCL list.

```
read_hdl -language v2001 top.v
elaborate
get_db design:top .hdl_filelist
{default -v2001 {SYNTHESIS} {top.v} {} {}}
```

```
read_hdl -language vhdl -library mylib sub.vhdl
elaborate
get_db module:top/sub .hdl_filelist
{mylib -v2001 {SYNTHESIS} {sub.vhd} {} {}}
```

Importing the Floorplan

Import the floorplan through the DEF file. 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.

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

In Genus, the most common use for the DEF file is to specify the floorplan and placement information. To import a DEF file, use the `read_def` command.

```
read_def tutorial.def
```

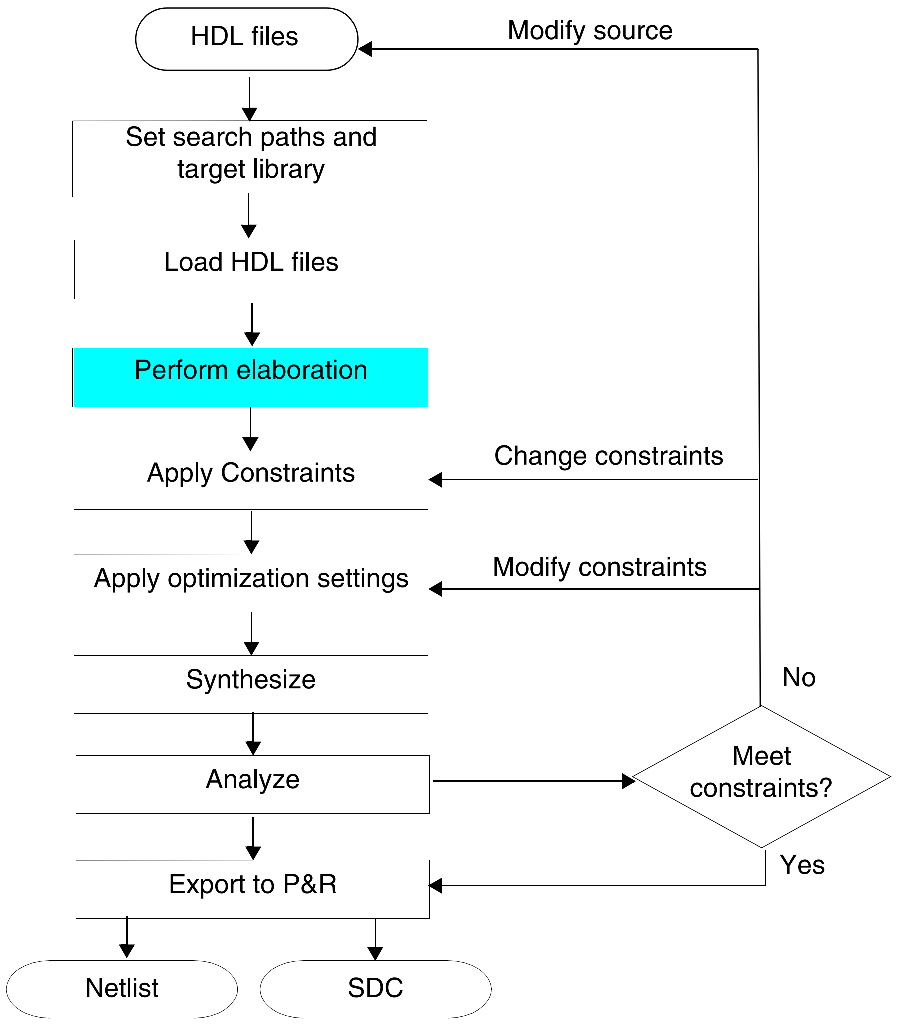
5

Elaborating the Design

- Overview on page 152
- Tasks on page 153
    - Performing Elaboration on page 153
    - Specifying Top-Level Parameters or Generic Values on page 154
    - Specifying HDL Library Search Paths on page 156
    - Elaborating a Specified Module or Entity on page 156
    - Naming Individual Bits of Array and Record Ports and Registers on page 156
    - Naming Parameterized Modules on page 159
    - Keeping Track of the RTL Source Code on page 162
    - Grouping an Extra Level of Design Hierarchy on page 163

Overview

Elaboration involves various design checks and optimizations and is a necessary step to proceed with synthesis. This chapter describes elaboration in detail.



Tasks

- Performing Elaboration on page 153
- Specifying Top-Level Parameters or Generic Values on page 154
- Specifying HDL Library Search Paths on page 156
- Elaborating a Specified Module or Entity on page 156
- Naming Individual Bits of Array and Record Ports and Registers on page 156
- Naming Parameterized Modules on page 159
- Keeping Track of the RTL Source Code on page 162
- Grouping an Extra Level of Design Hierarchy on page 163

Performing Elaboration

The `elaborate` command automatically elaborates the top-level design and all of its references. During elaboration, Genus 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, Genus automatically links the cells to their references in the technology library during elaboration. You do not have to issue an additional command for linking.

```
elaborate [-parameters param] [-lib_path path]...
        [-lib_extension  ext]... [module]...
```

At the end of elaboration, Genus 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, Genus has an internally created data structure for the whole design so you can apply constraints and perform other operations.

Specifying Top-Level Parameters or Generic Values

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 127.

2. Type the following command to start elaboration with no parameters:

   ```
   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 the value of `width` to `16` by issuing the following command:

```
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`.

```
elaborate design1 -parameters {16 8 32}
```

Overriding Top-Level Parameter or Generic Values

While automatic elaboration works for designs that are instantiated in a higher level design, some applications require to override the default parameters or generic values directly from the `elaborate` command, as in elaborating top-level modules or entities with different parameters or generic values.

➤ Override the default parameter values using the `-parameters` option with the `elaborate` command, as shown in Example 5-1. This option specifies the values to use for the indicated parameters.

Example 5-1 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.lib
Elaborating top-level block 'TOP_L3_R2' from file 'ex11.v'.
Done elaborating 'TOP_L3_R2'
```

➤ 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...]
```

By default, the top-level module is built. If fewer parameters are specified than the ones existing in the design, then the default values of the missing parameters will be used in building the design. If more parameters are specified than the ones existing 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]}}"
```

Specifying HDL Library Search Paths

➤ Specify a list of UNIX directories where Genus 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_db init_hdl_search_path {../location_of_top}
read_hdl top.v
set_db library tutorial.lib
elaborate -lib_path ../mylibs -lib_path /home/verilog/libs -lib_extension ".h"
-lib_extension ".v"
```

The latter command is equivalent to the following:

```
elaborate -lib_path {../mylibs /home/verilog/libs} -lib_extension { ".h" ".v" }
```

The `elaborate` command looks for the `top.v` file in the directories specified through the `init_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 `-lib_path` 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`

- ➡ Use the `hdl_array_naming_style` attribute to control the format for naming individual elements of an array variable or signal in the synthesized netlist.

  Default: `%s[%d]`

  The `hdl_array_naming_style` attribute value must include one instance of `%s` to represent the variable name followed by one instance of `%d` to represent the bit number. For example, possible values are `%s%d`, `%s[%d]`, and `%s__%d`.

- ➡ Use the `hdl_record_naming_style` attribute to control the format for naming individual elements of a record variable or signal in the synthesized netlist.

  Default: `%s[%s]`

  The `hdl_record_naming_style` attribute value must include two instances of `%s`, the first to represent the variable name and the second to represent the field name.

- ➡ Use the `hdl_reg_naming_style` attribute to control the format for naming flip-flop or latch instances inferred from signals or variables in the input HDL.

  Default: `%s_reg%s`

  The `hdl_reg_naming_style` attribute value must include two instances of `%s`, the first to represent the name of the variable from which the flip-flop or latch was inferred, and the second to represent the bit number as specified by the `hdl_array_naming_style` attribute if the variable is an array.

Note: When setting the naming style attribute values, Tcl-special characters such as brackets must be escaped with the `\` character. For example:

```
set_db hdl_array_naming_style %s\[%d\]
```

The following table shows how variables in input Verilog are specified in the netlist for the default values of the naming style attributes.

Description	Input Verilog	Netlist Wire	Netlist Instance
scalar	`reg a;`	`wire a;`	`CDN_flop a_reg (...);`
bit-vector	`reg [1:0] b;`	`wire [1:0] b;`	`CDN_flop \b_reg[0] (...);`

array	`reg [1:0] c[5:4][3:2]`	`wire [1:0] \c[4][2];`	`CDN_flop \c_reg[4][2][0] (...);`
			`CDN_flop \c_reg[4][2][1] (...);`

record	`typedef struct {`
	`[1:0] f1;`
	`f2;`
	`}rec_type;`
	`rec_type d;`
		`wire [1:0] \d[f1] ;`
		`wire \d[f2] ;`
			`CDN_flop \d_reg[f2] (...);`
			`CDN_flop \d_reg[f1][0] (...);`

Naming Individual Bits of Multi-Bit Wires

To specify the format to name individual bits of bus wires, set the `hdl_bus_wire_naming_style` root attribute before elaborating the design.

Default: `%s[%d]` where `%s` refers to the variable name and `%d` to the individual bit.

Example

Consider the following RTL code.

```
module test1(clk,d,q);
    input clk;
    input [0:3] d;
    output [0:3] q;
    reg [0:3] q, tmp;
    always @ (posedge clk) begin
        tmp = d;
    end
    always @ (posedge clk) begin
        q = tmp;
    end
endmodule
```

Assume the following script:

```
set_db library tutorial.lib
set_db hdl_bus_wire_naming_style %s__%d
read_hdl test.v
elaborate
```

After elaboration, the netlist will look like:

```
module test1(clk, d, q);
    input clk;
    input [0:3] d;
    output [0:3] q;
    wire clk;
    wire [0:3] d;
    wire [0:3] q;
    wire tmp__0, tmp__1, tmp__2, tmp__3;
    CDN_flop \tmp_reg[3] (.clk (clk), .d (d[3]), .sena (1'b1), .aclr
    (1'b0), .apre (1'b0), .srl (1'b0), .srd (1'b0), .q (tmp__3));
    ...
    CDN_flop \tmp_reg[0] (.clk (clk), .d (d[0]), .sena (1'b1), .aclr
        (1'b0), .apre (1'b0), .srl (1'b0), .srd (1'b0), .q (tmp__0));
    CDN_flop \q_reg[3] (.clk (clk), .d (tmp__3), .sena (1'b1), .aclr
        (1'b0), .apre (1'b0), .srl (1'b0), .srd (1'b0), .q (q[3]));
    ...
endmodule
```

```
'ifdef GEN_CDN_GENERIC_GATE
'else
module CDN_flop(clk, d, sena, aclr, apre, srl, srd, q);
...
endmodule
```

Naming Parameterized Modules

➤ Specify the format of module names generated for parameterized modules using the `hdl_parameter_naming_style` attribute. For example:

```
set_db hdl_parameter_naming_style _%s%d
```

The `elaborate` command automatically elaborates the design by propagating parameter values specified for instantiation, as shown in Example 5-2. 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 module (or module) will be `BOT_L7_R0`.

Example 5-2 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 5-3 is a VHDL design that will be used to show how specify different suffix formats using the `hdl_parameter_naming_style` attribute.

Example 5-3 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 5-4, then the modules names in the netlist will be as shown in Example 5-5.

Example 5-4 set_db hdl_parameter_naming_style _%s_%d

```
set_db hdl_parameter_naming_style "_%s_%d"
set_db library tutorial.lib
read_hdl -language vhdl test.vhd
elaborate top
write_hdl
```

Example 5-5 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 5-6, then the modules names in the netlist will be as shown in Example 5-7.

Example 5-6 set_db hdl_parameter_naming_style "_%s%d"

```
set_db hdl_parameter_naming_style "_%s%d"
set_db library tutorial.lib
read_hdl -language vhdl test.vhd
elaborate top
write_hdl
```

Example 5-7 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 5-8, then the modules names in the netlist will be as shown in Example 5-9.

Example 5-8 set_db hdl_parameter_naming_style "_%d"

```
set_db hdl_parameter_naming_style "_%d"
set_db library tutorial.lib
read_hdl -language vhdl test.vhd
elaborate top
write_hdl
```

Example 5-9 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
```

Keeping Track of the RTL Source Code

➤ Set the following attribute to `true` to keep track of the RTL source code:

```
set_db hdl_track_filename_row_col { true | false }
```

Default: `false`

This attribute enables Genus to keep track of filenames, line numbers, and column numbers for all instances before optimization. Genus also uses this information in subsequent error and warning messages. Set this attribute to `true` to enable file, row, column information before using the `elaborate` command.

Grouping an Extra Level of Design Hierarchy

In general, the design hierarchy described in the RTL code is sacred. Using the `elaborate` command never ungroups a design hierarchy that you have defined. By default, the `elaborate` command does not add a tool-defined hierarchy. The `elaborate` command creates an additional level of design hierarchy in the following two cases:

- When there are datapath components
- When the `group` attribute of an `hdl_procedure` or `hdl_block` object is given a value that is not an empty string. Use the `set_db` command to arrange values of the `group` attribute after using the `read_hdl` command and before using the `elaborate` command.

An `hdl_procedure` represents either a process in VHDL or the named begin and end block of an `always` construct in Verilog. An `hdl_block` represents a VHDL block. Each `hdl_procedure` and `hdl_block` has a `group` attribute, whose default value is an empty string. During elaboration, within a level of design hierarchy, for example within a Verilog module or a VHDL entity, all `hdl_procedure` and `hdl_block` objects whose `group` attribute share the same non-empty value is *grouped* as a level of extra design hierarchy.

Grouping of `hdl_procedure` and `hdl_block` objects does not go beyond the boundary of a user-defined design hierarchy. If two `hdl_procedure` and `hdl_block` objects in two different modules or entities have the same value assigned to the `group` attribute, then they will not be put into one module.

To shorten the netlist in the following examples, all these sample RTL designs only infer combinational logic. In actuality, there can be sequential logic in the extra level of design hierarchy created through this mechanism.

- Grouping Multiple Named Verilog Blocks in Verilog into One Module on page 164
- Grouping Multiple Labeled Processes in VHDL into One module on page 166
- Grouping Multiple Labeled Blocks in VHDL into One module on page 168
- Grouping Multiple Instances of Parameterized Named Blocks in Verilog into modules on page 169
- Grouping Multiple Instances of a Parameterized Process in VHDL into modules on page 171
- Grouping Multiple Instances of a Parameterized Block in VHDL into modules on page 174
- Grouping Generated Instances of Named Blocks in Verilog into modules on page 176

- Grouping Generated Instances of Labeled Processes in VHDL into modules on page 178

Grouping Multiple Named Verilog Blocks in Verilog into One Module

If there are multiple `hdl_procedure` objects from multiple named `begin` and `end` blocks, and the value of their `group` attributes are the same, their contents are *grouped* into one module.

If there are multiple `hdl_procedure` objects, from multiple named `begin` and `end` blocks, and the value of their `group` attributes are different, there will be multiple modules, one for each of these `begin`-`end` blocks.

As shown in Example 5-10, the `group` attribute of the `b1` and `b3` `always` blocks are given the same non-empty value of `xgrp`. Therefore, they are *grouped* as a module named `ex2_xgrp` during elaboration.

If there are multiple levels of begin and end blocks in the body of an `always` construct, then only the outermost begin and end block is made an `hdl_procedure` object. An `hdl_procedure` object is not created for an inner block.

To reproduce this example, take the Verilog, shown in Example 5-10:

Example 5-10 Grouping Multiple Named Blocks in Verilog into One module

```
module ex2 (y, a, b);
    input [3:0] a, b;
    output [3:0] y;
    reg [3:0] a_bv, av_b, y;
    always @(a or b)
    begin : b1 a_bv =a & ~b;
    end
    always @(a or b)
    begin : b2  av_b =~a & b;
    end
    always @(a_bv or av_b)
    begin : b3  y =a_bv | av_b;
    end
endmodule
```

And use the following commands:

```
set_db library tutorial.lib
read_hdl test.v
set_db [get_db hdl_procedures *b1] .group xgrp
set_db [get_db hdl_procedures *b3] .group xgrp
elaborate
write_hdl
```

Example 5-11 shows the resulting post-elaboration generic netlist:

Example 5-11 Post-Elaboration Generic Netlist for Grouping Multiple Named Blocks into One module

```
module ex2_xgrp (y, av_b, a, b);
    input [3:0] av_b, a, b;
    output [3:0] y;
    wire [3:0] a_bv, bv;
    not g3 (bv[3], b[3]);
    not g2 (bv[2], b[2]);
    not g1 (bv[1], b[1]);
    not g0 (bv[0], b[0]);
    and g7 (a_bv[3], a[3], bv[3]);
    and g6 (a_bv[2], a[2], bv[2]);
    and g5 (a_bv[1], a[1], bv[1]);
    and g4 (a_bv[0], a[0], bv[0]);
    or g13 (y[3], a_bv[3], av_b[3]);
    or g12 (y[2], a_bv[2], av_b[2]);
    or g11 (y[1], a_bv[1], av_b[1]);
    or g10 (y[0], a_bv[0], av_b[0]);
endmodule
    module ex2 (y, a, b);
    input [3:0] a, b;
    output [3:0] y;
    wire [3:0] av_b, av;
    not g1 (av[3], a[3]);
    not g3 (av[2], a[2]);
    not g4 (av[1], a[1]);
    not g5 (av[0], a[0]);
    and g6 (av_b[0], av[0], b[0]);
    and g2 (av_b[1], av[1], b[1]);
    and g7 (av_b[2], av[2], b[2]);
    and g8 (av_b[3], av[3], b[3]);
    ex2_xgrp ex2_xgrp (.y(y), .av_b(av_b), .a(a), .b(b));
endmodule
```

Grouping Multiple Labeled Processes in VHDL into One module

If there are multiple `hdl_procedure` objects from multiple labeled processes, and the value of their `group` attributes are the same, then their contents are *grouped* into one module.

However, if the value of their `group` attributes are different, then there will be multiple modules, one for each of these processes.

In Example 5-12, the `group` attribute of the b1 and b3 processes are given the same non-empty value of `xgrp`. Therefore they are *grouped* as a module named `ex2_xgrp` during elaboration.

To reproduce this example, take the VHDL, shown in Example 5-12:

Example 5-12 Grouping Multiple Labeled Processes in VHDL into One module

```
library ieee;
use ieee.std_logic_1164.all;
entity ex2 is
    port (y : out std_logic_vector (3 downto 0);
        a, b : in std_logic_vector (3 downto 0)  );
end;
architecture rtl of ex2 is
    signal a_bv, av_b : std_logic_vector (3 downto 0);
begin
    b1 : process (a, b)
    begin  a_bv <= a and (not b);
    end process;
    b2 : process (a, b)
    begin  av_b <= (not a) and b;
    end process;
    b3 : process (a_bv, av_b)
    begin  y <= a_bv or av_b;
    end process;
end;
```

And use the following commands:

```
set_db library tutorial.lib
read_hdl -language vhdl test.vhd
set_db [get_db hdl_procedures *b1] .group xgrp
set_db [get_db hdl_procedures *b3] .group xgrp
elaborate
write_hdl
```

Example 5-11 shows the resulting post-elaboration generic netlist.

Grouping Multiple Labeled Blocks in VHDL into One module

If there are multiple `hdl_block` objects from multiple labeled blocks, and the value of their `group` attributes are the same, then their contents are *grouped* into one module.

However, if the value of their group attributes are different, then there will be multiple modules, one for each of these processes.

In Example 5-13, the `group` attribute of the `b1` and `b3` blocks are given the same non-empty value of `xgrp`. Therefore they are *grouped* as a module named `ex2_xgrp` during elaboration.

To reproduce this example, take the VHDL, shown in Example 5-13:

Example 5-13 Grouping Multiple Labeled Blocks in VHDL into One module

```
library ieee;
use ieee.std_logic_1164.all;
entity ex2 is
    port (y : out std_logic_vector (3 downto 0);
        a, b : in std_logic_vector (3 downto 0) );
end;
architecture rtl of ex2 is
    signal a_bv, av_b : std_logic_vector (3 downto 0);
begin
    b1 : block  begin  a_bv <= a and (not b);
    end block;
    b2 : block  begin  av_b <= (not a) and b;
    end block;
    b3 : block  begin  y <= a_bv or av_b;
    end block;
end;
```

And use the following commands:

```
set_db library tutorial.lib
read_hdl -language vhdl test.vhd
set_db [get_db hdl_blocks *b1] .group xgrp
set_db [get_db hdl_blocks *b3] .group xgrp
elaborate
write_hdl
```

Example 5-11 shows the resulting post-elaboration generic netlist.

Grouping Multiple Instances of Parameterized Named Blocks in Verilog into modules

Assume that there is a parameterized sub-module that is instantiated multiple times, all with the same parameter value. There is also a named `begin` and `end` block in this sub-module and the `group` attribute of its `hdl_procedure` is given a non-empty value. After elaboration, the sub-module is represented by one module; therefore, the `hdl_procedure` becomes one module. This happens between the `u1` and `u2` instances, shown in Example 5-14.

If a parameterized sub-module is instantiated multiple times with different parameter values, then elaboration uniquifies this sub-module and makes one module for each unique set of its parameter values. Before elaboration, when unification has not taken place, a named `begin` and `end` block in such a sub-module is represented by one `hdl_procedure` object. Assume this `hdl_procedure` is given a non-empty `group` attribute. After elaboration, this one `hdl_procedure` becomes multiple modules, one from each of the uniquified parent module, as shown in Example 5-15.

This happens between the `u1` and `u3` instances, shown in Example 5-14. In other words, during elaboration, making a named block a level of design hierarchy takes place after uniquifying parameterized modules.

To reproduce this example, take the Verilog, shown in Example 5-14:

Example 5-14 Grouping Multiple Instances of Parameterized Named Blocks in Verilog into modules

```
module mid (y, a, b, c);
    parameter w = 8;
    input [w-1:0] a, b, c;
    reg [w-1:0] p;
    output [w-1:0] y;
    always @(a or b)
    begin : blok
    p = a & b;
    end
    assign y = p | c;
endmodule
module ex4 (x, y, z, a, b, c, d);
    parameter w = 4;
    input [w+1:0] a, b, c, d;
    output [w-1:0] x, y;
    output [w+1:0] z;
    mid #(w)   u1 (x, a[w-1:0], b[w-1:0], c[w-1:0]);
    mid #(w)   u2 (y, a[w-1:0], b[w-1:0], d[w-1:0]);
    mid #(w+2) u3 (z, c[w+1:0], d[w+1:0], a[w+1:0]);
endmodule
```

And use the following commands:

```
set_db library tutorial.lib
read_hdl test.v
set_db [get_db hdl_procedures *blok] .group xgrp
elaborate
write_hdl
```

Example 5-15 shows the resulting post-elaboration generic netlist.

Example 5-15 Post-Elaboration Netlist for Grouping Multiple Instances of Parameterized Named Blocks into modules

```
module or_op(A, B, Z);
  input [3:0] A, B;
  output [3:0] Z;
  wire [3:0] A, B;
  wire [3:0] Z;
  or g1 (Z[0], A[0], B[0]);
  or g2 (Z[1], A[1], B[1]);
  or g3 (Z[2], A[2], B[2]);
  or g4 (Z[3], A[3], B[3]);
endmodule
module and_op(A, B, Z);
  input [3:0] A, B;
  output [3:0] Z;
  wire [3:0] A, B;
  wire [3:0] Z;
  and g1 (Z[0], A[0], B[0]);
  and g2 (Z[1], A[1], B[1]);
  and g3 (Z[2], A[2], B[2]);
  and g4 (Z[3], A[3], B[3]);
endmodule
module or_op_2(A, B, Z);
  input [5:0] A, B;
  output [5:0] Z;
  wire [5:0] A, B;
  wire [5:0] Z;
  or g1 (Z[0], A[0], B[0]);
  or g2 (Z[1], A[1], B[1]);
  or g3 (Z[2], A[2], B[2]);
  or g4 (Z[3], A[3], B[3]);
  or g5 (Z[4], A[4], B[4]);
  or g6 (Z[5], A[5], B[5]);
endmodule
module and_op_1(A, B, Z);
  input [5:0] A, B;
  output [5:0] Z;
  wire [5:0] A, B;
  wire [5:0] Z
  and g1 (Z[0], A[0], B[0]);
  and g2 (Z[1], A[1], B[1]);
  and g3 (Z[2], A[2], B[2]);
  and g4 (Z[3], A[3], B[3]);
  and g5 (Z[4], A[4], B[4]);
  and g6 (Z[5], A[5], B[5]);
endmodule

module mid_w4_xgrp(p0, a0, b0);
  input [3:0] a0, b0;
  output [3:0] p0;
  wire [3:0] a0, b0;
  wire [3:0] p0;
  and_op g1(.A (a0), .B (b0), .Z (p0));
endmodule

module mid_w4(y, a, b, c);
  input [3:0] a, b, c;
  output [3:0] y;
  wire [3:0] a, b, c;
  wire [3:0] y;
  wire [3:0] p;
  or_op g1(.A (p), .B (c), .Z (y));
  mid_w4_xgrp xgrp(.p0 (p), .a0 (a), .b0 (b));
endmodule

module mid_w6_xgrp(p1, a1, b1);
  input [5:0] a1, b1;
  output [5:0] p1;
  wire [5:0] a1, b1;
  wire [5:0] p1;
  and_op_1 g1(.A (a1), .B (b1), .Z (p1));
endmodule

module mid_w6(y, a, b, c);
  input [5:0] a, b, c;
  output [5:0] y;
  wire [5:0] a, b, c;
  wire [5:0] y;
  wire [5:0] p;
  or_op_2 g1(.A (p), .B (c), .Z (y));
  mid_w6_xgrp xgrp(.p1 (p), .a1 (a), .b1 (b));
endmodule

module ex4(x, y, z, a, b, c, d);
  input [5:0] a, b, c, d;
  output [3:0] x, y;
  output [5:0] z;
  wire [5:0] a, b, c, d;
  wire [3:0] x, y;
  wire [5:0] z;
  mid_w4 u1(x, a[3:0], b[3:0], c[3:0]);
  mid_w4 u2(y, a[3:0], b[3:0], d[3:0]);
  mid_w6 u3(z, c, d, a);
endmodule
```

Grouping Multiple Instances of a Parameterized Process in VHDL into modules

For this example, assume there is a parameterized entity that is instantiated multiple times, all with the same parameter value. There is also a labeled process in this entity and the `group` attribute of its `hdl_procedure` is given a non-empty value. After elaboration, the entity is represented by one module; therefore, the `hdl_procedure` becomes one module.

This happens between u1 and u2 instances shown in Example 5-16. If a parameterized entity is instantiated multiple times with different parameter values, then elaboration uniquifies this entity and makes one module for each unique set of its parameter values. Before elaboration, when unification has not taken place, a labeled process in such an entity is represented by one `hdl_procedure` object. Assume this `hdl_procedure` is given a non-empty `group` attribute. After elaboration, this `hdl_procedure` object becomes multiple modules, one from each of the uniquified parent entity.

This happens between the `u1` and `u3` instances, as shown in Example 5-16. In other words, during elaboration, making a labeled process a level of design hierarchy takes place after uniquifying parameterized entities.

To reproduce this example, take the VHDL, shown in Example 5-16:

Example 5-16 Grouping Multiple Instances of a Parameterized Process in VHDL into modules

```
library ieee;
use ieee.std_logic_1164.all;
entity mid is
     generic (w : integer := 8);
     port (y : out std_logic_vector (w-1 downto 0);
          a, b, c : in  std_logic_vector (w-1 downto 0)  );
end;
architecture rtl of mid is
     signal p : std_logic_vector (w-1 downto 0);
begin
     blok : process (a, b)
     begin
          p <= a and b;
     end process;
     y <= p or c;
end;
library ieee;
use ieee.std_logic_1164.all;
entity ex4 is
     generic (w : integer := 4);
     port (x, y : out std_logic_vector (w-1 downto 0);
     z : out std_logic_vector (w+1 downto 0);
     a, b, c, d : in std_logic_vector (w+1 downto 0)  );
end;
architecture rtl of ex4 is
     component mid
     generic (w : integer := 8);
     port (y : out std_logic_vector (w-1 downto 0);
     a, b, c : in  std_logic_vector (w-1 downto 0)  );
     end component;
begin
     u1: mid generic map (w)
     port map (x, a(w-1 downto 0), b(w-1 downto 0), c(w-1 downto 0));
     u2: mid generic map (w)
     port map (y, a(w-1 downto 0), b(w-1 downto 0), d(w-1 downto 0));
     u3: mid generic map (w+2)
     port map (z, c(w+1 downto 0), d(w+1 downto 0), a(w+1 downto 0));
end;
```

And use the following commands:

```
set_db library tutorial.lib
read_hdl -language vhdl test.vhd
set_db [get_db hdl_procedures *blok] .group xgrp
elaborate
write_hdl
```

Example 5-15 shows the resulting post-elaboration generic netlist.

Grouping Multiple Instances of a Parameterized Block in VHDL into modules

Assume there is a parameterized entity that is instantiated multiple times, all with the same parameter value. There is also a labeled block in this entity, and the `group` attribute of its `hdl_block` is given a non-empty value. After elaboration, the entity is represented by one module; therefore, the `hdl_block` becomes one module.

This happens between the u1 and u2 instances, as shown in Example 5-17. If a parameterized entity is instantiated multiple times with different parameter values, then elaboration uniquifies this entity and creates one module for each unique set of its parameter values. Before elaboration, when unification has not taken place, a labeled block in such an entity is represented by one `hdl_block` object. Assume this `hdl_block` is given a non-empty `group` attribute. After elaboration, this one `hdl_block` becomes multiple modules, one from each of the uniquified parent entity.

This happens between the `u1` and `u3` instances, as shown in Example 5-17. In other words, during elaboration, making a labeled process a level of design hierarchy takes place after uniquifying parameterized entities.

To reproduce this example, take the VHDL, shown in Example 5-17:

Example 5-17 Grouping Multiple Instances of a Parameterized Process in VHDL into modules

```
library ieee;
    use ieee.std_logic_1164.all;
    entity mid is
        generic (w : integer := 8);
        port (y : out std_logic_vector (w-1 downto 0);
        a, b, c : in  std_logic_vector (w-1 downto 0)  );
end;
architecture rtl of mid is
    signal p : std_logic_vector (w-1 downto 0);
begin
    blok : block
        begin
            p <= a and b;
    end block;
    y <= p or c;
end;
library ieee;
    use ieee.std_logic_1164.all;
    entity ex4 is
        generic (w : integer := 4);
        port (x, y : out std_logic_vector (w-1 downto 0);
        z : out std_logic_vector (w+1 downto 0);
        a, b, c, d : in std_logic_vector (w+1 downto 0)  );
end;
architecture rtl of ex4 is
    component mid
    generic (w : integer := 8);
    port (y : out std_logic_vector (w-1 downto 0);
    a, b, c : in  std_logic_vector (w-1 downto 0)  );
    end component;
begin
    u1: mid generic map (w)
    port map (x, a(w-1 downto 0), b(w-1 downto 0), c(w-1 downto 0));
    u2: mid generic map (w)
    port map (y, a(w-1 downto 0), b(w-1 downto 0), d(w-1 downto 0));
    u3: mid generic map (w+2)
    port map (z, c(w+1 downto 0), d(w+1 downto 0), a(w+1 downto 0));
end;
```

And use the following commands:

```
set_db library tutorial.lib
read_hdl -language vhdl test.vhd
set_db [get_db hdl_blocks *blok] .group xgrp
elaborate
write_hdl
```

Example 5-15 shows the resulting post-elaboration generic netlist.

Grouping Generated Instances of Named Blocks in Verilog into modules

If a named begin and end block is the body of an `always` construct inside of a `for generate` statement, then the named block is represented by one `hdl_procedure` object before elaboration, even if there are multiple iterations in the `for generate` statement. The `for loop` has not been unrolled.

This happens between the `for generate` statement and the `blok` block, as shown in Example 5-18. During elaboration, making a named block a level of design hierarchy takes place after unrolling `for generate` loops.

Assume the `group` attribute of this `hdl_procedure` object is given a non-empty value. During elaboration, when the loop is unrolled, the `hdl_procedure` object is duplicated. With every duplicated copy of this `hdl_procedure` object, the `group` attribute carries the same value as the original copy. Since all these `hdl_procedure` objects share the same `group` setting, their contents are *grouped* into one module. This happens to all instances of the `blok` block, as shown in Example 5-18.

The unrolling of loops and duplication of the `hdl_procedure` objects occurs in the middle of elaboration. You cannot assign different values to the `group` attribute of the duplicated `hdl_procedure` objects.

To reproduce this example, take the Verilog, as shown in Example 5-18:

Example 5-18 Grouping Generated Instances of Named Blocks in Verilog into modules

```
module ex5 (y, a, c);
    parameter w = 4, d = 3;
    input [w*d-1:0] a;
    input [d-1:0] c;
    reg [d-1:0] p;
    output [d-1:0] y;
    genvar i;
    generate for (i=0; i<=d-1; i=i+1)
        always @(a)
        begin : blok
        p[i] = ^a[w*(i+1)-1:w*i];
        end
    endgenerate
    assign y = p & c;
endmodule
```

And use the following commands:

```
set_db library tutorial.lib
read_hdl -language v2001 test.v
set_db [get_db hdl_procedures *blok] .group xgrp
elaborate
write_hdl
```

Example 5-19 shows the resulting post-elaboration generic netlist:

Example 5-19 Post-Elaboration Netlist for Grouping Generated Instances of Named Blocks in Verilog into modules

```
module ex5_xgrp (p, a);
    input [11:0] a;
    output [2:0] p;
    wire n_5, n_6, n_8, n_9, n_10, n_11;
    xor g1 (n_5, a[8], a[11]);
    xor g4 (n_6, a[10], a[9]);
    xor g5 (p[2], n_5, n_6);
    xor g6 (n_8, a[4], a[7]);
    xor g2 (n_9, a[6], a[5]);
    xor g7 (p[1], n_8, n_9);
    xor g8 (n_10, a[0], a[3]);
    xor g9 (n_11, a[2], a[1]);
    xor g3 (p[0], n_10, n_11);
endmodule

module ex5 (y, a, c);
    input [11:0] a;
    input [2:0] c;
    output [2:0] y;
    wire [2:0] p;
    ex5_xgrp ex5_xgrp (.p(p), .a(a));
    and g1 (y[0], p[0] , c[0]);
    and g2 (y[1], p[1] , c[1]);
    and g3 (y[2], p[2] , c[2]);
endmodule
```

Grouping Generated Instances of Labeled Processes in VHDL into modules

If a labeled process is inside a `for generate` statement, the labeled process is represented by one `hdl_procedure` object before elaboration even if there are multiple iterations in the `for generate` statement.

The `for loop` has not been unrolled. This happens between the `for generate` statement and the `blok` process, as shown in Example 5-18.

During elaboration, making a labeled process a level of design hierarchy takes place after unrolling `for generate` loops.

In this example, assume the `group` attribute of this `hdl_procedure` object is given a non-empty value. During elaboration, when the loop is unrolled, the `hdl_procedure` is duplicated. With every duplicated copy of this `hdl_procedure` object the `group` attribute carries the same value as the original copy. Since all these `hdl_procedure` objects share the same `group` attribute setting, their contents are *grouped* into one module. This happens to all instances of the `blok` process, as shown in Example.

The unrolling of loops and duplication of `hdl_procedure` objects happens in the middle of elaboration. You cannot assign different values to the `group` attribute of the duplicated `hdl_procedure` objects.

To reproduce this example, take the VHDL, as shown in Example 5-20:

Example 5-20 Grouping Generated Instances of Labeled Processes in VHDL into modules

```
library ieee;
use ieee.std_logic_1164.all;
entity ex5 is
    generic (w : integer := 4;
        d : integer := 3);
    port (  y : out std_logic_vector (d-1 downto 0);
        a : in  std_logic_vector (w*d-1 downto 0);
        c : in  std_logic_vector (d-1 downto 0) );
end;
architecture rtl of ex5 is
    signal p : std_logic_vector (d-1 downto 0);
begin
    g0 : for i in 0 to d-1 generate
        blok : process (a)
        variable tmp : std_logic;
    begin
        tmp := '0';
        for j in w*(i+1)-1 downto w*i loop
        tmp := tmp xor a(j);
        end loop;
        p(i) <= tmp;
    end process;
    end generate;
    y <= p and c;
end;
```

And use the following commands:

```
set_db library tutorial.lib
read_hdl -language vhdl test.vhd
set_db [get_db hdl_procedures *blok] .group xgrp
elaborate
write_hdl
```

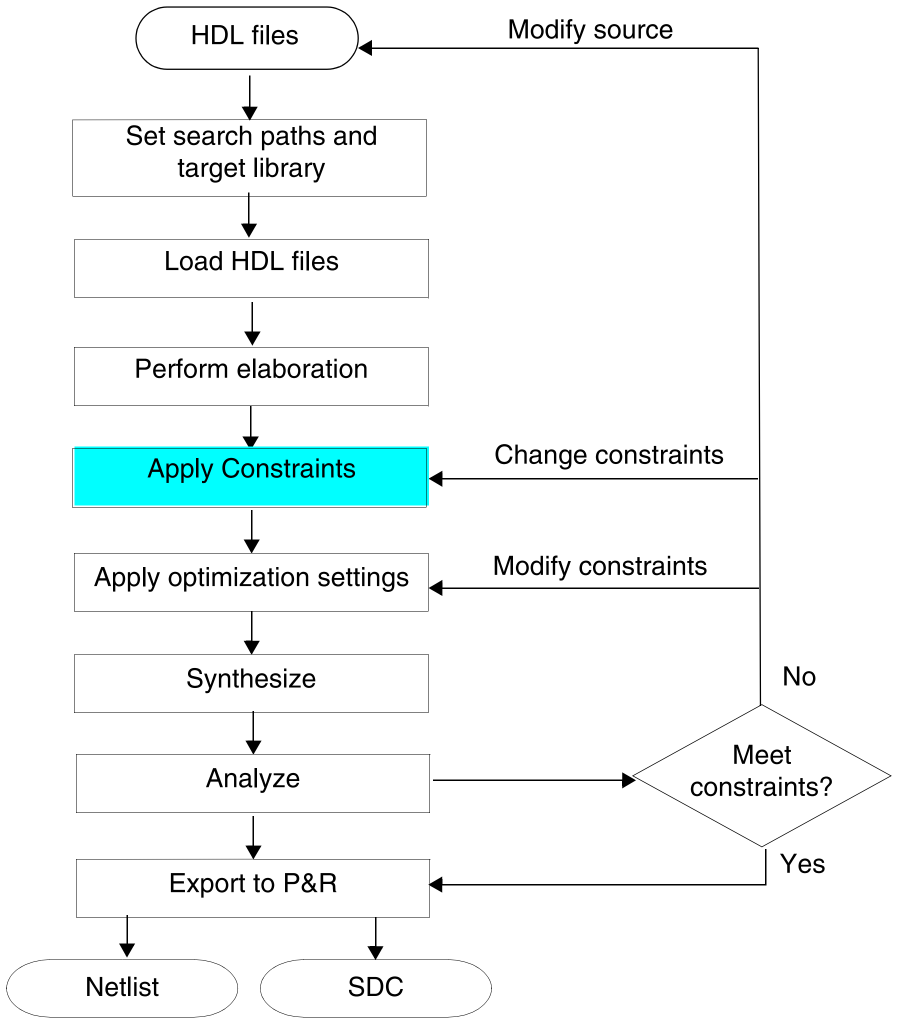
Example 5-19 shows the resulting post-elaboration generic netlist.

6

Applying Constraints

- Overview on page 182
- Tasks on page 183
    - Importing and Exporting SDC on page 183
    - Applying Timing Constraints on page 183
    - Importing Physical Information on page 184
    - Applying Design Rule Constraints on page 184
    - Creating Ideal Objects on page 185

Overview



This chapter describes how to apply the basic constraints in Genus.

Tasks

- Importing and Exporting SDC
- Applying Timing Constraints
- Importing Physical Information
- Applying Design Rule Constraints

Importing and Exporting SDC

Genus provides the ability to read in and write out SDC constraints.

➤ To import SDC constraints, use the `read_sdc` command:

```
read_sdc filename
```

➤ To export SDC constraints use the `write_sdc` command:

```
write_sdc > filename
```

Applying Timing Constraints

In Genus, 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 `create_clock` (SDC) command.

Note: Genus uses picoseconds and femtofarads as units. It *does not* use nanoseconds and picofarads.

You can group clocks that are synchronous to each other, allowing timing analysis to be performed between these clocks. This group of clocks is called a clock domain. If a clock domain is not specified, Genus will assume all the clocks are in the same domain.

By default, Genus assigns clocks to `domain_1`, but you can create your own domain name with the `-domain` argument to `create_clock`.

The following example demonstrates how to create two different clocks and assign them to two separate clock domains:

```
create_clock -domain domain1 -name clk1 -period 720 [get_db ports *SYSCLK]
create_clock -domain domain2 -name clk2 -period 720 [get_db ports *CLK]
```

To remove clocks, use the `delete_obj` 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:

```
delete_obj $clock1
```

The following example shows how to remove the clock if you have not saved the clock object as a variable:

```
delete_obj [get_db clocks 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.

Importing Physical Information

You can supply physical information to Genus to drive synthesis. The type of information that you supply depends on the physical flow that you use.

For more information about the physical design, refer to *Genus Physical Guide.*

Applying Design Rule Constraints

When optimizing a design, Genus 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:

```
set_db [design|port] .max_transition value
```

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:

```
set_db [design|port] .max_fanout value
```

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:

```
set_db [design|port] .max_capacitance value
```

To specify a specific wireload model to be used during synthesis, use the `force_wireload` attribute. The following example specifies the `1x1` wireload model on a design named `top`:

```
set_db top .force_wireload 1x1
```

Creating Ideal Objects

An ideal object is an object that is free of any DRCs. For example, an ideal network would not have any maximum transition, maximum fanout, and capacitance constraints.

To idealize a particular network, specify the `ideal_network` attribute on the network's driving pin:

```
set_db pin:moniquea/inst2/foo .ideal_network true
```

Use the `is_ideal` attribute to check whether a specified network is ideal:

```
get_db net:moniquea/ck .is_ideal
```

By default, Genus will automatically idealize the following objects:

- Clock nets
- Asynchronous set/reset nets
- Test signals (`shift_enable` and `test_mode`), if they are defined *with* the `-ideal` option of DFT commands.
- The enable driver of isolation/combinational cells. We do not idealize data pin drivers.
- Drivers in the common power format (CPF) files.
- If the no propagate option is not set, then the `ideal_driver` attribute will be set to `true` on the always driver pin.
- `state_retention` control signals.

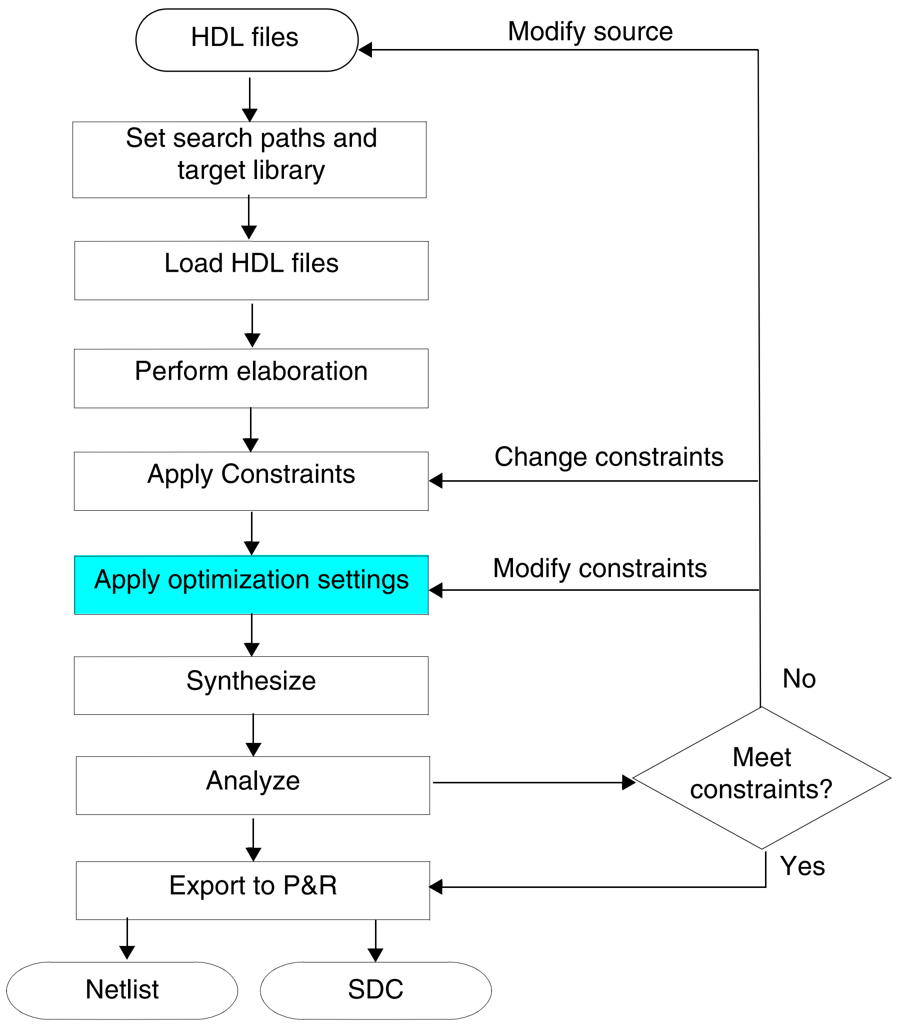
7

Defining Optimization Settings

- Overview on page 188
- Tasks
  - Preserving Instances and Modules on page 189
  - Grouping and Ungrouping Objects on page 190
    - Grouping on page 190
    - Ungrouping on page 191
  - Partitioning on page 193
  - Setting Boundary Optimization on page 194
  - Mapping to Complex Sequential Cells on page 196
  - Deleting Unused Sequential Instances on page 197
  - Controlling Merging of Combinational Hierarchical Instances on page 198
  - Optimizing Total Negative Slack on page 199
  - Making DRC the Highest Priority on page 200
  - Creating Hard Regions on page 201
    - Deleting Buffers and Inverters Driven by Hard Regions on page 201
    - Preventing Boundary Optimization through Hard Regions on page 202

Overview

This chapter describes how to apply optimization settings to your design before synthesis.



Preserving Instances and Modules

By default, Genus 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, by using the `preserve` attribute:

➤ To preserve hierarchical instances, type the following command:

```
set_db object .preserve true
```

where *`object`* is a hierarchical instance name.

➤ To preserve primitive instances, type the following command:

```
set_db object .preserve true
```

where *`object`* is a primitive instance name.

➤ To preserve modules or submodules, type the following command:

```
set_db object .preserve true
```

where *`object`* is a module or submodule name.

The default value of this attribute is `false`.

Genus can also simultaneously preserve instances and modules while allowing certain special actions to be performed on them. This allows for greater flexibility. For example:

- `The size_ok` argument enables Genus to preserve an instance (`g1` in the example below), while allowing it to be resized.

  ```
  set_db [get_db insts *g1] .preserve size_ok
  ```

- The `delete_ok` argument allows Genus to delete an instance (`g1` in the example below), but not to rename, remap, or resize it.

  ```
  set_db [get_db insts *g1] .preserve delete_ok
  ```

- `The size_delete_ok` argument allows Genus to resize or delete an instance (`g1` in the example below), but not to rename or remap it.

  ```
  set_db [get_db insts *g1] .preserve size_delete_ok
  ```

Tip

Preserving an instance within a hierarchy does not imply that the hierarchy is preserved or that the name of the preserved instance remains unchanged if its hierarchy changes. It is therefore important that constraint files and activity files are read in before you do any ungrouping on the design.

Grouping and Ungrouping Objects

Grouping and ungrouping are helpful when you need to change your design hierarchy as part of your synthesis strategy. Genus provides a set of commands that enable you to group or ungroup any existing instances, designs, or modules.

- *Grouping* builds a level of hierarchy around a set of instances.
- *Ungrouping* flattens a level of hierarchy.

Grouping

If your design includes several modules, you can group some of the module instances into another single module for placement or optimization purposes using the `group` command.

For example, the following command creates a new module called `CRITICAL_GROUP` that includes instances `I1` and `I2`.

```
group -name CRITICAL_GROUP [get_db insts *I1] [get_db insts *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/hinsts/CRITICAL_GROUPi
or
hinst:top_counter/CRITICAL_GROUPi
```

To change the suffix of the new instance name, set the following attribute:

```
set_db group_instance_suffix my_suffix
```

Ungrouping

Genus can automatically ungroup user hierarchies during synthesis, but you can also manually flatten a hierarchy in the design.

In either case, Genus will respect all preserved instances in the hierarchy. For more information on preserving instances, see "Preserving Instances and Modules" on page 189.

Tip

When the parent of a preserved leaf instance is ungrouped, the name of the preserved instance ***can*** change. It is therefore important that constraint files and activity files are read in before you attempt ungrouping.

Manual Ungrouping

To manually flatten a hierarchy in the design, use the `ungroup` command.

```
ungroup instance
```

where *`instance`* is the name of the instances to be ungrouped.

For example, 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:

```
ungroup -all [get_db insts *CRITICAL_GROUP]
```

If you defined an exception on a pin of a hierarchical instance that you ungroup, Genus tries to preserve the exception when you ungroup that hierarchical instance.

In most cases Genus adds a CDN_EXCEPTION buffer for each hierarchical instance pin for which an exception was defined to retain the exceptions on these pins.

In the following cases, no buffer is added:

- The driver of the hierarchical instance pin has only one fanout and the driver is the pin of a sequential or hierarchical instance.

  The exception of the hierarchical instance pin will be moved to the driver pin.

- The driver of the hierarchical instance pin has a multiple fanout but the load pin of the hierarchical instance pin is either a sequential or hierarchical instance pin.

  The exception of the hierarchical instance pin will be moved to the load pin.

Automatic Ungrouping

Genus can also automatically ungroup user hierarchies during synthesis.

During high effort RTL optimization (`syn_generic` with `syn_generic_effort` set to `high`) Genus ungroups user hierarchies containing only datapath hierarchies.

During high effort global mapping (`syn_map` with `syn_map_effort` set to `high`) Genus explores ungrouping of user hierarchies to expose more opportunities for structuring, mapping and other optimizations. Ungrouping is done **bottom up** and the ungrouping depends on the number of levels (logic depth) of the instances. The tool takes the wireload model into account before ungrouping the instance to make sure that the timing does not degrade after ungrouping. The tool distinguishes between critical and non-critical instances.

By default, Genus performs automatic ungrouping during high effort synthesis (`syn_generic` with `syn_generic_effort` set to `high` and `syn_map` with `syn_map_effort` set to `high`) to optimize for timing and area.

Note: Some ungrouping may occur with low or medium effort as well.

➡ To prevent automatic ungrouping, set the following root attribute before synthesis:

```
set_db auto_ungroup none
```

If the attribute is set to `both` (default), automatic ungrouping will happen during ***high*** effort mapping.

➡ To prevent ungrouping of all instances of a module, set the `ungroup_ok` attribute for the module to `false`:

```
set_db [get_db design:design .modules *name] .ungroup_ok false
```

➡ To prevent ungrouping of a specific instance, set the `ungroup_ok` attribute for the instance to `false`:

```
set_db [get_db design:design .insts *name] .ungroup_ok false
```

Partitioning

Partitioning is the process of disassembling (partitioning) designs into more manageable block sizes. This enables faster run-times and an improved memory footprint without sacrificing the accuracy of synthesis results. To enable partitioning, set the `auto_partition` attribute to `true` before synthesis:

```
set_db auto_partition true
```

Setting Boundary Optimization

Genus performs boundary optimization for all hierarchical instances in the design during synthesis. Examples of boundary optimizations include:

- Constant propagation across hierarchies

  This includes constant propagation through both input ports and output ports.
- Removing undriven or unloaded logic connected
- Collapsing equal and opposite pins

  Two hierarchical boundary pins are considered equal (opposite), if Genus determines that these pins always have the same (opposite or inverse) logic value.
- Hierarchical pin inversion

  Genus might invert the polarity of a hierarchical boundary pin to improve QoR. However it is not guaranteed, that this local optimization will always result in a global QoR improvement.
- Rewiring of equivalent signals across hierarchy

  Hierarchical boundary pins are feedthrough pins, if output pins always have the same (or inverted) logic value as an input pin. Such feedthrough pins can be routed around the module and no connections or logic is needed inside the module for these pins.

If two inputs or outputs of a module are identical, Genus can disconnect one of them and use the other output to drive the fanout logic for both. The disconnected pin is connected to constant 0. During incremental optimization, the tool determines whether to leave the pin unconnected or connected to constant 0 depending on the `driver_for_unloaded_hier_pins` root attribute' setting.

Genus can also rewire opposite signals which are functionally equivalent, but of opposite polarity.

You can control boundary optimization during synthesis using the following attributes:

- `boundary_opto`
- `delete_unloaded_seqs`
- `boundary_optimize_constant_hpins`
- `boundary_optimize_equal_opposite_hpins`
- `boundary_optimize_feedthrough_hpins`
- `boundary_optimize_invert_hpins`

➤ To disable boundary optimization on the module, type the following command:

```
set_db [get_db design:design_name .modules *name] .boundary_opto false
```

➤ To prevent Genus from removing flip-flops and logic if they are not transitively fanning out to output ports, use the `delete_unloaded_seqs` attribute:

```
set_db [modules or /] .delete_unloaded_seqs false
```

If you cannot perform top-down formal verification on the design, you should turn off boundary optimization for sub-blocks so that they can be individually verified.

For hierarchical formal verification of designs with inverted boundary pins, the verification tool uses information about inverted pins. For the Conformal® Equivalence Checking tool the necessary naming rule is generated automatically via the `write_do_lec` command.

Mapping to Complex Sequential Cells

The sequential mapping feature of Genus 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).

Genus performs sequential mapping when the flops are inferred in RTL. For instantiated flops, other than sizing, Genus 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:

```
set_db hdl_ff_keep_feedback true
```

Setting this attribute may have a negative impact on the area and timing.

```
set_db optimize_constant_1_flops false
```

Deleting Unused Sequential Instances

Genus 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`.

  ```
  set_db 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`.

  ```
  set_db 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 `true`.

  ```
  set_db 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`.

  ```
  set_db optimize_constant_latches false
  ```

Controlling Merging of Combinational Hierarchical Instances

By default Genus merges combinational hierarchical instances during RTL optimization (`syn_generic`) and mapping (`syn_map`).

➤ To prevent merging of all combinational hierarchical instances, set the `merge_combinational_hier_instances` root attribute to `false`.

➤ To control whether a specific instance can be merged use the `merge_combinational_hier_instance` instance attribute.

  You can specify the following values:

  `false`—Prevents merging of this combinational hierarchical instance.

  `inherited`—If the instance is a combinational hierarchical instance, it inherits the value of the `merge_combinational_hier_instances` root attribute.

  `true`—Allows merging of this combinational hierarchical instance.

For example to prevent merging on an instance, specify

```
set_db [get_db design:design .insts *name] .merge_combinational_hier_instance false
```

Optimizing Total Negative Slack

By default, Genus 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, Genus will not work on the other paths if it cannot improve timing on the WNS.

➤ To make Genus work on all the paths to reduce the total negative slack (TNS), instead of just WNS, type the following command:

```
set_db tns_opto true
```

Ensure that you specify the attribute on the root-level. This attribute instructs Genus 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.

Making DRC the Highest Priority

By default, Genus 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 Genus 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:

```
set_db drc_first true
```

By default, this attribute is `false`, which means that DRCs will not be fixed if it introduces timing violations.

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 technology mapping.

To create hard regions, follow these steps:

1. Specify the hard region, for example `pbu_ctl`:
   ```
   set_db [get_db insts *pbu_ctl] .hard_region 1
   ```
2. Eliminate buffers and inverter trees between hard regions using the following variable:
   ```
   set map_rm_hr_driven_buffers 1
   ```
3. Run the `syn_map` command.

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, use the following command:
```
set map_rm_hr_driven_buffers 1
```

This instructs Genus 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 Genus 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.

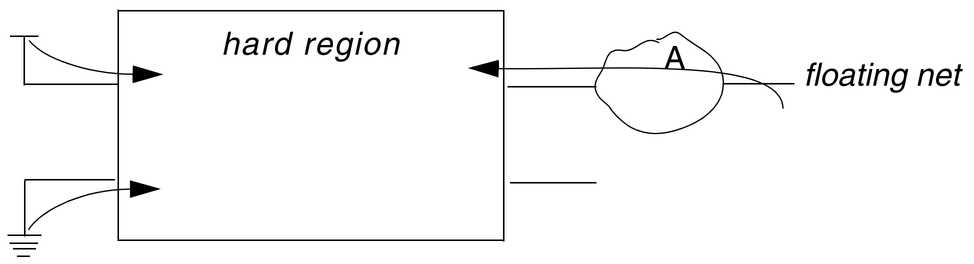
Note: Timing may become worse due to this buffer removal. This clean-up phase occurs during the last step of incremental optimization.

Preventing Boundary Optimization through Hard Regions

The regular boundary optimization techniques also apply through hard regions. In this case, boundary optimization can propagate inwards the hard region but not outwards.

For example, if logic A connected to an output of the hard region is optimized away (for example, dead logic removed), the optimization propagates inside the hard region. Optimization inside the hard region does not get propagated outside the hard region.

Figure 7-1 Boundary optimization through a hard region



If you want to prevent boundary optimization into the hard regions, set the following attribute before performing synthesis:

```
set_db subdesign_of_hard_region .boundary_opto false
```

8

Reducing Runtime Using Super-Threading

- Overview on page 204
- Licensing and CPU Usage on page 205
- Using Super-Threading on Local Host on page 206
- Using Super-Threading on Remote Shell on page 206
- Using Super-Threading on Platform Load Sharing Facility (LSF) on page 207
- Using Super-Threading on Sun Grid Engine (SGE) on page 209

Overview

Genus has a massively parallel architecture which allows parallel synthesis (multi-threading, super-threading, and distributed processing) on the same machine or on different machines across the network.

You can specify to distribute jobs on multiple machines (M), and also to use multiple CPUs (N) per machine.

➡ To specify the number of machines (M), use the following command:

```
set_db super_thread_servers {machine_names}
```

Note: *machine_names* can be a real machine names, `batch`, or `localhost`.

The `super_thread_servers` attribute specifies the set of machines on which to launch remote server processes. If no machines are specified, a default value of `localhost` is assumed.

➡ To specify the number of CPUs (N) per machine, use the following command:

```
set_db max_cpus_per_server integer
```

➡ To specify an upper bound for the waiting time (in minutes) to obtain the required resources, use the following command:

```
set_db st_launch_wait_time integer
```

By default, the waiting time is set to 10 minutes. However, a bigger value gives you more opportunities to obtain the resource.

As soon as all required resources are obtained, Genus will start the jobs. Otherwise, Genus will go on with the obtained resources after this waiting time.

The following sections discuss licensing usage and show how Genus supports parallelization

Licensing and CPU Usage

Note: The Genus base license allows the use of 8 CPUs.

When you start with a Genus license, the initial license will give you access to eight remote server processes.

Each subsequent license, gives you access to eight more CPUS.

Table 8-1 CPU Access

Product/Option	Number of remote server processes enabled by	
	First license	Subsequent license
Genus (GEN100)	8	8
Genus CPU Accelerator Option GEN80)	8	8

For example, when you request 16 servers in total, Genus will automatically look for another license on the server in this order:

- ❑ Genus_CPU_ Option
- ❑ Genus_Synthesis

Recommendations:

- For designs with less than 1.5 million instances, use 8 CPUs
- For designs with over 1.5 million, up to 5 million instances, use 8 to 16 CPUs
- For designs with more than 5 million instances, use 16 to 32 CPUs
- For best performance results, use machines with similar configurations.

Using Super-Threading on Local Host

To enable super-threaded optimization, use the `max_cpus_per_server` attribute to control the maximum number of available CPUs on each given machine.

Using 16 CPUs on a machine

```
set_db max_cpus_per_server 16
set_db super_thread_servers "localhost"
```

Using Super-Threading on Remote Shell

By default, Genus launches the remote server processes using the UNIX command `rsh`. For security reasons, some hosts do not allow you to use the `rsh` command to connect to them, but they might allow you to use another command, such as `ssh`.

➡ To specify the preferred alternative to `rsh`, use the following command:

```
    genus:root: 1> set_db super_thread_rsh_command rsh_command
```

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_command machine_name echo hello world
```

where *`rsh_command`* is the value of the `super_thread_rsh_command` attribute.

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.

Using 16 CPUs on one machine

```
set_db max_cpus_per_server 16
set_db super_thread_servers "remote_host1"
set_db super_thread_rsh_command rsh
```

Using 8 CPUs on two machines

```
set_db max_cpus_per_server 8
set_db super_thread_servers "remote_host1 remote_host2"
set_db super_thread_rsh_command rsh
```

Note: You can mix `localhost` and remote host:

```
set_db max_cpus_per_server 8
set_db super_thread_servers "localhost remote_host"
set_db super_thread_rsh_command rsh
```

This example uses 8 CPUs on `localhost` and 8 CPUs on remote host.

Using Super-Threading on Platform Load Sharing Facility (LSF)

➡ To launch jobs to a queuing system, like LSF, you need to retrieve the available queue clusters in your environment or network. Use this UNIX command to retrieve such clusters:

```
unix> qconf -sql
```

➡ To super-thread on LSF queuing systems, use the `batch` argument with the `super_thread_servers` attribute.

➡ To pass commands to the queuing system, use the `super_thread_batch_command` and `super_thread_kill_command` attributes.

```
set_db super_thread_batch_command \
{bsub -q lnx-penny -o /dev/null -J RC_server}
set_db super_thread_kill_command {bkill}
```

Using 16 CPUs on one machine

```
set_db max_cpus_per_server 16
set_db super_thread_servers "batch"
set_db super_thread_batch_command {bsub -n 16 -q ee50 \
    -R "span\[hosts=1\]"}
set_db super_thread_kill_command {bkill}
```

Using 8 CPUs on two machines

```
set_db max_cpus_per_server 8
set_db super_thread_servers [string repeat "batch" 2]
set_db super_thread_batch_command {bsub -n 8 -q ee50 \
    -R "span\[hosts=1\]"}
set_db super_thread_kill_command {bkill}
```

Note: You can mix `localhost` and LSF:

```
set_db max_cpus_per_server 8
set_db super_thread_servers "localhost batch"
set_db super_thread_batch_command {bsub -n 8 -q ee50 \
    -R "span\[hosts=1\]"}
set_db super_thread_kill_command {bkill}
```

This example uses 8 CPUs on `localhost` and 8 CPUs on LSF.

Notes on the bsub Options

1. `-n` *`M`* is required to ensure proper working when requesting more than one CPU per server.

   `M` must correspond to the value specified with the `max_cpus_per_server` attribute.

2. To ensure that the FARM returned machine is suitable for Genus, you must specify the correct queue (`-q`) and the resource requirements with `-R`.
3. If you need to specify other options to `bsub` that contain brackets (“[ ]”) in the argument values, you need to use escape characters (“\”) on the brackets to prevent Tcl from evaluating the content of the expression.
4. `"span\[hosts=1\]"` is required to reserve all requested CPUs on the same machine for each batch.

Note: For more information on the `bsub` command, refer to Product documentation (manuals) for Platform LSF.

Using Super-Threading on Sun Grid Engine (SGE)

➡ To launch jobs to a queuing system, like LSF, you need to retrieve the available queue clusters in your environment or network. Use this UNIX command to retrieve such clusters:

```
unix> qconf -sql
```

➡ To super-thread on LSF queuing systems, use the `batch` argument with the `super_thread_servers` attribute.

➡ To pass commands to the queuing system, use the `super_thread_batch_command` and `super_thread_kill_command` attributes.

```
genus:root: 1> set_db super_thread_batch_command \
    {qsub -N RC_server -q lnx-penny -b y -j y -o /dev/null}
genus:root: 2> set_db super_thread_kill_command {qdel}
```

Using 16 CPUs on one machine

```
set_db max_cpus_per_server 16
set_db super_thread_servers "batch"
set_db super_thread_batch_command \
    {qsub -hard -N GenusST -q ee50 -b y -j y -pe pe_name 16}
set_db super_thread_kill_command {qdel}
```

Using 8 CPUs on two machines

```
set_db max_cpus_per_server 8
set_db super_thread_servers [string repeat "batch" 2]
set_db super_thread_batch_command \
    {qsub -hard -N GenusST -q ee50 -b y -j y -pe pe_name 8}
set_db super_thread_kill_command {qdel}
```

Note: You can mix `localhost` and SGE:

```
set_db max_cpus_per_server 8
set_db super_thread_servers "localhost batch"
set_db super_thread_batch_command \
    {qsub -hard -N GenusST -q ee50 -b y -j y -pe pe_name 8}
set_db super_thread_kill_command {qdel}
```

This example uses 8 CPUs on `localhost` and 8 CPUs on SGE.

Notes on the qsub Options

1. `-pe` *`pe_name M`* is required to ensure proper working when requesting more than one CPU per server

   `M` must correspond to the value specified with the `max_cpus_per_server` attribute.

pe_name is the name of the suitable parallel environment (pe) which limits all slots (CPUs) to be with one host machine.

To determine the suitable *pe_name,* do the following:

- ❑ Use `"qconf -spl"` to list all currently defined *pe_names*.
- ❑ Use "`qconf -sp` *pe_name*" to check the configuration information for the specified *pe_name*.

  Use the *pe_name* for which you see the following in its information:

  ```
  allocation_rule $pe_slots
  ```

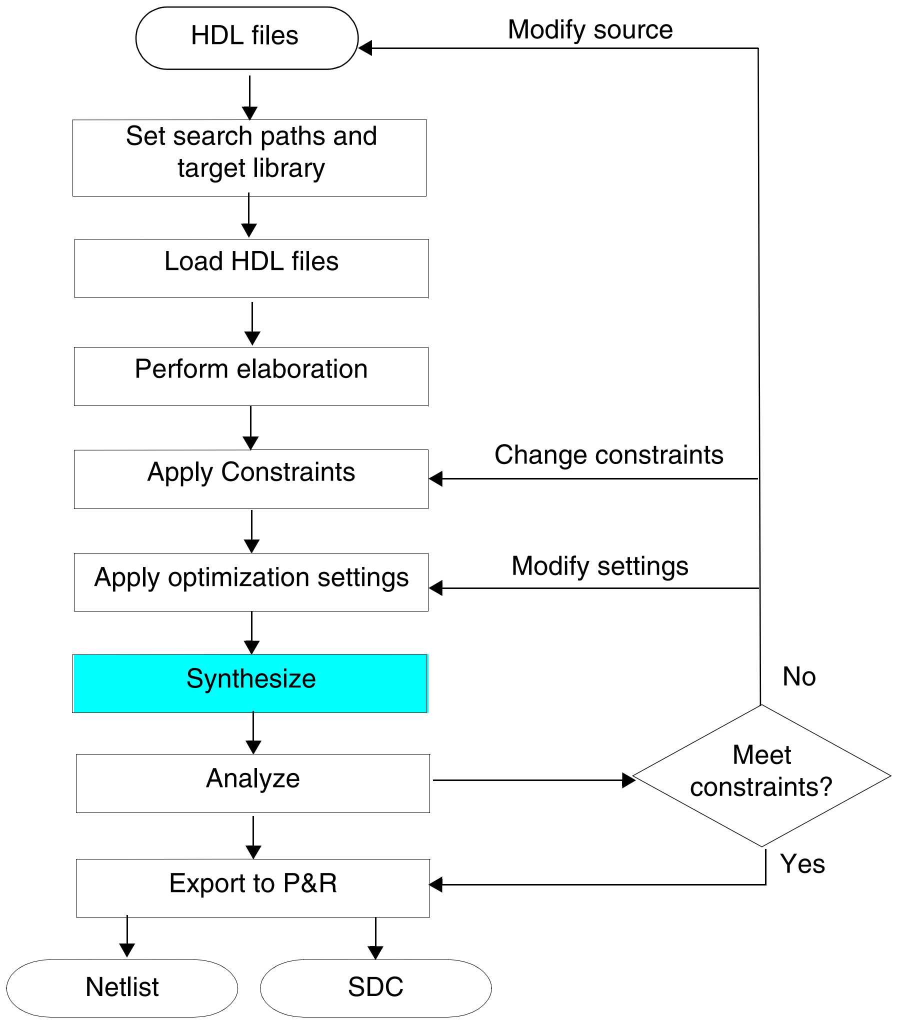
2. `-b y` is required to permit appending a binary command after the batch command.
3. `-N` (optional) specifies the job name.
4. If you need to specify other options to `qsub` that contain brackets ("[ ]") in the argument values, you need to use escape characters ("\") on the brackets to prevent Tcl from evaluating the content of the expression.

Note: For more information on the `qsub` command, refer to the Sun Grid Engine Reference.

9 Performing Synthesis

- Overview on page 212
  - RTL Optimization on page 213
  - Global Focus Mapping on page 213
  - Global Incremental Optimization on page 213
  - Incremental Optimization (IOPT) on page 214
- Tasks on page 215
  - Synthesizing your Design on page 216
  - Setting Effort Levels on page 221
  - Quality of Silicon Prediction on page 222
  - Generic Gates in a Generic Netlist on page 223
  - Writing the Generic Netlist on page 228
  - Reading the Netlist on page 238
  - Analyzing the Log File on page 239

Overview



Synthesis is the process of transforming your HDL design into a gate-level netlist, given all the specified constraints and optimization settings.

In Genus, synthesis involves the following four processes:

- RTL Optimization on page 213
- Global Focus Mapping on page 213
- Global Incremental Optimization on page 213
- Incremental Optimization (IOPT) on page 214

RTL Optimization

During RTL optimization, Genus performs optimizations like datapath synthesis, resource sharing, speculation, mux optimization, and carrysave arithmetic (CSA) optimizations. After this step, Genus performs logic optimizations like structuring and redundancy removal.

For more information on datapath synthesis, see *Genus Datapath Synthesis Guide*.

Global Focus Mapping

Genus performs global focus mapping at the end of the RTL technology-independent optimizations (during the `syn_map` command).

This step includes restructuring and mapping the design concurrently, including optimizations like splitting, pin swapping, buffering, pattern matching, and isolation.

Global Incremental Optimization

After global focus mapping, Genus performs synthesis global incremental optimization. This phase is mainly targeted at area optimization and power optimization (if enabled). Optimizations performed in this phase include global sizing of cells and optimization of buffer trees.

Incremental Optimization (IOPT)

The final optimization Genus performs is incremental optimization. Optimizations performed during IOPT improve timing and area and fix DRC violations.

Optimizations performed during this phase include multibit cell mapping, incremental clock gating, incremental retiming, tie cell insertion, and assign removal.

For more information on multibit cell mapping, refer to Mapping to Multibit Cells in the *Genus Synthesis Flows Guide*.

By default, timing has the highest priority and Genus will not fix DRC violations if doing so causes timing violations. This priority can be overridden by setting the `drc_first` attribute to `true`. In this case, all violations will be fixed as well as those paths with positive slack.

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 `syn_opt` command. It is asserted by specifying the `high` effort level.

If for some reason you need to cancel the Genus session in the middle of IOPT, press the `Ctrl-c` key sequence. You will be given a warning message with a particular IOPT state and brought back to the command line. Next time you enter a Genus session (with the same commands, constraints, script, etc. that preceded the `ctrl-c` halt) you can specify the IOPT state at which you stopped with the `stop_at_iopt_state` attribute. Genus will continue with the netlist it had generated at the specified state.

Tasks

- Synthesizing your Design on page 216
    - Synthesizing Submodules on page 218
    - Synthesizing Unresolved References on page 219
    - Re-synthesizing with a New Library (Technology Translation) on page 219
- Setting Effort Levels on page 221
- Quality of Silicon Prediction on page 222
- Generic Gates in a Generic Netlist on page 223
    - Generic Flop on page 224
    - Generic Latch on page 225
    - Generic Mux on page 225
    - Generic Dont-Care on page 227
    - Writing the Generic Netlist on page 228
    - Reading the Netlist on page 238

Synthesizing your Design

After you set the constraints and optimizations for your design, you can proceed with synthesis. 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 performed by the `syn_generic` and `syn_map` commands (see Table 9-1):

- The `syn_generic` command performs RTL optimization on your design.
- The `syn_map` 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 the `syn_map` command are:

  - Technology-independent Boolean optimization
  - Technology mapping
  - Technology-dependent gate optimization

  The `syn_map` command queries the library for detailed timing information. After you use the `syn_map` command to generate an optimized netlist, you can analyze the netlist using the `report_*` commands and output it to a file using the `write_*` commands. For more information on the `write_*` commands, see "Writing Out the Design Netlist" on page 302.

  Table 9-1 shows a matrix of actions performed by synthesis depending on the state of the design and the option specified.

Table 9-1 Actions Performed by the syn_* Commands

Specified Command	**Current Design State**		
	RTL	**Generic**	**Mapped**
`syn_generic`	■ RTL Optimization		■ Unmapping
`syn_map`	■ RTL Optimization ■ Mapping ■ Incremental optimizations	■ Mapping ■ Incremental optimizations	■ Unmapping ■ Mapping ■ Incremental optimizations
`syn_opt -physical`	■ RTL Optimization ■ Mapping ■ Placement ■ Post-placement incremental optimizations	■ Mapping ■ Placement ■ Post-placement incremental optimizations	■ Placement ■ Post-placement incremental optimizations

Synthesizing Submodules

In Genus you can have multiple designs, each with its own design hierarchy. You can synthesize any of these top-level designs separately.

Whenever you need to synthesize any submodule in your design hierarchy, use the `create_derived_design` command to promote this module 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:

   ```
   genus:root: 1> elaborate module_top
   ```

2. Apply constraints.

3. Synthesize the design to gates using the `low` effort level (to more accurately extract the constraints):

   ```
   set_db syn_generic_effort low
   syn_generic
   set_db syn_map_effort low
   syn_map
   ```

4. Promote the submodule into a top-level module using the `create_derived_design` command:

   ```
   create_derived_design -name <new_top> <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.

   ```
   vls /designs
   ./    module_top/    new_top/
   ```

5. Write out the *new_top* design constraints using the `write_script` command:

   ```
   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):

   ```
   delete_obj new_top
   read_hdl <read_RTL_files>
   elaborate <new_top>
   include new_top.con
   syn_generic
   syn_map
   ```

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:

```
syn_map /designs/new_top
```

Synthesizing Unresolved References

In Genus, 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 blackboxes, 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. Genus will maintain any logic leading into or out of the I/Os of unresolved references and treat them as unconstrained.

For details regarding the resolution of these unresolved references, refer to *Modeling Logic Abstracts* in the *Genus HDL Modeling Guide*.

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 technology library files

   ```
   read_libs <list of technology library .lib files>
   ```

2. Read-in the mapped netlist using the `read_netlist` command:

   ```
   read_netlist mapped_netlist.v
   ```

3. 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:

   ```
   foreach cell [get_db lib_cells *] { \
   ==> set_db $cell .preserve false; \
   ==> set_db $cell .avoid false \
   ==> }
   ```

4. Unmap the netlist to generic gates using the `syn_generic` command:

   ```
   syn_generic
   ```

5. Write-out the generic netlist:

   ```
   write_hdl -generic > generic_netlist.v
   ```

6. Remove the design from the design information hierarchy:

```
delete_obj [get_db designs]
```

7. Set the new technology library:

```
read_libs new_library.lib
```

8. Re-read and elaborate the generic netlist

```
read_hdl generic_netlist.v
elaborate
```

9. Apply constraints and synthesize to technology mapped gates using the following commands:

```
syn_generic
syn_map
```

After the final step, proceed with your Genus session.

Setting Effort Levels

You can specify an effort level by setting the `syn_generic_effort`, `syn_map_effort`, and `syn_opt_effort` attributes. The possible values for the effort attributes are as follows:

- `low`

  The design is mapped to gates, but Genus does very little RTL optimization, incremental clean up, DRC fixing, or redundancy identification and removal. The low setting is generally not recommended.

- `medium` (default setting)

  Genus performs better timing-driven structuring, incremental synthesis, and redundancy identification and removal on the design.

- `high`

  Genus 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.

If you wish to set the same value for all the three attributes, you can use the common attribute `syn_global_effort` to simultaneously set the effort level for the three synthesis stages.

Quality of Silicon Prediction

Predict the quality of silicon through the `syn_opt -physical` command. This prediction process enhances the correlation between results from place and route pre-clock tree synthesis and the results from Genus.

Specifically, the `syn_opt -physical` 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 `syn_opt -physical` command will operate in incremental mode if the standard cells are placed. The `syn_opt -physical` command will perform virtual buffering by default.

For more information, refer to the Genus-P Flow in *Genus Physical Guide*

Generic Gates in a Generic Netlist

Genus 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 Genus.

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, Genus 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 = 1’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, sel2, data2, z);
    input sel0, data0, sel1, data1, sel2, 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 -generic` 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 -generic` command may produce a netlist that has a mixture of Verilog primitives and Genus 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_db library tutorial.lib
set_db hdl_ff_keep_feedback false
read_hdl test.v
elaborate
write_hdl -generic
```

Setting the `hdl_ff_keep_feedback` attribute to `false` tells Genus to use the `sena` logic inside of the generic flop to implement the load enable logic. If you do not set this attribute, Genus 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 9-1, Genus produces a netlist, such as shown in Example 9-2.

Example 9-1 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 9-2 Generic Netlist From Example 9-1

```
module bmux(ctl, in_0, in_1, z);
  input ctl, in_0, in_1;
  output z;
  wire ctl, in_0, in_1;
  wire z;
  CDN_bmux2 g1(.sel0 (ctl), .data0 (in_0), .data1 (in_1), .z (z));
endmodule
module test(q, d, clk, rstn, enb);
  input d, clk, rstn, enb;
  output q;
  wire d, clk, rstn, enb;
  wire q;
  wire UNCONNECTED, n_2;
  bmux mux_q_6_7(.ctl (n_2), .in_0 (d), .in_1 (1'b0), .z (UNCONNECTED));
  not g1 (n_2, rstn);
  CDN_flop q_reg(.clk (clk), .d (d), .sena (enb), .aclr (1'b0), .apre
       (1'b0), .srl (n_2), .srd (1'b0), .q (q));
endmodule
`ifdef RC_CDN_GENERIC_GATE
`else
  module CDN_flop(clk, d, sena, aclr, apre, srl, srd, q);
  input clk, d, sena, aclr, apre, srl, srd;
  output q;
  wire clk, d, sena, aclr, apre, srl, srd;
  wire 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
`endif
`ifdef RC_CDN_GENERIC_GATE
`else
```

```
`ifdef ONE_HOT_MUX
module CDN_bmux2(sel0, data0, data1, z);
  input sel0, data0, data1;

  output z;
  wire sel0, data0, data1;
  reg  z;
  always
    @(sel0 or data0 or data1)
      case ({sel0})
       1'b0: z = data0;
       1'b1: z = data1;
      endcase
endmodule
`else
module CDN_bmux2(sel0, data0, data1, z);
  input sel0, data0, data1;
  output z;
  wire sel0, data0, data1;
  wire z;
  wire inv_sel0, w_0, w_1;
  not i_0 (inv_sel0, sel0);
  and a_0 (w_0, inv_sel0, data0);
  and a_1 (w_1, sel0, data1);
  or org (z, w_0, w_1);
endmodule
`endif // ONE_HOT_MUX
`endif
```

Using the `sync_set_reset` pragma tells Genus 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, Genus 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 9-3, Genus produces a netlist, such as shown in Example 9-4.

Example 9-3 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 9-4 Generic Netlist From Example 9-3

```
module bmux(ctl, in_0, in_1, z);
  input ctl, in_0, in_1;
  output z;
  wire ctl, in_0, in_1;
  wire z;

  CDN_bmux2 g1(.sel0 (ctl), .data0 (in_0), .data1 (in_1), .z (z));
endmodule
```

```
module test(q, d, clk, rstn, enb);
  input d, clk, rstn, enb;
  output q;
  wire d, clk, rstn, enb;
  wire q;
  wire UNCONNECTED, n_2;
  bmux mux_q_7_6(.ctl (n_2), .in_0 (d), .in_1 (1'b0), .z (UNCONNECTED));
  not g1 (n_2, rstn);
  CDN_flop q_reg(.clk (clk), .d (d), .sena (enb), .aclr (n_2), .apre
       (1'b0), .srl (1'b0), .srd (1'b0), .q (q));
endmodule

`ifdef RC_CDN_GENERIC_GATE
`else
module CDN_flop(clk, d, sena, aclr, apre, srl, srd, q);
  input clk, d, sena, aclr, apre, srl, srd;
  output q;
  wire clk, d, sena, aclr, apre, srl, srd;
  wire 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`endif
`ifdef RC_CDN_GENERIC_GATE
`else
`ifdef ONE_HOT_MUX
module CDN_bmux2(sel0, data0, data1, z);
  input sel0, data0, data1;
  output z;
  wire sel0, data0, data1;

reg  z;

always
    @(sel0 or data0 or data1)
      case ({sel0})
       1'b0: z = data0;
       1'b1: z = data1;
      endcase
endmodule
`else
module CDN_bmux2(sel0, data0, data1, z);
  input sel0, data0, data1;
  output z;
  wire sel0, data0, data1;
  wire z;
  wire inv_sel0, w_0, w_1;
  not i_0 (inv_sel0, sel0);
  and a_0 (w_0, inv_sel0, data0);
  and a_1 (w_1, sel0, data1);

o
 org (z, w_0, w_1);
endmodule
`endif // ONE_HOT_MUX
`endif
```

CDN_latch

With the following RTL code, as shown in Example 9-5, Genus produces a netlist, such as the one shown in Example 9-6.

Example 9-5 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 9-6 Generic Netlist From Example 9-5

```
module bmux(ctl, in_0, in_1, z);
  input ctl, in_0, in_1;
  output z;
  wire ctl, in_0, in_1;
  wire z;
  CDN_bmux2 g1(.sel0 (ctl), .data0 (in_0), .data1 (in_1), .z (z));
endmodule

module test(q, d, g, rstn);
  input d, g, rstn;
  output q;

wire d, g, rstn;
  wire q;
  wire UNCONNECTED, n_2;
  bmux mux_q_6_5(.ctl (n_2), .in_0 (d), .in_1 (1'b0), .z (UNCONNECTED));
  not g1 (n_2, rstn);
  CDN_latch q_reg(.d (d), .ena (g), .aclr (n_2), .apre (1'b0), .q (q));
endmodule

`ifdef RC_CDN_GENERIC_GATE
`else
module CDN_latch(ena, d, aclr, apre, q);
  input ena, d, aclr, apre;
  output q;
```

```
wire ena, d, aclr, apre;
  wire 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 <= 1'b0;
endmodule
`endif
`ifdef RC_CDN_GENERIC_GATE
`else
`ifdef ONE_HOT_MUX
module CDN_bmux2(sel0, data0, data1, z);
  input sel0, data0, data1;
  output z;
  wire sel0, data0, data1;
  reg  z;
  always
    @(sel0 or data0 or data1)
      case ({sel0})
       1'b0: z = data0;
       1'b1: z = data1;
      endcase
endmodule
`else
module CDN_bmux2(sel0, data0, data1, z);
  input sel0, data0, data1;
  output z;
  wire sel0, data0, data1;
  wire z;

wire inv_sel0, w_0, w_1;
  not i_0 (inv_sel0, sel0);
  and a_0 (w_0, inv_sel0, data0);
  and a_1 (w_1, sel0, data1);
  or org (z, w_0, w_1);
endmodule
`endif // ONE_HOT_MUX
`endif
```

Using the `async_set_reset` pragma tells Genus 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, Genus 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 9-7, Genus produces a netlist, such as shown in Example 9-8.

Example 9-7 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 9-8 Generic Netlist From Example 9-7

```
module bmux(ctl, in_0, in_1, z);
  input ctl;
  input [2:0] in_0, in_1;
  output [2:0] z;
  wire ctl;
  wire [2:0] in_0, in_1;
  wire [2:0] z;
  CDN_bmux2 g1(.sel0 (ctl), .data0 (in_0[2]), .data1 (in_1[2]), .z
       (z[2]));
  CDN_bmux2 g2(.sel0 (ctl), .data0 (in_0[1]), .data1 (in_1[1]), .z
       (z[1]));
  CDN_bmux2 g3(.sel0 (ctl), .data0 (in_0[0]), .data1 (in_1[0]), .z
       (z[0]));
endmodule


odule test(y, a, b, s);
  input [2:0] a, b;
  input s;
  output [2:0] y;
  wire [2:0] a, b;
  wire s;
  wire [2:0] y;
  bmux mux_5_12(.ctl (s), .in_0 (a), .in_1 (b), .z (y));
endmodule

`ifdef RC_CDN_GENERIC_GATE
`else
`ifdef ONE_HOT_MUX
module CDN_bmux2(sel0, data0, data1, z);
  input sel0, data0, data1;
  output z;
  wire sel0, data0, data1;
  reg  z;
  always
    @(sel0 or data0 or data1)
      case ({sel0})
       1'b0: z = data0;
       1'b1: z = data1;
      endcase
endmodule
```

```
`else
module CDN_bmux2(sel0, data0, data1, z);
  input sel0, data0, data1;
  output z
  wire sel0, data0, data1;
  wire z;
  wire inv_sel0, w_0, w_1;
  not i_0 (inv_sel0, sel0);
  and a_0 (w_0, inv_sel0, data0);
  and a_1 (w_1, sel0, data1);
  or org (z, w_0, w_1);
endmodule
`endif // ONE_HOT_MUX
`endif
```

With the following RTL code, as shown in Example 9-9, Genus produces a netlist, such as shown in Example 9-10.

Example 9-9 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 9-10 Generic Netlist From Example 9-9

```
module bmux(ctl, in_0, in_1, in_2, in_3, in_4, z);
  input [2:0] ctl, in_0, in_1, in_2, in_3, in_4;
  output [2:0] z;
  wire [2:0] ctl, in_0, in_1, in_2, in_3, in_4;
  wire [2:0] z;
  CDN_bmux5 g1(.sel0 (ctl[0]), .data0 (in_0[2]), .data1 (in_1[2]),
       .sel1 (ctl[1]), .data2 (in_2[2]), .data3 (in_3[2]), .sel2
       (ctl[2]), .data4 (in_4[2]), .z (z[2]));
  CDN_bmux5 g2(.sel0 (ctl[0]), .data0 (in_0[1]), .data1 (in_1[1]),
       .sel1 (ctl[1]), .data2 (in_2[1]), .data3 (in_3[1]), .sel2
       (ctl[2]), .data4 (in_4[1]), .z (z[1]));
  CDN_bmux5 g3(.sel0 (ctl[0]), .data0 (in_0[0]), .data1 (in_1[0]),
       .sel1 (ctl[1]), .data2 (in_2[0]), .data3 (in_3[0]), .sel2
       (ctl[2]), .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, s;
  output [2:0] y;
  wire [2:0] a, b, c, d, e, s;
  wire [2:0] y;
  bmux mux_y_7_7(.ctl (s), .in_0 (a), .in_1 (b), .in_2 (c), .in_3 (d),
       .in_4 (e), .z (y));
endmodule

`ifdef RC_CDN_GENERIC_GATE
`else
`ifdef ONE_HOT_MUX
module CDN_bmux5(sel0, data0, data1, sel1, data2, data3, sel2, data4,
     z);
  input sel0, data0, data1, sel1, data2, data3, sel2, data4;
  output z;
  wire sel0, data0, data1, sel1, data2, data3, sel2, data4;
  reg  z;
  always
    @(sel0 or sel1 or sel2 or data0 or data1 or data2 or data3 or
        data4)
      case ({sel0, sel1, sel2})
       3'b000: z = data0;
       3'b100: z = data1;
       3'b010: z = data2;
       3'b110: z = data3;
       3'b001: z = data4;
       default: z = 1'bX;
      endcase
endmodule
`else
module CDN_bmux5(sel0, data0, data1, sel1, data2, data3, sel2, data4,
     z);
  input sel0, data0, data1, sel1, data2, data3, sel2, data4
  output z;
  wire sel0, data0, data1, sel1, data2, data3, sel2, data4;
  wire z;
  wire inv_sel0, inv_sel1, inv_sel2, w_0, w_1, w_2, w_3, w_4;
  not i_0 (inv_sel0, sel0);
  not i_1 (inv_sel1, sel1);
  not i_2 (inv_sel2, sel2);
  and a_0 (w_0, inv_sel2, inv_sel1, inv_sel0, data0);
  and a_1 (w_1, inv_sel2, inv_sel1, sel0, data1);
  and a_2 (w_2, inv_sel2, sel1, inv_sel0, data2);

  and a_3 (w_3, inv_sel2, sel1, sel0, data3);
  and a_4 (w_4, sel2, inv_sel1, inv_sel0, data4);
  or org (z, w_0, w_1, w_2, w_3, w_4);
endmodule
`endif // ONE_HOT_MUX
`endif
```

Reading the Netlist

This section applies to both the `read_hdl` command and the `read_hdl -netlist` command.

As described in Chapter 6.2 of IEEE Std 1364.1-2002, Genus, by default, has a macro named `SYNTHESIS` defined. Therefore, Genus does not see the description of generic gates and Genus 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, Genus uses the built-in generic definition.
- If your definition is found in the input HDL code, Genus does not try to identify whether it is the same as (or equivalent to) what the `write_hdl -generic` command writes out; Genus synthesizes your description.

In a bottom-up structural flow, the following scenario can happen.
At an early stage of netlist loading, Genus cannot resolve a `CDN_flop`, `CDN_latch`, `CDN_mux`*, `CDN_dc` instantiation. Therefore, Genus uses the built-in generic definition for it. At a later stage of netlist loading, Genus 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.

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 239
- Reporting Area in the Log File on page 239
- Incremental Optimization on page 240
- Reporting Run Time on page 241
- Generating Target Timing Values on page 241
- Global Map Report on page 243
- Tracking Total Negative Slack on page 244

Status Messages

During certain processes, like optimization, Genus 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 convey 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 275 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 `syn_map` 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 `real_runtime` attribute. This feature is available at any time and does not include the run time of the query itself. This is a root attribute.

➤ To report Genus's design process time in CPU seconds, not actual clock time, type the following command:

```
get_db real_runtime
```

The value is printed to `stdout`. To format the output, use the following command:

```
puts "The RUNTIME is [get_db real_runtime]"
```

➤ To report memory utilization, type the following command:

```
get_db memory_usage
```

or

```
puts "The MEMORY USAGE is [get_db memory_usage]"
```

Genus will return the memory usage in kilobytes.

Tip

To get reference points throughout the synthesis process, use these commands in your script after elaboration, `syn_generic`, `syn_map`, and at the end of the session.

Generating Target Timing Values

Target timing values help you determine whether the design goals are realistic. Genus 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 Genus proceed with synthesis.

A target path is not printed by default. It will be printed only when the value of the attribute `information_level` (*default*: 1) is set to be more than 2.

The log file will have the output similar to the example shown below.

```
Cost Group 'C2C':-
max rise (Pin: top_execexec/top_execmac/mac_reg/reg_out_[15]/d)    target   97

    Pin                         Type        Fanout  Load Arrival
                                                    (fF)   (ps)
-------------------------------------------------------------------------------

(clock gg_clk)                   <<<  launch                          0 R

top_execexec

reg_out_[0]/clk
        reg_out_[0]/q           (u)  unmapped_d_flop     36  30.0
    top_execm_8x8_n_reg_0/reg_out[0]
        g690/in_0
        g690/z                  (u)  unmapped_not        40 200.0
        cb_parti689/top_execpart_gen_ll_4_select_dup[2]
    top_execpart_gen_ll_5/select_dup[0]

....

       g_t830/z                 (u)  unmapped_nand2       3  15.0
      top_execcmp62_wl_high/x2[6]

....

        g_t2438/z               (u)  unmapped_nand2       5  25.0
    top_execcmp42_wl_all/top_execout1[14]
     cb_parti693/top_execcmp42_wl_all_top_execout1[7]
        g_t1084/in_1
        cb_parti693/top_execm_mac_pp1src[7]
    top_execmul_16x16_8x8/top_execm_mac_pp1src[15]
     mac_pp1_reg/reg_in[15]
      reg_out_no_delay_reg[15]/d   <<<  unmapped_d_flop
      reg_out_no_delay_reg[15]/clk      setup

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
(clock gg_clk)                        capture                       810 R
-------------------------------------------------------------------------------

Exception    : 'path_adjusts/adjust_C2C' path adjust    -100ps

Cost Group   : 'C2C' (path_group 'C2C')

Start-point  : top_execexec_no_cmp_2/top_execmac/top_execm_mac_dual_8x8_n_reg_0/
reg_out_no_delay_reg[0]/clk

End-point    : top_execexec_no_cmp_2/top_execmac/mac_pp1_reg/reg_out_no_delay_reg[15]/d

The global mapper estimates a slack for this path of 97ps.
```

- When the `target` is positive, Genus can achieve a faster clock speed, which is the specified clock period minus the target number.
- When the `target` is negative, Genus 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, Genus 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, Genus 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, Genus 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 (`syn_map`) Genus 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 `tns_opto` attribute is turned on, Genus will work on either the Worst Negative slack or all the violating paths. You can track Genus's progress by looking at the *Worst Path* column in the log file.

As Genus works on the paths, the *Total Area* will be adjusted accordingly.

10

Clock Mapping Flow

- Overview on page 246
    - Clock Mapping Flow on page 246
    - Flow Example on page 247

Overview

Genus introduces a new way of mapping clock tree logic. Till now, the command `remap_to_dedicated_clock_library` was used to remap the clock tree logic to a subset of library cells but this could only be done on a post-map netlist, which is very late in the flow. The new approach lets you specify the set of clock library cells at the start and then all clock logic is mapped accordingly early in the flow.

This topic describes the new behavior and presents the related attributes to specify all input needed for the new flow.

Clock Mapping Flow

Clock tree logic must fulfill more strict requirements than datapath logic, e.g. multicorner stability. To achieve a certain level of robustness, only a limited set of library cells are often allowed to be used to implement clock tree logic, like multiplexers that switch between a functional and a test clock signal. With the new flow, you can now specify a set of clock library cells at start of the flow and then `syn_generic` and `syn_map` will map the clock logic as specified.

There are basically two steps needed to run the new clock mapping flow:

- Specifying clock tree library cells.

Four new attributes are introduced to specify valid clock tree library cells.

```
set_db cts_buffer_cells list of base_cells
set_db cts_inverter_cells list of base_cells
set_db cts_clock_gating_cells list of base_cells
set_db cts_logic_cells list of base_cells
```

A user can specify a list of `base_cells` for these attributes, also using a wildcard pattern like CLKBUF*. Typically, such clock cells will be marked `dont_use true`, since often they shall not be used for normal datapath logic. The clock mapping flow will ignore the `dont_use` setting, like the Clock Tree Synthesis (CTS) does.

- Enabling clock tree mapping.

The new clock mapping flow is enabled by setting the following attribute:

```
set_db map_clock_tree true
```

By default, the attribute is set to `false`, which means the feature is inactive. All clock mapping attributes must be specified before `syn_generic`, as mapping clock logic is one of the first steps inside the command. There is some post-processing done in the second half of `syn_map`, so the best place to check the clock logic for correct mapping is post `syn_map`. This can be done with reporting commands such as `report_clock_tree_structure`.

The tool treats all cells as clock logic that are returned by the command '`all_fanout -clock_tree`'. These are cells that propagate a clock phase from one of their input pins to an output pin.

The clocktree mapping identifies generic muxes that propagate clock signals and tries to map them to MUX cells from the set of specified clock cells. Instantiated library cells are not touched, if their `dont_touch` attribute is set (explicitly by the user or implicit via an SDC constraint applied to them). Mapped cells that are not marked `dont_touch` are being unmapped and remapped to the specified clock cells.

Flow Example

The following flow script shows an example of the new clock mapping flow:

```
read_mmmc ./scripts/mmmc_config.tcl
read_hdl ./rtl/top.v
elaborate top
init_design
<… more design setup …>
set_db cts_buffer_cells CLKBUF*
set_db cts_inverter_cells CLKINV*
set_db cts_clock_gating_cells CLKLAT*
set_db cts_logic_cells "CLKAND* CLKMUX* CLKXOR*"
set_db map_clock_tree true
syn_generic
syn_map
report_clock_tree_structure
<… continue flow …>
```

11

Retiming the Design

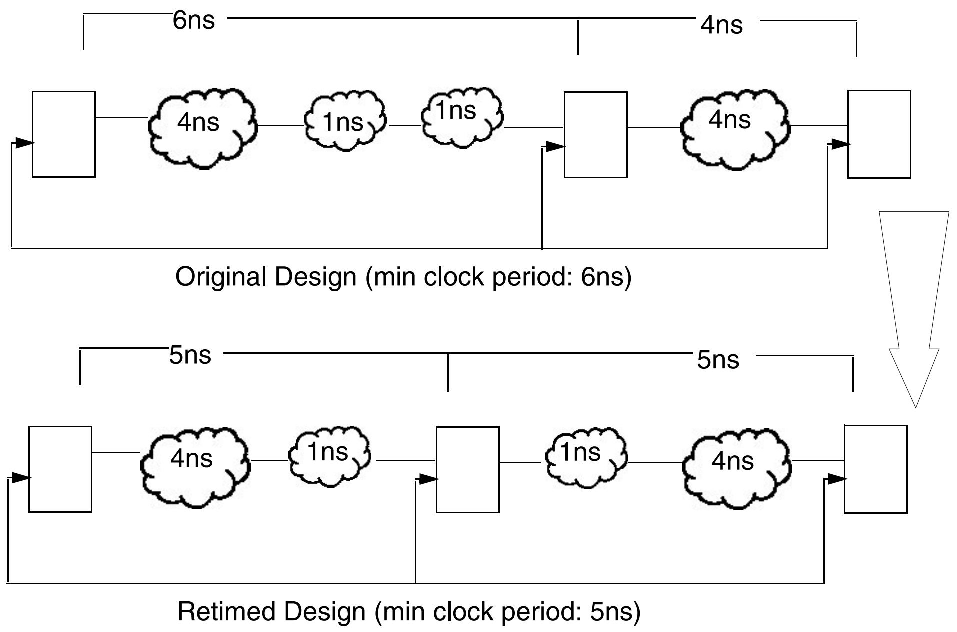
- Overview on page 250
  - Retiming for Timing on page 251
  - Retiming for Area on page 251
- Tasks on page 252
  - Retiming Using the Automatic Top-Down Retiming Flow on page 252
  - Manual Retiming (Block-Level Retiming) on page 255
  - Incorporating Design for Test (DFT) and Low Power Features on page 257
  - Localizing Retiming Optimizations to Particular modules on page 260
  - Controlling Retiming Optimization on page 261
  - Retiming Registers with Asynchronous Set and Reset Signals on page 262
  - Identifying Retimed Logic on page 266
  - Retiming Multiple Clock Designs on page 267

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 on page 250 shows how to use retiming to reduce the clock period from 6ns to 5ns.

Figure 11-1 Retiming for Minimum Delay


6ns
4ns
4ns
1ns
1ns
4ns
Original Design (min clock period: 6ns)
5ns
5ns
4ns
1ns
1ns
4ns
Retimed Design (min clock period: 5ns)

Genus supports both automatic and manual retiming.

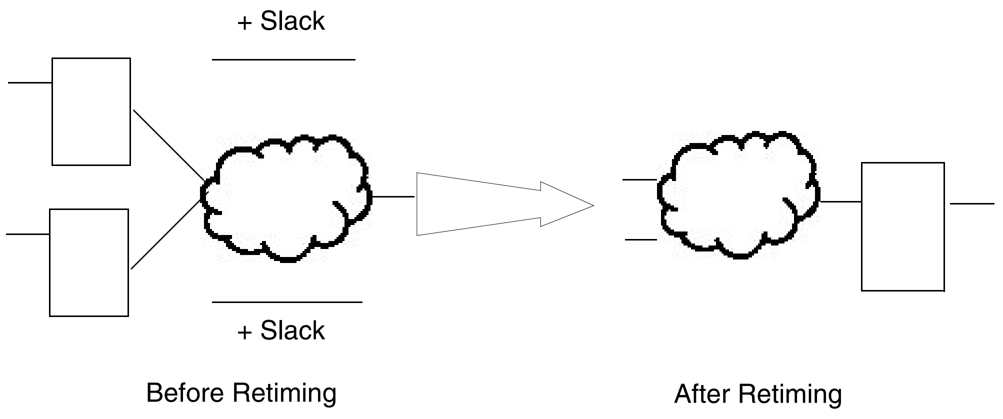
Retiming for Timing

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 a number of pipeline registers that satisfy the latency requirement. It can also be a sequential design that is not meeting the required timing. Genus 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, Genus 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 on page 251.

Figure 11-2 Retiming for Area



Tasks

- Retiming Using the Automatic Top-Down Retiming Flow
- Manual Retiming (Block-Level Retiming)
- Incorporating Design for Test (DFT) and Low Power Features
- Localizing Retiming Optimizations to Particular modules
- Retiming Multiple Clock Designs

Retiming Using the Automatic Top-Down Retiming Flow

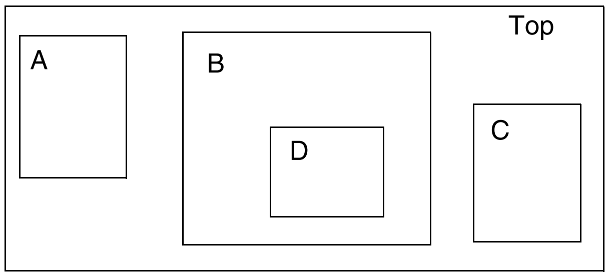
In the top-down (implicit) retiming flow, Genus retimes those blocks that were marked with the `retime` attribute. In this flow, retiming focuses on minimizing the delay. To retime the design to minimize area, you must use the manual retiming flow. See Manual Retiming (Block-Level Retiming) for more information about manual retiming.

If the `retime` attribute is set on a top-level design, all the modules will also be retimed. Use this flow when retiming is part of a well-planned synthesis strategy and the design has retimeable modules. Set the `retime` attribute to `true` on the desired modules after elaboration and then synthesize the entire design using `syn_generic`. Genus automatically derives appropriate constraints, synthesizes, and retimes the specified modules.

When synthesizing, you must synthesize to a technology mapped netlist. That is, after you used the `syn_generic` command, you must use the `syn_map` command.

Figure 11-3 on page 252 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, module `B` contains module `D` and some glue logic.

Figure 11-3 Graphic Illustration of a Hierarchical Design



Example 11-1 Top-Down Retiming on Submodules

The following flow illustrates how to retime only the `A` and `D` modules referred in Figure 11-3 on page 252:

1. Read the HDL for the entire design:

   ```
   genus:root: 1> read_hdl Top.v
   ```

2. Elaborate the top-level design:

   ```
   genus:root: 2> elaborate Top
   ```

3. Set the `retime` retiming attribute on the modules you want to retime. In this example, this would be `A` and `D`:

   ```
   genus:root: 3> set_db module:Top/A .retime true
   genus:root: 4> set_db module:Top/B/D .retime true
   ```

   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 Genus native format and optimization settings. In the following step, SDC is used:

   ```
   genus:root: 8> read_sdc top.sdc
   genus:root: 9> 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 to generic gates:

   ```
   genus:root: 11> syn_generic
   ```

   During this step, retiming is performed automatically on the blocks marked with the `retime` attribute and focuses on minimizing the delay.

6. Map the entire design to technology gates:

   ```
   genus:root: 16> syn_map
   ```

   During this step the design is optimized, including technology independent RTL optimization, advanced datapath synthesis, global focus mapping, and incremental optimization.

7. Evaluate the results using the following commands:

   ```
   genus:root: 23> report_timing
   genus:root: 24> report_gates
   genus:root: 25> report_area
   ```

If you wanted to retime `Top` and all its modules, not just *`A`* and *`D`*, merely set the `retime` attribute on `Top`:

```
genus:root: 27> set_db design:Top .retime true
```

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 `set_path_adjust` command before synthesis or synthesize the design automatically while deriving realistic constraints using the `-prepare` option of the `retime` command:

```
genus:root: 1> 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 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 module, you can use the `-prepare` and `-min_delay` options together:

```
genus:root: 10> retime -prepare -min_delay
```

Alternatively, you can issue `retime -prepare` before `retime -min_delay` sequentially:

```
genus:root: 11> retime -prepare
genus:root: 12> retime -min_delay
```

If you are specifying multiple modules, then issuing the commands separately will first map all modules and then retime them. If you specify the options together on multiple modules, then each module will be mapped and retimed before the next module 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:

```
genus:root: 15> retime -min_area
```

Note: 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 on page 256 does not use the `retime -prepare` command while Example 11-3 on page 257 does. Example 11-3 on page 257 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.
syn_generic
delete_obj massaged_clock_constraints //Remove any clock constraints that were
massaged
                                 //for retiming purposes
delete_obj /designs/block/timing/exceptions/path_adjusts/*
delete_obj /designs/block/timing/exceptions/multi_cycles/*
report_timing
retime -min_delay | -min_area
report_timing
report_gates
syn_map
..
```

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
syn_generic
syn_map
..
```

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 synthesizing the design to gates. Example 11-4 on page 258 illustrates this flow.

Note: For more information on DFT, see *Genus Design for Test Guide*

Example 11-4 Recommended Flow for Retiming with DFT and Low Power

```
set_db lp_insert_clock_gating true
read_hdl test.v
elaborate
...
set_db design:* .lp_clock_gating_max_flops 18
set_db design:* .lp_clock_gating_min_flops 6
set_db design:top_design .lp_clock_gating_test_signal test_signal
set_db design:top_design .max_leakage_power number
...
define_test_mode test_mode_signal
define_shift_enable shift_enable_signal
check_dft_rules
...
set_db [design | module] .retime true
syn_generic
syn_map
report_timing
report_clock_gating
report_scan_registers
...
connect_scan_chains -auto_create_chains
report_scan_chains
syn_opt
```

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.

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_db lp_insert_clock_gating true

read_hdl test.v
elaborate
set_db lp_clock_gating_max_flops 18
set_db lp_clock_gating_min_flops 6
set_db 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
syn_generic
syn_map                  //Synthesizes the netlist that has
                         //the scan flops and clock-gating logic
set_db unmap_scan_flops true

retime -min_delay
report_timing

replace_scan
connect_scan_chains -auto_create_chains
report_scan_chains
report_clock_gating
report_scan_setup

syn_opt
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_chains` 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.

Localizing Retiming Optimizations to Particular modules

Use the `retime_hard_region` attribute to contain the retiming operations to a specific module. By default, Genus operates on all retimeable logic through all levels of hierarchy. Therefore, if multiple modules on the same level of hierarchy are being retimed, their interfaces may get modified. Setting the `retime_hard_region` attribute on these modules 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* module from being moved across its boundary:

```
genus:root: 15> set_db [get_db modules *SUB_1] .retime_hard_region true
    Setting attribute of module '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:

```
genus:inst:retime_eg/U1 21> set_db a_reg1 .dont_retime true
genus:inst:retime_eg/U2 22> set_db B_reg .dont_retime true
```

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, Genus 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.

Figure 11-4 on page 262 through Figure 11-6 on page 264 below illustrate the *`my_flop`* register experience retiming as well as retiming with asynchronous reset optimization.

Figure 11-4 Register with Asynchronous Reset

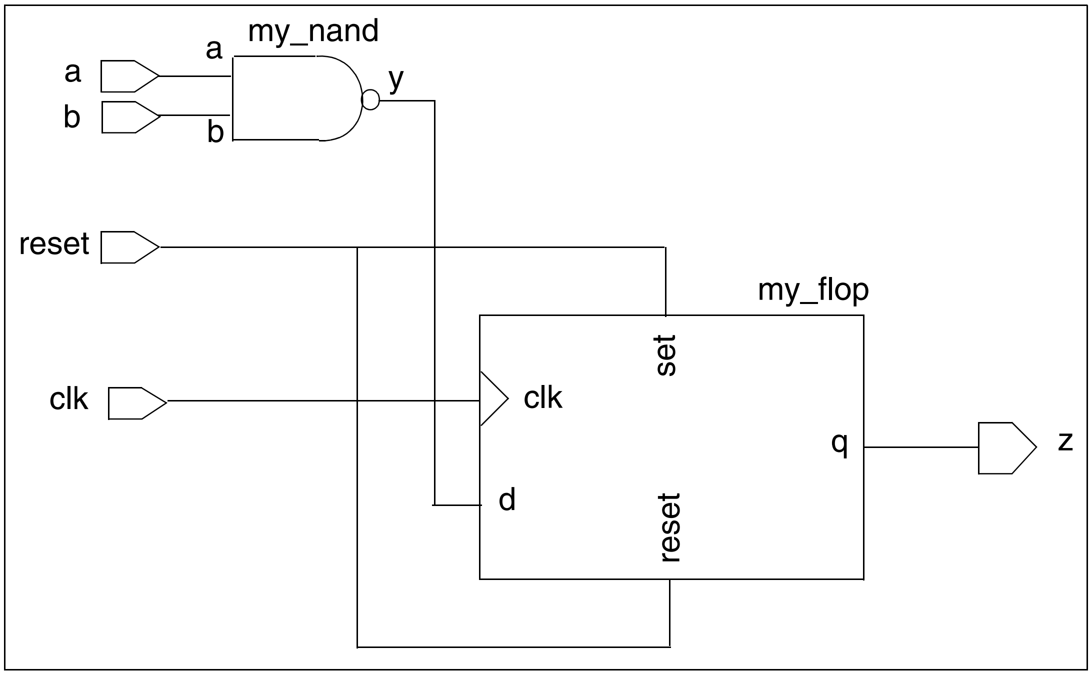


Figure 11-5 Register with Asynchronous Reset after Retiming

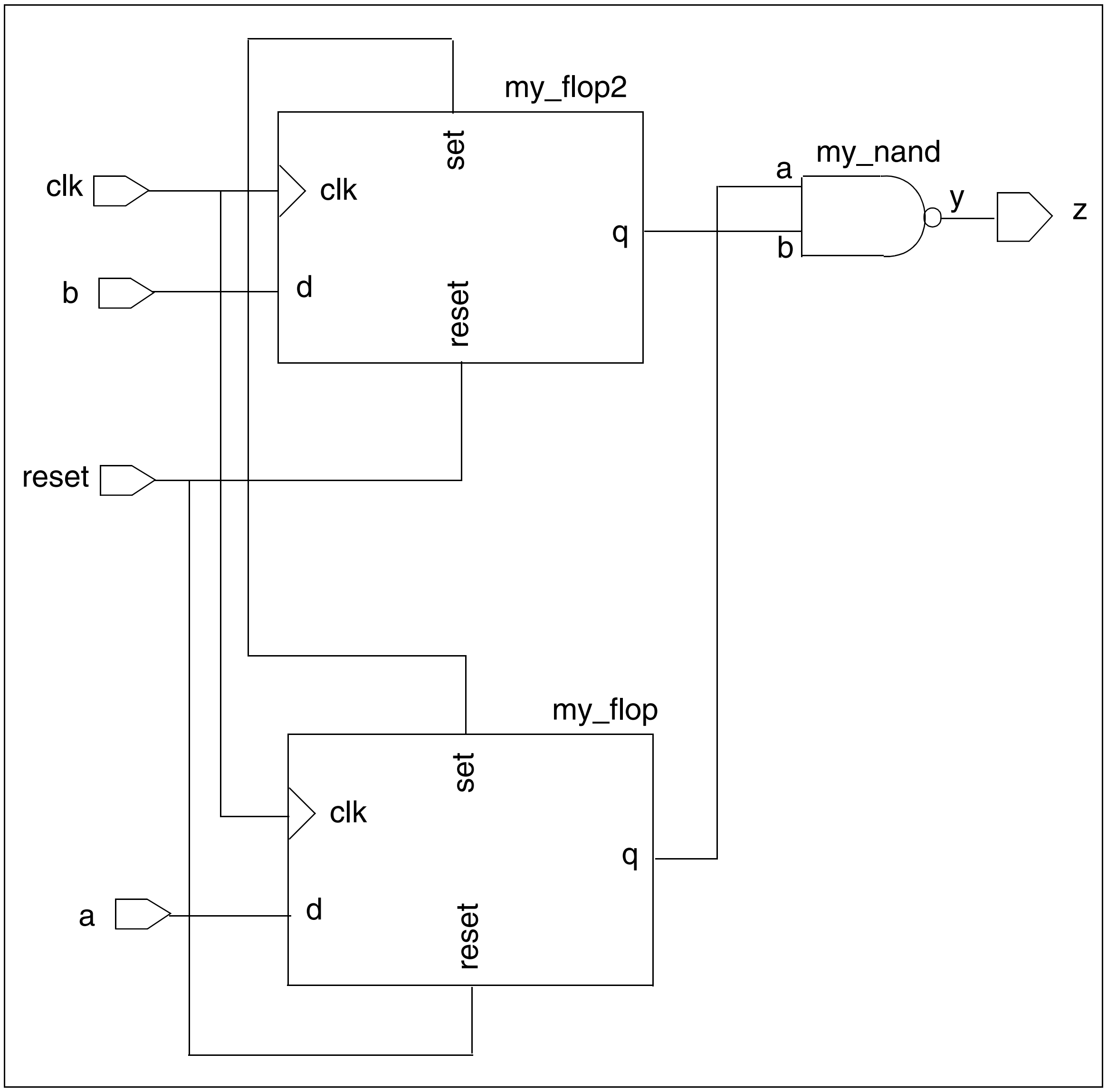
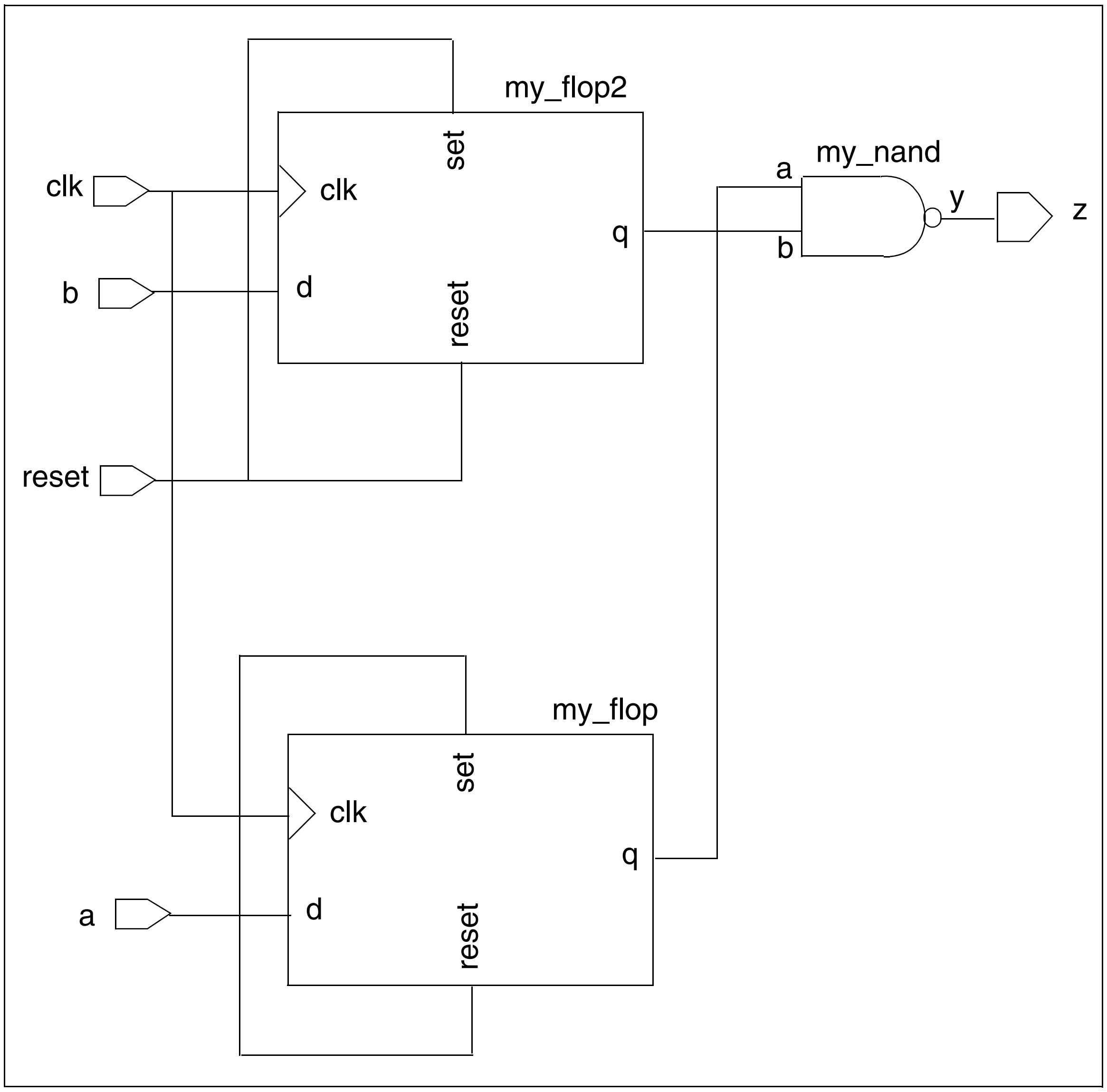


Figure 11-6 Register with Asynchronous Reset after Retiming and Optimization



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

```
...
genus:root: 41> set_db retime_async_reset true
genus:root: 42> set_db retime_optimize_reset true
genus:root: 43> retime -prepare
genus:root: 44> 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. This 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 Genus to add the `__retimed_reg` suffix to all registers that are moved during retiming optimization:

```
genus:root: 1> set_db 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));
```

Genus 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.

Example 11-7 on page 266 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 `test_reg[7]`.

Example 11-7 Retrieving the Original Name of a Retimed Register

```
genus:root: 1> set_db retime_reg_naming_suffix _stormy_reg
genus:root: 2> set_db [get_db insts *test_reg[7]] .trace_retime true
...
genus:root: 3> retime -prepare
genus:root: 4> retime -min_delay
genus:root: 5> get_db retime_1_stormy_reg .retime_original_registers
test_reg[7]
```

Retiming Multiple Clock Designs

Genus 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

```
genus:root: 1> read_hdl test2clk.v
genus:root: 2> elaborate

specify_multiclock_constraints

genus:root: 7> set_db [all::all_seqs -clock clk2] .dont_retime true
genus:root: 8> retime -prepare    //Optional
genus:root: 9> retime -min_delay
genus:root: 10> set_db [all::all_seqs -clock clk2] .dont_retime false
genus:root: 11> set_db [all::all_seqs -clock clk1] .dont_retime true
genus:root: 12> retime -prepare    //Optional
genus:root: 13> 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

Performing Functional Verification

- Overview on page 270
- Tasks on page 270
    - Writing Out dofiles for Formal Verification on page 270

Overview

Because synthesis involves complex optimizations and transformations, we strongly suggest that you perform functional verification after synthesis. Functional verification helps to ensure that the synthesized netlist is functionally equivalent to your original RTL design. You can perform one form of functional verification – equivalence checking – with Conformal. This chapter provides an overview on how to interface to Conformal Logical Equivalence Checker (Conformal in short) from Genus.

Tasks

Writing Out dofiles for Formal Verification

To interface with Conformal, Genus generates "dofiles" that should be loaded into Conformal. The following steps illustrate a high-level flow on creating dofiles.For more detailed explanations and examples, refer to *Genus Interface to Conformal.*

1. Check if the final netlist is functionally equivalent to the initial design read into Genus.

   To perform this check, use the Genus `write_do_lec` command to generate a dofile to interface with Conformal:

   ```
   genus:root: 1> write_do_lec -revised UNIX_path_to_the_netlist > Dofile
   ```

2. Check if the netlist conforms to low power rules defined in the Common Power Format (CPF) file.

   To perform this check, use the Genus `write_do_clp` generates a dofile to interface with Conformal Low Power. The usage is as follows:

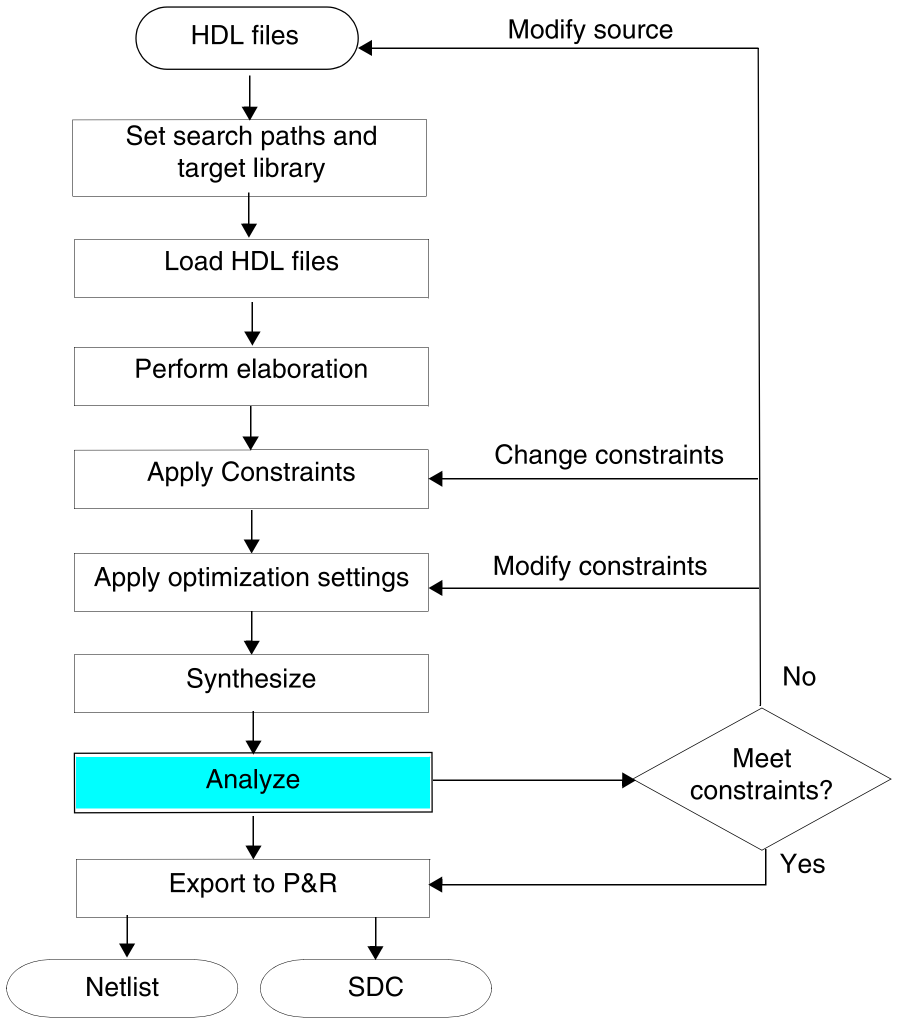
   ```
   genus:root: 12> write_do_clp -netlist UNIX_path_to_the_netlist > Dofile
   ```

13

Generating Reports

- Overview on page 272
- Tasks on page 273
    - Generating Timing Reports on page 273
    - Generating Area Reports on page 275
    - Tracking and Saving QoR Metrics on page 277
    - Summarizing Messages on page 285
    - Redirecting Reports on page 286
    - Customizing the report_* Commands on page 286

Overview



This chapter discusses how to analyze your synthesis results using the `report_*` commands and the log file.

Tasks

- Generating Timing Reports on page 273
- Generating Area Reports on page 275
- Tracking and Saving QoR Metrics on page 277
- Summarizing Messages on page 285
- Redirecting Reports on page 286
- Customizing the report_* Commands on page 286

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, `vcd` into the design directory and type the following command:

```
genus:design:top 15> 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.

```
genus:design:top 17> 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.

```
genus:design:top 21> report_timing -from [get_db insts *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.

```
genus:design:top 25> report_timing -exceptions <exception_name>
```

or

```
genus:design:top 25> report_timing -group <cost_group_name>
```

If timing is not met, “Timing Slack” is reported with a minus (-) number and “TIMING VIOLATION” is written out.

Genus generates the timing accuracy report down to the gate and net level.

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, Genus shows the technology library name, operating conditions, and the wire-load mode used to generate these reports.

➤ To generate a report that shows a profile of all library cells inferred during synthesis, type the following command:

```
genus:root: 34> report_gates
```

Genus generates a report listing all the gates, the number of instances in the design, and the total area for all these instances.

```
==========================================================
  Generated by:          Genus Synthesis Solution version
  Generated on:          date
  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 report, Genus 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.

➡ To generate an area report, type the following command:

```
genus:root: 37> report_area
```

Genus generates a report similar to the example below.

```
========================================================
  Generated by:          Genus Synthesis Solution version
  Generated on:          date
  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
```

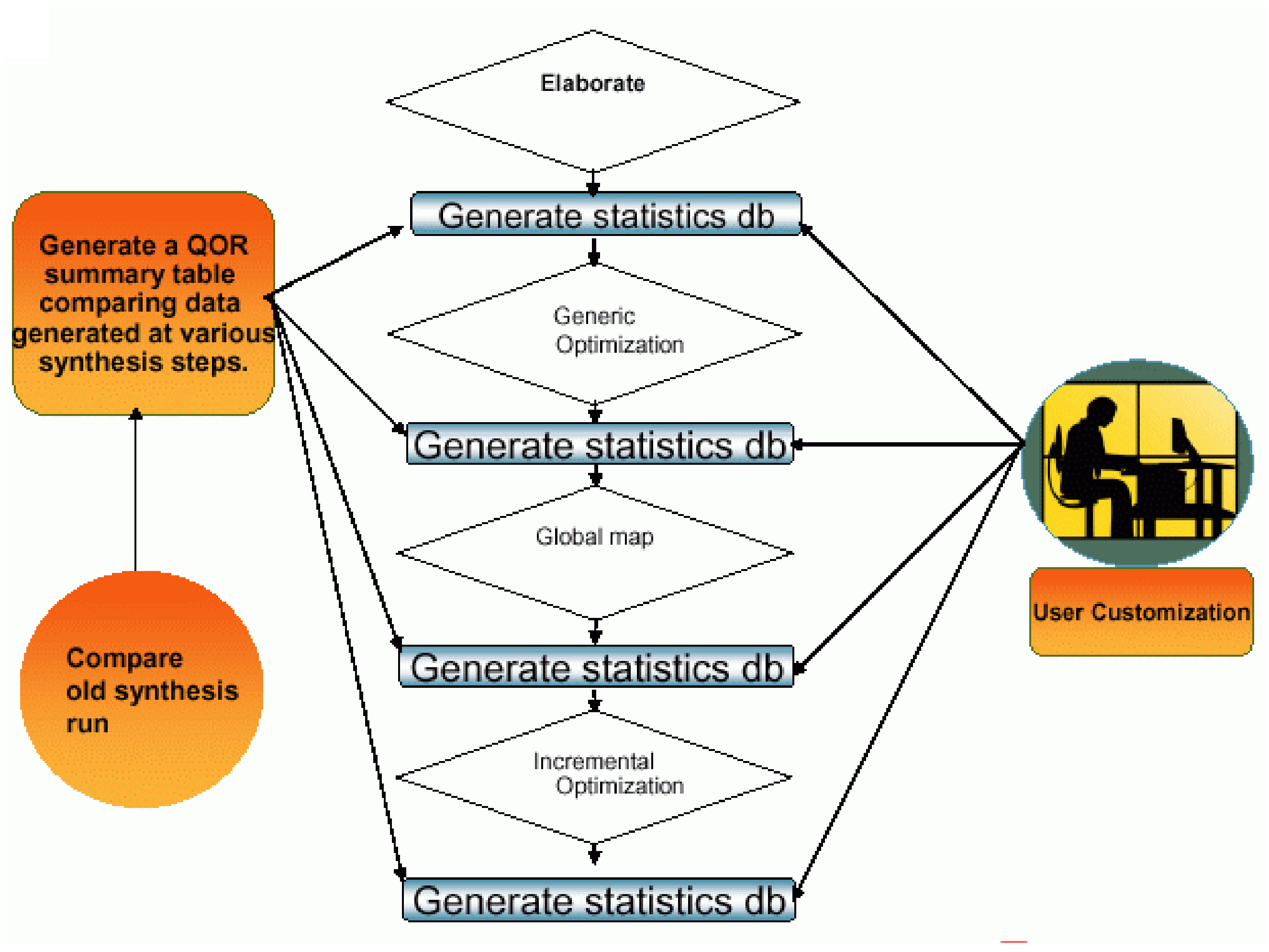
Tracking and Saving QoR Metrics

Querying individual reports and attributes to retrieve various metrics, debug QoR issues, or to compare data between multiple runs and stages can be very cumbersome.

Instead of running various reports to retrieve metrics for the design, you can track and save statistics information (QoR metrics) at various predefined and user-defined stages of the design and query predefined and user-defined metrics. Metrics data can be saved in and read from a statistics database. In addition you can compare the metrics of two runs. Figure 13-1 on page 277 shows the key features of this command.

Note: To use the `statistics` command, you can have only one design loaded in the tool.

Figure 13-1 Using the statistics database



Enabling Tracking and Generation of the QoR Metrics at Predefined Stages

➤ To enable *automatic* tracking and saving of the QoR metrics at the *predefined* stages during synthesis, set the following attribute to `true` before you elaborate the design:

```
set_db statistics_log_data true
```

By default, the metrics are not tracked or saved for the predefined stages.

Table 13-1 Predefined stages and corresponding commands

Predefined stage	Command
`elaborate`	`elaborate`
`generic`	`syn_generic`
`global map`	`syn_map`
`incremental`	`syn_opt`
`placed`	`syn_opt -physical`
`incrementally placed`	`syn_opt -physical -incremental`

Note: If you repeat any of the previously mentioned commands, the tool adds an increment to the stage name starting with 0 (for example, `incremental0`, `incremental1` and so on)

Tip

It is recommended to track and save the QoR metrics at the predefined stages to prevent loss of information in case your run would not finish successfully.

Enabling Tracking of Power Metrics

Because the computation of the power metrics is runtime-intensive, they are not tracked by default. To track the power metrics, you must enable the `statistics_enable_power_report` root attribute.

Adding Stages at Which the Metrics Must Be Computed

➤ To add a stage at which you want the tool to compute the metrics, use the `create_snapshot` command at the required stage.

```
create_snapshot {-name string | -label string}
[-auto {default|min}] [-exclude_time_metric]
```

```
    [-categories {all|{check|design|flow|hold|power|route|setup}...}]
    [pattern]
```

You need to specify a unique label name.

Note: The `create_snapshot` command is executed automatically at the predefined stages if you set the `statistics_log_data` attribute to `true`.

Example

To compute the metrics after you read in the SDC constraints, you can add a stage called `constraints`:

```
read_sdc my_constraints.sdc
create_snapshot -name constraints
```

Writing the Statistics Information to the Database

➤ To write out the metrics that were recorded at various stages, use the `save_snapshot` command:

```
save_snapshot
    [-to_file file]
```

If the file exists, the tool will overwrite the existing data. If you do not specify the file name, the name of the database file defaults to the setting of the `statistics_db_file` root attribute. It is recommended to set this attribute before you start tracking the metrics.

Note: The `save_snapshot` command is executed automatically at the predefined stages if you set the `statistics_log_data` attribute to `true`.

Identifying the Session for Which the Metrics are Computed

When you compare the metrics of different synthesis runs, the tool adds a run identification label to the stage name as a suffix.

➤ To define a user-defined identification label for the run, set the following root attribute:

```
set_db statistics_run_id string
```

The default is *`design.date_time_stamp`*

➤ To document the parameters of a session (run) for which you want to save the QoR metrics, set the following attribute:

```
set_db statistics_run_description string
```

By default, no run description is added.

Tip

These two attributes take affect the next time the `create_snapshot` command is executed. It is recommended to set these attributes at the beginning of the session before you start tracking metrics.

Example

In the following example, automatic tracking and generation of metrics at the predefined stages is enabled. The session starts with the default values for the `statistics_run_id` and `statistics_run_description`. As a consequence, the name of the database file is determined by the default setting of `statistics_run_id` attribute.

```
genus:root: 12> elaborate
  Elaborating top-level block 'cscan' from file 'qor_netlist.v'.
...
  Done elaborating 'cscan'.
Info    : Writing statistics database to file. [STAT-3]
        : Writing to db file 'cscan.Aug10-13:56:56.stats_db'
...
```

After elaborating the design, but before mapping to generic gates, the two attributes are set. When tracking the metrics at the next stage, they will be labeled with the new run ID and run description. The change of the `run_id` does not affect the name of the database.

```
genus:root: 13> set_db statistics_run_id "medium_effort"
  Setting attribute of root '/': 'statistics_run_id' = medium_effort
genus:root: 14> set_db statistics_run_description "run with medium effort mapping"
  Setting attribute of root '/
': 'statistics_run_description' = run with medium effort mapping
genus:root: 15> syn_generic
  Done unmapping 'cscan'
Info    : Writing statistics database to file. [STAT-3]
        : File 'cscan.Aug10-13:56:56.stats_db' exists. Overwriting db file
 'cscan.Aug10-13:56:56.stats_db'
  Synthesis succeeded.
```

Generating the Report for the Current Session

➤ To report the metrics at all predefined and user-defined stages, use the `report_metric` command:

```
report_metric -file file
[-format string ]
[-id string] [-names string] [pattern]
```

You must specify the run for which you want to report the metrics.

Example

```
genus:root: 21> report_metric -id test1

        QOR statistics summary
        ----------------------


Metric                  elaborate    generic      global_map   incremental       place
---------------------------------------------------------------------------------------
WNS.I2C                 n/a       9039.7        9738.0        9738.0        75.9
WNS.I2O                 n/a       8918.7        6533.6        6533.6        -1.0
WNS.C2C                 n/a       no_value      no_value      no_value      no_value
WNS.C2O                 n/a       9192.6        7852.9        7852.9        1144.4
WNS.default             n/a       no_value      no_value      no_value      no_value
```

```
TNS                      n/a       0            0            0            1
Violating_paths          n/a       0            0            0            1
runtime                  20        18.00        206.00       62.00        583.00
memory                   245.00    -44.00       98.00        -51.00       7.00
Leakage_power            n/a       15526.28     12271.55     12271.23     16359.53
Net_power                n/a       3055869.65   443139.21    443091.32    978543.89
Internal_power           n/a       9833530.08   1687395.23   1687332.94   1731198.96
Clock_gating_instances   n/a       0            766          766          766
total_net_length         n/a       n/a          n/a          n/a          3890913.25
average_net_length       n/a       n/a          n/a          n/a          248.83
routing_congestion       n/a       H0.00%,V0.00% H0.00%,V0.00% H0.00%,V0.00% H72.10%,V0.11%
utilization              0.0       0.0          65.97        65.97        82.67
Inverter_count           790       899          75           75           158
Buffer_count             0         0            0            0            1050
timing_model_count       0         0            766          766          766
sequential_count         6128      6128         6128         6128         6128
unresolved_count         0         0            0            0            0
logic_count              23320     2638         6126         6126         6126
Total_area               363201.20 176018.57    107566.41    107563.17    479235.83
Cell_area                335419.98 172496.31    92068.56     92065.32     102451.68
Net_area                 27781.22  3522.26      15497.85     15497.85     376784.15
```

Comparing Two Runs

If you wrote out the statistics database for several runs, you can load the data in the tool to compare the results of two runs at a time.

➤ To load a previously written statistics database, use the `read_metric` command.

➤ To compare two runs, use the `report_metric` command:

```
report_metric -file file
[-format string ]
[-id metric_id] [-names name] [pattern]
```

You must specify the id and the name of the stage for which you want the report.

Adding and Removing User-Defined Metrics

The tool has a number of predefined metrics, including metrics for timing, power, gate count, area, and more.

➤ To add your own metric, use the `define_metric` command.

```
define_metric -name metric [-type string]
[-tcl tcl_command_list]
[-inheritance_tcl string] [-no_inheritance]
[-inherit_no_children]
[-description string]
[-units string] [-precision num_dec_places]
[-target metric_name]
[-cmp function] [-threshold units] [-warning string]
```

➤ To remove a previously defined metric, use the `delete_metric` command.

```
delete_metric
    -name metric
```

Note: You can only remove user-defined metrics.

Measuring the Runtime

➤ To measure the elapsed runtime used to compute the statistics and write out the database file, use the `statistics_db_runtime` root attribute.

Sample Script

```
set_db statistics_log_data true
set_db statistics_run_id medium_effort
set_db statistics_run_description "global map with medium effort"
set_db statistics_db_file test1.stats_db

set DESIGN mydesign
...
suppress_messages { LBR-162 }
set_db library {tutorial.lib HighVt.lib}
set_db wireload_mode default
set_db lp_insert_clock_gating true
read_hdl qor_netlist.v
elaborate $DESIGN  //predefined stage
# define user metric
proc get_state {design} {
  return [get_db $design .state]
}
define_metric -name state -function get_state [find_unique_design]
read_sdc clk.sdc
create_snapshot -name constraints //user-defined stage
define_cost_group -name I2C -weight 1 -design $DESIGN
define_cost_group -name C2O -weight 1 -design $DESIGN
define_cost_group -name I2O -weight 1 -design $DESIGN
define_cost_group -name C2C -weight 1 -design $DESIGN
group_path -from [all::all_seqs] -to [all::all_outs] -group C2O -name C2O
group_path -from [all::all_inps] -to [all::all_seqs] -group I2C -name I2C
syn_generic //predefined stage
set_db syn_map_effort medium
syn_map   //predefined stage
syn_opt  //predefined stage
report_metric -id medium_effort
```

Summarizing Messages

Use the `report_messages` command to summarize all the info, warning, and error messages that were issued by Genus 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, Genus returns nothing. Consult the *Genus Command Reference* for more information on the `report_messages` command.

The following example is the first request to `report_messages` in a session:

```
genus:root: 12> 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), Genus would return nothing.

Redirecting Reports

The `report_*` commands send 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:

  ```
  genus:root: 26> report_gates > gates.rep
  ```

  or

  ```
  genus:root: 28> redirect gates.rep "report_gates"
  ```

- To append information into the existing *`gates.rep`* file, type the following command:

  ```
  genus:root: 34> redirect -append gates.rep "report_gates"
  ```

- To send the reports to `stdout` and to a file on the disk, type the following command:

  ```
  genus:root: 45> report_gates
  genus:root: 46> report_gates > gates.rep
  ```

  or

  ```
  genus:root: 54> redirect -tee gates.rep "report_gates"
  ```

Customizing the report_* Commands

The `etc/synth/rc/rpt` directory in your installation contains various `rpt*.tcl` files that 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.

14

Using the Genus Database

- Overview on page 288
- Tasks on page 289
    - Saving the Netlist and Setup on page 289
    - Restoring the Netlist and Setup on page 289
    - Splitting the Database on page 289

Overview

Genus supports a native binary database for design archival and restoration. You can save a snapshot of the design in the Genus memory at any point during the synthesis flow starting with elaboration. The database saves the design information (including netlist, timing, low power, DFT constraints, and physical information) and the setup. The database provides a more efficient and faster mechanism to save and restore a design compared to saving the netlist, `write_script` or `write_sdc` and design setup.

Note: User defined variables in the flow will not be saved in the database.

The setup consists of

- Non-default settings of root attributes
- Definitions of user-defined attributes
- Definitions of library domains
- Non-default attribute values of messages, libraries and their objects (library cells, pins, arcs).

The setup can be saved as part of the database or in a separate script.

Tip

Saving the setup to the database has the following advantages:

- ❑ Makes reading the setup less noisy: root attributes stored in the database are only set if they do not already have the same value. This prevents unnecessarily setting attributes like `library` and `lef_library` to the same value as setting those can take time and issue many messages.
- ❑ Can save attribute settings that cannot be saved by Tcl scripts.

The setup script will set all attributes regardless whether they have already the same value. They will also create library domains regardless if those domains already exist. Settings of root attributes which are not user-writable will be commented out in the script.

You can later restore the design and the setup without any loss of information.

Tasks

Saving the Netlist and Setup

➡ To save the netlist and optionally the setup, use the write_db command:

```
write_db -to_file db_file
    [-all_root_attributes | -no_root_attributes]
    [-script file] [design] [-quiet] [-verbose]
```

By default, the setup is saved in the database. To save the setup in a separate script, use the `-script` option. Saving the setup to a script can be useful to review or modify the setup. To prevent saving of the setup, specify the `-no_root_attributes` option.

Restoring the Netlist and Setup

➡ If the database contains the netlist and the setup information, use the read_db command:

```
read_db [db_file ] [-from_tcl string] [-quiet] [-verbose]
    [-mmmc_file file]
```

➡ If the setup was written to a separate script, follow these steps to restore the information:

```
source script_file
```

```
read_db db_file [-quiet] [-verbose]
```

Splitting the Database

➡ To remove the setup information from the database and write it to a setup script, use the split_db command.

```
split_db [input_db_file] [-from_tcl file]
          -script file [-to_file file]
```

Tip

Writing the setup information to a Tcl script can be useful when the setup needs to be reviewed or modified.

15

Interfacing to Place and Route

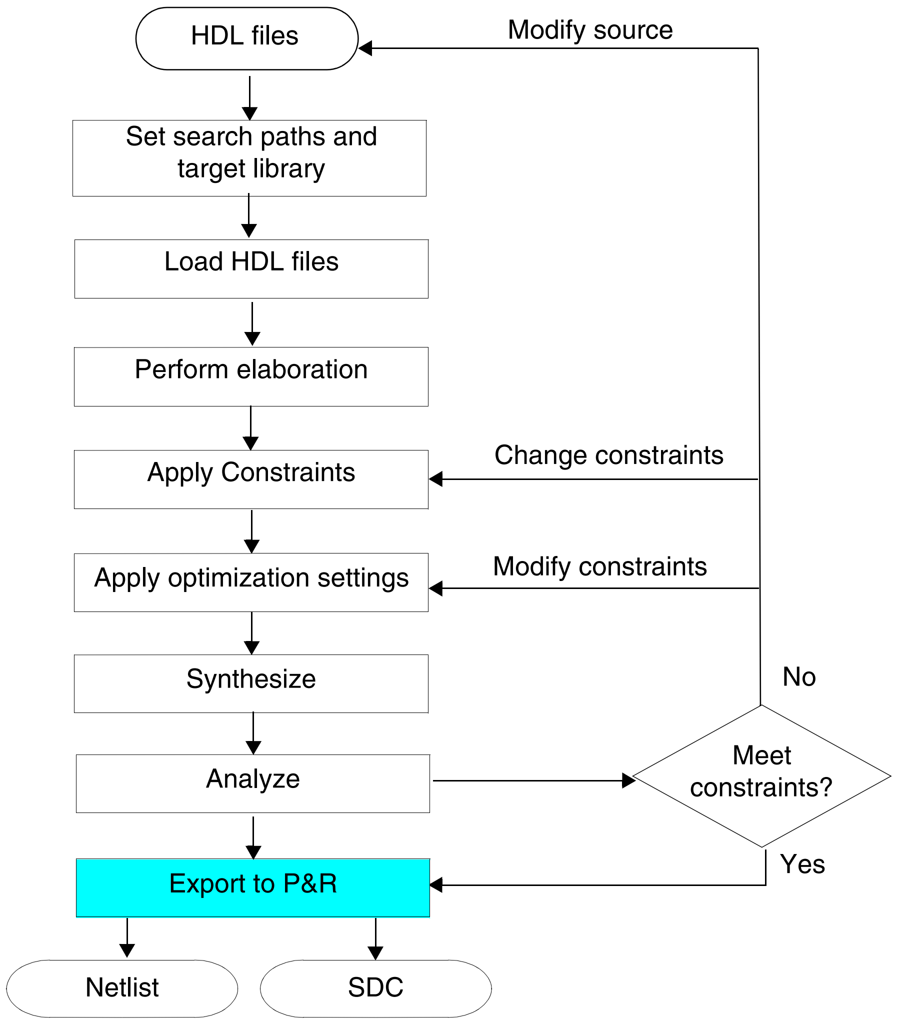
- Overview on page 292
- **Tasks**
  - Preparing the Netlist for Place-and-Route or Third-Party Tools on page 293
    - Changing Names on page 293
    - Naming Flops on page 294
    - Removing Assign Statements on page 295
    - Inserting Tie Cells on page 297
    - Handling Bit Blasted Port Styles on page 298
    - Handling Bit-Blasted Constants on page 299
  - Generating Design and Session Information on page 300
    - Saving and Restoring a Session in Genus on page 301
  - Writing Out the Design Netlist on page 302
  - Writing SDC Constraints on page 305
  - Writing an SDF File on page 306

Overview

After you have completed synthesis of your current design, you can write out files for processing by your place and route tools.

Figure 15-1 on page 292 shows where you are in the top-down synthesis flow.

Figure 15-1 Top-Down Synthesis Flow



This chapter describes how to write out the synthesized design so that the netlist and constraints can interface smoothly with third-party tools.

Preparing the Netlist for Place-and-Route or Third-Party Tools

When interfacing with other tools (such as place-and-route tools), you may need to make modifications to the gate-level netlist.

Changing Names

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 `update_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. All changes are global unless you specify the `-local` option, in which case only the current directory is affected.

- To rename all `module` objects with the `top_` prefix in the output netlist, use the following command:

  ```
  genus:root: 12> update_names -prefix top_ -module
  ```

- To add the suffix `_m` on all the `design` and `module` objects, use the following command and options:

  ```
  genus:root: 15> update_names -design -module -suffix _m
  ```

- The following example will change all instances of lowercase `n` with uppercase `N` and underscores (_) with hyphens (-).

  ```
  genus:root: 18> update_names -map {{"n" "N"} {"_" "-"}}
  ```

- In the following example, all instances of `@` will be replaced with `at`. If the `replace_str` option is not specified, the default character of underscore (_) will be used.

  ```
  genus:root: 19> update_names -restricted "@" -convert_string "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.

  ```
  genus:root: 22> update_names -nocase
  ```

- 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:

```
genus:design:test.ports 12> vls
./       SI2      clk1     in1[0]   in2[0]   in2[3]   in3[2]   in3[5]
genus:design:test.ports 13> change_names -port_bus -map {{"[" "("} {"]" ")"}}
genus:design:test.ports 14> vls
./       SI2      clk1     in1[0]   in2[0]   in2[3]   in3[2]   in3[5]
```

Naming Flops

You may need to change the naming style of the flops to match third-party requirements on the netlist.

Genus uses the following default flop naming styles:

- For vectored variables, such as `reg[2:0] cout`, the style string is `%s_reg%s`. The reg names produced are `cout_reg2`, `cout_reg1`, `cout_reg0`.
- For scalar variables, such as `reg cout`, the style string is `%s_reg`. The reg name produced is `cout_reg`.

➤ To customize the default naming scheme, use the `hdl_reg_naming_style` attribute:

The default setting is:

```
set_db hdl_reg_naming_style %s_reg%s
```

The first `%s` is the variable name. If the variable is a vector, the second `%s` is the individual bit of the vector as specified by the `hdl_array_naming_style` attribute.

Synopsys Design Compiler Compatibility Settings

To match Design Compiler nomenclature, specify the following attribute before you elaborate the design:

```
set_db hdl_array_naming_style %s_%d
```

Two-dimensional arrays will then be represented in the following format in the Genus output netlist: `<var_name>_reg_<idx1>_<idx2>` . For example, `cout_reg_1_1`

Removing Assign Statements

Some place and route tools cannot recognize `assign` statements. For example, the generated gate-level netlist could contain `assign` statements like:

```
...
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[15] = 1'b0;
DFFRHQX4 cout_reg_0(.D (n_15), .CK (clock), .RN (n_13), .Q (cout[0]));
...
```

Note: Innovus can handle Verilog assign statements natively and may not need assigns removal in Genus flow. If assign removal is needed, the following section explains the use model.

Replacing Assignments during Incremental Optimization

➤ To replace assign statements with buffer or inverter instantiations, set the `remove_assigns` root attribute to `true` before incremental optimization.

➤ To control the aspects of the replacement of `assign` statements in the design with buffers or inverters, you can use the `add_assign_buffer_options` command.

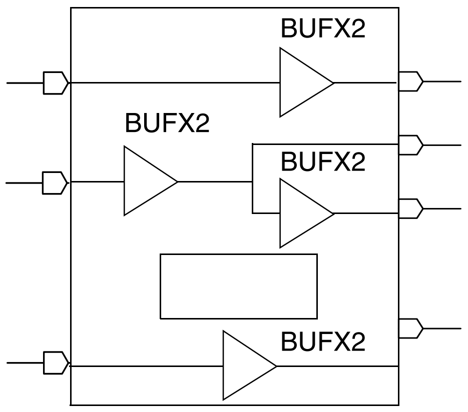
To specify the module in which to replace the `assign` statements, use the `-design` option. The following command specifies to only remove `assign` statements from the `sub` module:

```
genus:root: 3> add_assign_buffer_options -design [get_db modules *sub]
```

To specify a particular buffer to use to replace the `assign` statements, use the `-buffer_or_inverter` option of this command. The following usage specifies to replace the `assign` statements with the *BUFX2* cell. Figure 15-2 on page 296 shows the result of the optimization.

```
genus:root: 4> add_assign_buffer_options -buffer_or_inverter \
    lib_cell:default_library/slow/BUFX2
```

Figure 15-2 Assign Statements Replaced with Buffers



Inserting Tie Cells

Inserting Tie Cells during Incremental Optimization

➡ To allow that a constant assignment can be replaced with a tie cell during incremental optimization, set the `use_tiehilo_for_const` root attribute to `true`.

The tool will select a usable tie cell.

To allow the use of a tie cell with an inverter if either the tie high or tie low cell is not found, set the `iopt_allow_tiecell_with_inversion` root attribute to `true`.

To ignore all preserve settings when inserting tie-cells during synthesis, set the `ignore_preserve_in_tiecell_insertion` root attribute to `true`.

If you want finer control over the tie cell insertion, you can replace the constant assignments after incremental synthesis.

Inserting Tie Cells after Incremental Optimization

➡ To insert tie cells after incremental synthesis, use the `add_tieoffs` command.

```
add_tieoffs
[-high_low lib_cell] | -high lib_cell -low lib_cell]
[ -always_on_high_low lib_cell
| -always_on_high lib_cell -always_on_low lib_cell]
[-allow_inversion] [-max_fanout integer]
[-all] [-skip_unused_hier_pins] [-place_cells]
[-verbose] [module | design]
```

The options of the `add_tieoffs` command allow you to control the aspects of the tie cell insertion.

You can select the tie cell to be used to tie the constants 0s (1s) by specifying the `-low` (`-high`) option.

You can select the tie cell to be used to tie the constants 0s and 1s by specifying the `-high_low` option.

You can also allow the use of a tie cell with an inverter if either the tie high or tie low cell is not found by specifying the `-allow_inversion` option.

By default this command skips scan pins, preserved pins, preserved nets, and modules. You can specify to connect to scan pins by specifying the `-all` option.

You can specify to insert tie cells in the entire design or in the specified module. If you omit the design name, the top-level design of the current directory of the design hierarchy is used.

Handling Bit Blasted Port Styles

Some place and route tools prefer to see port names in expanded format, rather than as vector representations, which is how Genus generates the gate-level netlist:

```
module addinc(A, B, Carry, Z);
    input [7:0] A, B;
...
```

Bit blasting is the process of individualizing multi-bit ports through nomenclature. For example, Verilog port `A[0:3]` has four bits.

Bit blasting port `A` can produce the following result in the netlist:

```
A_0
A_1
A_2
A_3
```

1. To control the bit blasted port naming style, set the `bit_blasted_port_style` attribute.
2. To bit blast all ports of the specified design use the `bitblast_all_ports` command.

Example

```
    set_db bit_blasted_port_style %s\[%d\]
    bitblast_all_ports
```

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] ;
...
```

If you used the default setting of the `bit_blasted_port_style` attribute, the netlist would look like:

```
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;
....
```

Handling Bit-Blasted Constants

Some place and route tools cannot properly handle bus constants in the netlist.

➤ To bit blast all constants in the design, set the `write_vlog_bit_blast_constants` root attribute to `true`.

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}`.

Generating Design and Session Information

➤ To generate all files needed to be loaded in an Innovus session, use the following command:

```
write_design -innovus -base_name mydesign
```

For example, if you specified `session1/top` as the base name, Genus will generate the following files in you working directory under subdirectory `session1`:

- `top.genus_init.tcl`
- `top.invs_init.tcl`
- `top.mmmc.tcl`
- `top.g`
- `top.def (if input DEF is read)`
- `top.genus_setup.tcl`
- `top.v`
- `top.invs_setup.tcl`
- `top.mode`
- `top.sdc`

To start an Innovus session, you only need to source the `top.invs_setup.tcl` file which will in turn load the necessary files, such as the libraries, the generated netlist file, the SDC constraints written out by Genus, and a mode file.

Saving and Restoring a Session in Genus

The `write_design` command also writes out the necessary files and information to restore a Genus session.

```
genus:root:.designs> write_design -base_name mydesign
Exporting design data for 'fifo' to ./mydesign...
Info    : Generating design database. [PHYS-90]
        : Writing netlist: ./mydesign.v
Info    : Generating design database. [PHYS-90]
        : Writing Metrics file: ./mydesign.metrics.json
Info    : Generating design database. [PHYS-90]
        : Writing write_script: ./mydesign.g
Info    : Generating design database. [PHYS-90]
        : Writing floorplan: ./mydesign.def
Info    : Generating design database. [PHYS-90]
        : Writing congestion map: ./mydesign.cmap.gz
Info    : Generating design database. [PHYS-90]
        : Writing multi-mode multi-corner file: ./mydesign.mmmc.tcl
Finished DEF export (command execution time mm:ss (real) = 00:00).
Finished SDC export (command execution time mm:ss (real) = 00:00).
Info: file .//mydesign.cstr_mode_a.sdc has been written
Info    : Design has no library or power domains. [INVS_MSV-301]
        : No power domains will be created for Innovus.
Finished SDC export (command execution time mm:ss (real) = 00:00).
Info: file .//mydesign.cstr_mode_b.sdc has been written
File .//mydesign.mmmc.tcl has been written.
Info    : Generating design database. [PHYS-90]
        : Writing INIT setup file for Genus: ./mydesign.genus_init.tcl
Info    : Generating design database. [PHYS-90]
       : Writing Genus(TM) Synthesis Solution setup file: ./mydesign.genus_setup.tcl
** To load the database source ./mydesign.genus_setup.tcl in a Genus(TM) Synthesis
Solution session.
Finished exporting design data for 'fifo' (command execution time mm:ss cpu = 00:00,
real = 00:23).
```

To restore the Genus session:

1. Invoke Genus
2. `source ./mydesign.genus_setup.tcl`

Writing Out the Design Netlist

The final part of the Genus 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 Genus:

- Mapped gate-level netlist
- Genus 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 `syn_map` 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:

```
    write_hdl > design.v
```

Note: If you issue the `write_hdl` command before issuing the `syn_map` 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`:

```
    write_hdl  /designs/top/ > top.v
```

If you wanted to write out a specific module, without its parent or child design, use the `write_hdl` command with the unresolved attribute:

```
set_db [get_db [get_db modules *bottom] .hinsts] .unresolved true
write_hdl [ get_db modules *middle ] > middle.v
```

In this example, even though the `middle` design instantiates the `bottom` module, 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 module and any child designs it instantiates, specify the top-level design with the `write_hdl` command.

```
write_hdl module:top/middle > middle_and_bottom.v
```

In this example, the `middle` and its module, `bottom`, were written out to `middle_and_bottom.v`.

➤ To write out each Verilog primitive in the netlist as an assign statement with a simple Verilog logic expression, type the following command:

```
write_hdl -equation
```

For example, the RTL code, shown in Example 15-1:

Example 15-1 RTL Code

```
module test (y, a, b);
    input [3:0] a, b, c;
    output [3:0] y;
    assign y = (a + b) | c;
endmodule
```

Using the following commands, without the `write_hdl -equation` command:

```
set_db library tutorial.lib
read_hdl test.v
elaborate
write_hdl
```

Creates the post-elaboration generic netlist, shown in Example 15-2.

Example 15-2 Post-Elaboration Generic Netlist Without the write_hdl -equation Command

```
module test (y, a, b);
    input [3:0] a, b, c;
    output [3:0] y;
    wire n_6, n_7, n_8, n_9;
    and g1 (n_6, a[0], b[0]);
    and g3 (n_7, a[1], b[1]);
    and g4 (n_8, a[2], b[2]);
    and g5 (n_9, a[3], b[3]);
    or g6 (y[0], n_6, c[0]);
    or g2 (y[1], n_7, c[1]);
    or g7 (y[2], n_8, c[2]);
    or g8 (y[3], n_9, c[3]);
endmodule
```

If you use the same sequence of commands with the addition of the `write_hdl -equation` command, then Example 15-3 shows the post-elaboration generic netlist:

Example 15-3 Post-Elaboration Generic Netlist With the write_hdl -equation Command

```
module test (y, a, b);
    input [3:0] a, b, c;
    output [3:0] y;
    wire n_6, n_7, n_8, n_9;
    assign n_6 = a[0] & b[0]);
    assign n_7 = a[1] & b[1]);
    assign n_8 = a[2] & b[2]);
    assign n_9 = a[3] & b[3]);
    assign y[0] = n_6 & c[0]);
    assign y[1] = n_7 & c[1]);
    assign y[2] = n_8 & c[2]);
    assign y[3] = n_9 & c[3]);
endmodule
```

For debugging and analysis purposes, it is sometimes useful to generate a gate-level representation of a design without using technology-specific cells. In such cases, use the `-generic` option. You can write out a generic netlist either after issuing the `syn_generic` command or after `syn_map`.

➤ To create a gate-level netlist that is not technology specific, type the following command:

```
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:

```
write_sdc
```

Like the other Genus 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:

```
write_sdc > constraints.sdc
```

Note: Genus writes out the SDC constraints in SDC format.

Writing an SDF File

➤ To write out a Standard Delay Format (SDF) file, use the write_sdf command immediately after synthesis.

For example, to write out the SDF file into the `ksable.sdf` file, enter the following command:

```
write_sdf > ksable.sdf
```

Analysis and verification or timing simulation tools can use SDF files for delay annotation. The SDF file itself contains constructs that specify the delay of all the cells and interconnects in the design in the Standard Delay Format. Specifically, it includes the delay values for all the timing arcs of a given cell in the design.

Example 15-4 shows the header and combinational cell description in an SDF file.

Example 15-4 SDF File

```
(DELAYFILE
    (SDFVERSION  "OVI 3.0")
    (DESIGN      "ksable")
    (DATE        "Day Mon Date Time Time_Zone Year")
    (VENDOR      "Cadence, Inc.")
    (PROGRAM     "Genus Synthesis Solution")
    (VERSION     "7.1")
    (DIVIDER     .)
    (VOLTAGE     ::1.08)
    (PROCESS     "::1.0")
    (TEMPERATURE ::125.0)
    (TIMESCALE   1ps)
    (CELL
        (CELLTYPE "ADDFX1HS")
        (INSTANCE g44)
        (DELAY
            (ABSOLUTE
                (PORT A (::0.0))
                (PORT B (::0.0))
                (PORT CI (::0.0))
                (IOPATH (posedge B) S (::306) (::291))
                (IOPATH (negedge B) S (::306) (::291))
                (COND B == 1’b0 && CI == 1’b0 (IOPATH (posedge A) S (::139) ()))
                (COND B == 1’b0 && CI == 1’b0 (IOPATH (negedge A) S () (::224)))
                (IOPATH (posedge CI) S (::312) (::322))
                (IOPATH (negedge CI) S (::296) (::306))
                (COND A == 1’b0 && CI == 1’b1 (IOPATH (posedge B) S () (::291)))
                (COND A == 1’b0 && CI == 1’b1 (IOPATH (negedge B) S (::297) ()))
                (COND A == 1’b1 && CI == 1’b0 (IOPATH (posedge B) S () (::268)))
                (COND A == 1’b1 && CI == 1’b0 (IOPATH (negedge B) S (::306) ()))
                (COND B == 1’b1 && CI == 1’b1 (IOPATH (posedge A) S (::138) ()))
                (COND B == 1’b1 && CI == 1’b1 (IOPATH (negedge A) S () (::231)))
            )
        )
    )
)
```

16

Modifying the Netlist

- Overview on page 308
- **Tasks**
  - Connecting Pins, Ports, and hports on page 309
  - Disconnecting Pins, Ports, and hports on page 309
  - Creating New Instances on page 310
  - Overriding Preserved Modules on page 311
  - Creating Unique Parameter Names on page 312
  - Naming Generated Components on page 313
  - Changing the Instance Library Cell on page 313

Overview

This chapter describes how to modify the netlist.

Note: Netlist modifications for the purpose of meeting third-party requirements on the netlist are described in Chapter 15, “Interfacing to Place and Route.”

Connecting Pins, Ports, and hports

The `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 hports that are in different levels of hierarchy. This is illegal Verilog.
- Pins, ports, or hports that are already connected
- An object to itself.
- An 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 hports

The `disconnect` command disconnects a single hport, 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 `create_inst` command creates an instance type in a specified level of the design hierarchy. You can instantiate inside a top-level design or a module. There is a command option `name` for the `create_inst` command.

- You cannot instantiate objects that require a change to a preserved module.
- You cannot create a hierarchical loop. If module `A` contains module `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` and `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/hinsts/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 modules and the top design
foreach module [get_db modules *] {
  # 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 module
    set inst_dir [lindex [get_db $module .instances ] 0]
    set const_dir $inst_dir/constants
  }
  # Work on both logic constants
  foreach const {0 1} lib_pin {TIELO/Y TIEHI/Y} {
    # Find the logic 0 or logic 1 pin within this module
    set const_pin $const_dir/$const
    # find the lib_cell that we want to instantiate
    set lib_cell [get_db lib_cells [dirname $lib_pin]]

    # Find all the loads driven by this logic constant pin
    set net [get_db $const_pin .net ]
    if {[llength $net]} {
      foreach load [get_db $net .load ] {
        # At each load instantiate a tie_inst
        set tie_insts \
          [edit_netlist new_instance -name "tie_${const}_cell" \
               $lib_cell $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 $lib_pin]
        # 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_db $load .net ] "logic_${const}_net"
      }
    }
  }
}
```

Creating Unique Parameter Names

Use the `hdl_parameter_naming_style` attribute to define the naming style for each binding (*`parameter`*, *`value`*).

■ To specify naming style, type the following command:

```
set_db hdl_parameter_naming_style "_%d"
```

Ensure that you specify the attribute on the root-level ("`/`").

Table 16-1 illustrates the naming style results of various `hdl_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 16-1 Specifying Naming Styles

Naming Style Setting	Resulting Naming Style
`set_db hdl_parameter_naming_style "_%d"`	`foo_1_2`
`set_db hdl_parameter_naming_style "_%s_%d"`	`foo_p_1_q_2`
`set_db hdl_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:

```
set_db hdl_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 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 Genus internally generated names.

For example, if you were to set the attribute to `CDN_DP_` by typing:

```
    set_db 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:

```
    foreach i [get_db design:design modules CDN_DP_*] \
     {ungroup [get_db $i .instances]}
```

Changing the Instance Library Cell

After Genus 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 *lib_cell* 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:

```
get_db inst:top_counter/I2/cout_reg_5 .lib_cell
```

Genus will show the library cell and its source library:

```
lib_cell:slow/DFFRHQX4
```

To manually force the instance to have a different library cell, you can use the same *lib_cell* 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:

```
set_db inst:top_counter/I2/cout_reg_5 \
    .lib_cell [get_db lib_cells /lib_cell_path/DFFRHQX2]
```

This command will force the instance to be mapped to *DFFRHQX2*.

Note: Make sure to generate all the reports, especially the timing report, to ensure that no violations exist in the design.

17

IP Protection

- Overview on page 316
  - Decryption and Encryption using xmprotect on page 316
- Supported Encryption Flows on page 317
  - Variation due to encryption pragma on page 317
  - Variation due to type of encryption key on page 319
- Levels of Protection on page 321
- Round-trip Protection Flow on page 322
- Details and Examples of Protection Features on page 323
  - Encrypting Designs within Genus on page 323
  - Encrypting Designs outside Genus on page 324
  - Loading Encrypted Designs on page 325
  - Writing Encrypted Designs on page 326
  - Attributes — "protected" and "encrypted" on page 326

Overview

Synthesis users sometimes need their designs to include some HDL files that are IPs (Verilog or VHDL blocks owned by specific designers or teams) and these IPs have restrictions on who can see the HDL definitions of these blocks. To prevent unrestricted access or theft of the IPs that are part of the input RTL design, users may want to keep such files (or parts of files) in an encrypted form. For usage of such IPs in EDA design flow, EDA tools should be able to accept these encrypted design inputs and generate an encrypted output for those encrypted parts of the design. EDA tools, which are able to read an encrypted design input, also need to ensure that there is no method by which a user can get a decrypted form of the original design input.

Genus supports reading and synthesizing design inputs (Verilog, VHDL, tcl) which were encrypted using `xmprotect` (Cadence® Xcelium Simulator). Genus also supports reading and synthesizing of designs which were encrypted using non-cadence tools, provided they have been encrypted using the P1735 IEEE standard. Genus provides a limited support for IP protection during synthesis flow as discussed in this chapter.

Decryption and Encryption using xmprotect

To read a Verilog or VHDL file, that is partially or fully encrypted, Genus needs the ability to decrypt encrypted parts of the HDL file. Similarly, in some cases, users want that after synthesis, the parts of the netlist that are derived from an encrypted part of the synthesis input, should be kept encrypted while writing out the Verilog for the synthesized netlist. Genus uses Xcelium functions (`xmprotect` utility of the Cadence® Xcelium Simulator) to decrypt the encrypted design input, and to re-encrypt these parts of the design that are derived from the encrypted input. All this is done in the internal memory.

Note: Genus does not provide any command or option to write the decrypted contents of the encrypted file.

A particular release of Genus is tied to a particular release of Xcelium. To find out which release of the `xmprotect` utility is supported by Genus, query the `xm_protect_version` root attribute. Refer to Encrypting Designs outside Genus on page 324 for more detailed examples for `xmprotect` usage for encrypting design files.

Commands to Support Encryption

For convenience, Genus provides an `encrypt` command that is based on `xmprotect`. It allows to encrypt Verilog, VHDL and Tcl files. Refer to Encrypting Designs within Genus on page 323 for more detailed examples.

Supported Encryption Flows

In general, `read_hdl` command supports reading the encrypted Verilog and VHDL files without requiring any additional option or attribute setting. These files can be partially or fully encrypted. However, variations in methods of encrypting HDL files can lead to additional steps for reading the encrypted design. Some of the encryption variants that Genus can read are:

1. Variation due to encryption pragma on page 317
2. Variation due to type of encryption key on page 319

Variation due to encryption pragma

At the time of encrypting HDL files, the user needs to differentiate the region of the file that needs encryption. This is indicated through pragma comments in the HDL file, that surround the region to be encrypted. The pragma `protect begin` and `protect end` are used in the comments that surround the region. These pragma comments are of two types:

1. HDL specific – In Verilog, pragmas start with // or /*... */ and in VHDL, the pragmas start with `--`.

   VHDL Example:

   ```
   --pragma protect
   --pragma protect begin

   library ieee;
   use ieee.std_logic_1164.all;
   use work.p.all;
   entity test_03 is
   port(d :  in std_ulogic_vector(0 to 31);
   q : out std_ulogic_vector(0 to 31));
   end;

   architecture rtl of test_03 is
   component sub
   generic(g : integer_vect);
   port(d :  in std_ulogic_vector(0 to 31);
   q : out std_ulogic_vector(0 to 31));
   end component;

   begin
   u1: sub
   generic map(g => (10, 12, 15, 17))
   port map(q => q, d => d);
   end;

   --pragma protect end
   ```

   Verilog Example:

```
//pragma protect
//pragma protect begin

module t_01_nand(q, d1, d2);
output q;
input d1, d2;
nand(q, d1, d2);
endmodule

//pragma protect end
```

2. IEEE standard Designer — Use the IEEE standard syntax of `` `pragma protect `` (for Verilog) and `` `protect `` (for VHDL) for encapsulating the protected sections.

   Example of Verilog file containing `` `pragma: ``

```
`pragma protect
`pragma protect begin

module t_01_nand(q, d1, d2);
output q;
input d1, d2;
nand(q, d1, d2);
endmodule

`pragma protect end
```

   Example of VHDL file containing `` `protect: ``

```
`protect begin

library ieee;
use ieee.std_logic_1164.all;
use work.p.all;
entity test_03 is
port(d : in std_ulogic_vector(0 to 31);
q : out std_ulogic_vector(0 to 31));
end;

architecture rtl of test_03 is
component sub
generic(g : integer_vect);
port(d : in std_ulogic_vector(0 to 31);
q : out std_ulogic_vector(0 to 31));
end component;

begin
u1: sub
generic map(g => (10, 12, 15, 17))
port map(q => q, d => d);
end;

`protect end
```

Note: Use the `-pragma` option of the encrypt command to encrypt only the pragma protected sections of the input files. Otherwise, if `-pragma` option is not specified, the complete file will be encrypted. When using `xmprotect` for encryption, the `-pragma` option is not required for partial encryption.

Variation due to type of encryption key

Encryption tools (like `xmprotect`), use a key (generally an ASCII text used in encryption algorithms) to control the encryption of the input text. This key is essential at the time of decrypting the encrypted content. IPs can be protected either using a Default key method or User key method (using `xmprotect`). Genus supports reading of RTL files which were encrypted using either of these methods.

Default key method

In this method, `xmprotect` uses its default internal key for encryption when an explicit key is not provided. `xmprotect` makes the key information part of the encrypted text.

- ❑ Encryption on the Genus Prompt

  ```
  genus:root: 12> encrypt test_1.v
  ```

  Default extension used by `xmprotect` is `.vp`. The encrypted file is named `test_1.vp`. Note that this file will be fully encrypted as `-pragma` switch is not specified.

- ❑ Encryption on Linux Prompt

  ```
  % xmprotect test_1.v
  ```

  The protected file is named `test_1.vp`.

  Files protected with default key method, can be read by Genus, without requiring any special setting.

  ```
  genus:root: 13> read_hdl test_1.vp
  ```

User key method

In this method, the IP designer provides a key for encryption. At the time of encryption, he has to follow these steps to encrypt the input files:

1. Generate a key file, using the `-RSAKeyGenerate` command option of `xmprotect`.

   User may provide a seed string to generate the key. In the example below, string “hello world” is used to generate the key. User may also provide path and name of the file, in which the key is stored (default filename is *`key`*).

   ```
   % mkdir ./mykeys
   % setenv seed "hello, world"
   % xmprotect -rsakeygenerate -seed $seed -keyname ./mykeys/key
   ```

2. Set the environment variable `NCPROTECT_KEYDB` to point to the path of the user key file in step 1.

```
% setenv NCPROTECT_KEYDB ./mykeys // Here directory mykeys has a file with
the name key.
```

3. Call `xmprotect` to encrypt the design files. `xmprotect` automatically detects the `NCPROTECT_KEYDB` environment variable, infers that user key is to be used for encryption and uses this key to encrypt the files.

The designer needs to provide the key file to Genus to enable it to read the encrypted files through `read_hdl`.

1. Set the environment variable `NCPROTECT_KEYDB` to the path of the user key file.

   ```
   set env(NCPROTECT_KEYDB) home/var/mykeys
   ```

2. Read the design using `read_hdl` in Genus session.

   ```
   genus:root: 15> read_hdl test_1.vp
   ```

Levels of Protection

Often designers do not want the synthesized netlist obtained from an encrypted (partially or fully) input design to be written out as an encrypted Verilog. In such a case, no special protection is required for the synthesized netlist. However in some situations, designers may want that Genus (while writing out the Verilog of synthesized netlist), should encrypt the part of the netlist that was derived from an encrypted HDL module. These two variants of treatment of encrypted HDL, are referred as two different levels of protection. This degree of IP protection, where Genus encrypts the output Verilog for parts derived from encrypted modules, is called as the **Level-1** protection (for reference, as there is no standard naming convention). Similarly, for the designs where output netlist is not required to be encrypted (referred to as clear text), the level of protection is referred to as **Level-0** protection.

The level of protection chosen by Genus, for an encrypted module, is determined by the choice of options used during encryption using `xmprotect`. By default, Level-0 protection is considered. Example:

Level-0 Protection with the following command:

```
% xmprotect -lang vhdl -autoprotect -synthesis output_netlist:cleartext -
synthesis viewers:debugall test_1.vhd
```

OR

```
% xmprotect -lang vhdl -autoprotect test_1.vhd
```

Level-1 Protection with the following command:

```
% xmprotect -lang vhdl -autoprotect -synthesis output_netlist:none -synthesis
viewers:none test_1.vhd
```

Level-1 protection for the encrypted module means the Verilog for the module is encrypted in the output of `write_hdl` command.

Caution

There is a caveat about Genus's level-1 protection support. An expert user can use report_* commands, the user interface, and the LOG file to get information about the protected parts of the synthesized design. The report_* commands, the Tcl command interface and the LOG messages are not yet capable of hiding information about protected parts of the synthesized design.

Round-trip Protection Flow

The designers who are contributing encrypted IPs, sometimes want special password based protection for their encrypted files (using `xmprotect`). The expectation of designer is to generate a password at the time of encryption of IP, perform synthesis and other EDA steps in a safe manner. That is, any synthesis output written in files, should have those parts as encrypted, which were derived from an encrypted input. The encryption of output files, should be done using the same password, which was generated at the time of encryption. And only the IP owners, should be able to decrypt, using `xmprotect`, the synthesis output files by providing the original password file. For, such a flow possibility, the EDA tools (that are part of design flow), need ability to not only read the encrypted design input, but also encrypt the output, derived from them, using the same password. The designers should be able to decrypt the final output using the same password key.

We refer to such an IP protection based EDA flow as “Round-trip Protection Flow”. If the encryption or decryption utility is `xmprotect`, Genus has the ability to be a part of this round-trip protection flow.

To use the round-trip protection flow, designer needs to do two things:

1. Generate EIF (Encryption Information File) during encryption:

   EIF file contains a combination of a special key and an algorithm set in an encrypted format. This file contains all the information needed by `xmprotect` to decrypt the netlist. To generate eif, use the -`generate_eif` option of `xmprotect`. For example,

   ```
   % xmprotect -generate_eif clear.eif -outdir ./newdir -lang vhdl counter.vhd
   ```

   `counter.vhd` file is now protected for round-trip protection flow. The output password file (`clear.eif`) will be kept in the folder `newdir`. The output file is `netlist.hdp.`

2. Convert the encrypted netlist back to clear text.

   To decrypt the netlist, use `-decrypt_with_eif` option

   ```
   % xmprotect -decrypt_with_eif ./newdir/clear.eif -language vhdl netlist.hdp
   ```

   This clear text output file will be kept in the folder where `eif` file resides.
   You can use the -`outdir` option of `xmprotect` to change the output directory.

Note: While reading the IP files during synthesis, the password file is not required.

With -`generate_eif`, Xcelium sets the protection level (for Genus) to be Level-1. Refer to the Levels of Protection for more details.

Details and Examples of Protection Features

■ Encrypting Designs within Genus on page 323

■ Encrypting Designs outside Genus on page 324

■ Loading Encrypted Designs on page 325

■ Writing Encrypted Designs on page 326

■ Attributes — "protected" and "encrypted" on page 326

Encrypting Designs within Genus

Genus uses the `xmprotect` encryption and decryption library that is also used by the Cadence NC-Verilog and Cadence NC-VHDL Simulators.

The Genus `encrypt` command takes a plain text file, encrypts it, and then writes out an encrypted file.

```
genus:root: 21> encrypt -vhdl ksable.vhdl > ksable_encrypted.vhdl
genus:root: 22> read_hdl -language vhdl ksable_encrypted.vhdl
```

By default, the command encrypts for basic Level-0 protection. In the above example, since the `-pragma` command option of the encrypt command is not used for encryption, the complete file will the encrypted.

All protection is achieved through the use of pragmas. You can encrypt sections of the files (HDL Modules) by enclosing them with protection pragmas.

The following example illustrates Verilog code with Verilog style NC Protect pragmas. You must specify `//pragma protect` before specifying the protected block. Begin the section with `//pragma protect begin` and end with `//pragma protect end` pragmas.

```
 module secret_func (y, a, b);
    parameter w = 4;
    input [w-1:0] a, b;
    output [w-1:0] y;
// pragma protect
// pragma protect begin
    assign y = a & b;
// pragma protect end
endmodule
```

Specify the `-vlog` and `-pragma` options together to encrypt only the text between the pragmas. The `encrypt` command encrypts the original Verilog file (`ori.v`) containing the NC Protect pragmas. The encrypted file is called `enc.v`.

```
genus:root: 25> encrypt -vlog -pragma org.v > enc.v
```

The following example illustrates VHDL code with VHDL style NC Protect pragmas. You must specify `--pragma protect` before specifying the beginning (`--pragma protect begin`) and ending (`--pragma protect end`) pragmas.

```
 entity secret_func is
    generic (w : integer := 4);
    port (  y: out bit_vector (w-1 downto 0);
        a, b: in bit_vector (w-1 downto 0)      );
end;

-- pragma protect
-- pragma protect begin
architecture rtl of secret_func is
begin
    y <= a and b;
end;
-- pragma protect end
```

Specify the `-vhdl` and `-pragma` options together to only encrypt the text between the pragmas. The `encrypt` command encrypts the original VHDL file (`ori.vhdl`) containing the NC Protect pragmas. The encrypted file is called `enc.vhdl`:

```
genus:root: 26> encrypt -vhdl -pragma org.vhdl > enc.vhdl
```

Encrypting Designs outside Genus

Use NC-Protect to encrypt RTL files. The encryption key can be either the `xmprotect` default key or any non-default user-provided key of his choice. The level of protection can be either Level-0 or Level-1, depending on the encryption methodology used.

```
% xmprotect -lang vlog -autoprotect test_1.v
```

Loading Encrypted Designs

Encryption is supported by the parser in both the RTL mode and the structural mode, that is. by all the following commands:

- `read_hdl`
- `read_hdl -netlist`
- `read_netlist`

Genus can understand whether a file is encrypted, and process it accordingly. These commands can take any mixture of plain-text and encrypted files, in any order. For example:

```
read_hdl plain_1.v enc_2.v plain_3.v enc_4.v
```

or

```
read_hdl -language vhdl enc_1.vhd plain_2.vhd enc_3.vhd plain_4.vhd
```

Each HDL file can be completely in plain text, fully encrypted, or a mixture of plain and encrypted text (partially encrypted).

If a design is described in multiple files, it can be:

- A mixture of plain-text and encrypted files
- A mixture of Verilog and VHDL files

Hence, if you are using one `read_hdl` or `read_netlist` command to load multiple files, they can be a mixture of plain-text and encrypted files.

In each of the loaded HDL files, where each Verilog file describes one or more modules while each VHDL file describes one or more entities or packages:

- Each encrypted HDL file can be either fully or partially encrypted
- Each encrypted `module` or `entity` can be either fully or partially encrypted
- Each encrypted module or entity has one or more protection blocks and each protection block is enclosed by a pair of NC-Protect pragmas

Reading encrypted designs which have been encrypted by non-cadence tools

Genus can read designs that have been encrypted using non-cadence tools provided they have been encrypted using the P1735 IEEE standard.

VHDL Example – The following example shows a VHDL file which was encrypted using a non-cadence tool:

```
`protect begin_protected
`protect encrypt_agent=<non-cadence tool>
`protect encrypt_agent_info=<abc version>
`protect data_keyowner=<...>

…….....
P]X9LXJ99W999999+KO27xYqi7kINbzQmPPcTVAZ2+e/mGvuOSthZNlZvWr+g7/0Av4hzO8ZYZQx
5RtvdMBcKFo1kzJUTv9A7+JsXQmdmGwsYsKFyqWSlzfMLEksLn1ltqe8FasgxOu7umDqWpWmcqlf
….....

`protect end_protected
```

Writing Encrypted Designs

Use the `write_hdl` command to write any encrypted (or partially encrypted) design as you would with a non-encrypted design. With protected modules which have Level-1 protection, the `write_hdl` command would encrypt their gate-level design description on a module-by-module basis. With each Level-1 protected module, the generated netlist is encrypted using the same encryption key or method as its source code and is marked at the same level of protection as its source code.

Attributes — "protected" and "encrypted"

`hdl_architecture` objects that are created with the `read_hdl` command, get tagged with boolean attributes `encrypted` and `protected`.

- An `hdl_architecture` attribute `encrypted` is `true` if and only if the definition of the architecture in input HDL file was partially or fully encrypted.
- The attribute `protected` is set to `true`, if and if only if the architecture was assigned protection Level-1.
- After elaboration, the design or module also gets tagged with boolean attribute named `protected`. For a given design or module object, the attribute `protected` is `true` if and only if its protection level is Level-1.

For details on protection levels, refer Levels of Protection on page 321

Propagation of design or module attribute "protected"

An internally-generated tool-defined module (e.g. multiplier module) is `protected` if its parent module is `protected`.

If a certain tool-defined module (for example, `mult_unsigned`) is instantiated by both a protected parent module and an unprotected parent module, the child module (`mult_unsigned`) is uniquified as two objects, one protected and the other one not.

If an unprotected module instantiates a protected module, and the child module is ungrouped (using the `ungroup` command), the parent module becomes a protected one and Genus issues a warning message.

A
Simple Synthesis Template

The following script is a simple script which delineates the very basic Genus flow.

```
# ********************************************************
# *
# * A very simple script that shows the basic Genus flow
# *
# ********************************************************

set_db init_lib_search_path <full_path_of_technology_library_directory>
set_db init_hdl_search_path <full_path_of_hdl_files_directory>

set_db library <technology_library>
read_hdl <hdl_file_names>

elaborate <top_level_design_name>

set clock [create_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>

syn_generic

syn_map

report_timing >  <specify_timing_report_file_name>
report_area   > <specify_area_report_file_name>

write_hdl > <specify_netlist_name>
write_script > <script_file_name>

quit
```

Index

Symbols

204
‘define statements
  defining a macro 134
  reading a design with Verilog macros 136

A

arithmetic packages 143
attributes
  encrypted 326
  gen_module_prefix 53
  hdl_language 130
  hdl_parameter_naming_style 159
  hdl_track_filename_row_col 162
  hdl_vhdl_case 144
  hdl_vhdl_environment 141, 143
  hdl_vhdl_lrm_compliance 143
  information_level 33
  init_hdl_search_path 132, 156
  path 104
  preserve 189
  protected 326
  super_thread_rsh_command 206

B

bitblasting, all ports 298, 299
blackbox 219
boundary optimization 194
buffers, inserted
  removing between hard regions 201

C

capacitance units 183
case_insensitive 293
cells
  preserving 189
  sequential cell mapping 196
clocks
  defining 183
  definition 64
  domains 183
  removing 184
commands
  create_clock 183
  delete_obj 184
  elaborate 155
  get_db 90
  new_seq_map 196
  read_hdl 127, 133
  read_sdc 183
  redirect 286
  report gates 275
  report timing 273
  report_area 276
  syn_gen 241
  syn_map 241
  ungroup 86
  write_hdl 302
  write_sdc 183
constants
  definition 58
constraints
  timing 183
cost_groups
  definition 64
  generating timing reports for 274
Critical Region Resynthesis 214

D

default values
  override
    for generics 159
delete_unloaded_seqs 194
design
  definition 54, 55, 56
design constraints
  applying 181
  timing 183
design information hierarchy 52
designs, encrypting 323
DRC 184
drc_first 200

E

elaborate design 159
encrypting designs 323
exceptions
    definition 64
    generating timing reports for 274
excluding library cells 107
external_delays
    definition 65

F

finding
    HDL files 103
    libraries 103
    scripts 103

G

gen_module_prefix 313
genus_startup 28
group total worst slack 240
grouping 190

H

hard_region 201
HDL designs
    specifying language 130
HDL files
    directories, specifying 132
    setting the search path 103
HDL search paths
    specify 132, 156
hdl_architechture
    definition 74
hdl_bind
    definition 79
hdl_block
    definition 74
hdl_component
    definition 79
hdl_configuration
    definition 79
hdl_implementation
    definition 79
hdl_inst
    definition 75
hdl_label
    definition 75
hdl_lib
    definition 74
hdl_operator
    definition 81
hdl_package
    definition 81
hdl_paramater
    definition 79
hdl_pin
    definition 79
hdl_procedure
    definition 76
hdl_reg_naming_style 294
hdl_subprogram
    definition 76
hierarchy
    created by Genus 53
hinsts
    definition 59
hport
    and pin, difference 65

I

incremental optimization (IOPT) 240
Information 33
information_level 33
init_lib_search_path 103

L

lbr 108
lib_arcs
    definition 69
lib_cell
    definition 69
lib_pin
    definition 70
libraries
    predefined
        Synergy 142
library
    definition 68
library cells
    excluding 107

library format 108
  converting 108
  lib 108
Licensing 27
log file
  generating 32

M

macros
  Verilog 133
mapping sequential cells 196
Messages
  setting the level 33
messages
  definition 55
  in log file 32
module
  synthesize 156
modules
  correlating to filename 92
  definition 63
  naming 53
  preserving 189
  ungrouping 86
multiple CPUs 204
multiple machines 204

N

name
  individual bits of array ports and registers 156
nets
  definition 61
new_seq_map 196

O

operating_conditions
  definition 71
optimization settings 187

P

parameter
  override default values 155
parameter values, propagating 159
pin
  and hport, difference 65
pin_busses
  definition 61
pins
  definition 61
port_busses
  definition 62
ports
  bitblasting 298
  definition 62
predefined
  VHDL
    Environments 141
preserve_cell 189
preserving cells 189
preserving modules 189

R

redundancy removal 213
replace_str 293
report timing -lint 273
reports
  runtime 241
root
  definition 54
runtime
  reporting 241

S

scripts
  recording in a log file 32
  running 126
  setting the search path 103
SDC constraints 183
search paths
  setting 103
sequential cell mapping 196
set_output_delay 38
setting search paths 103
setting the level 33
size_delete_ok 189
size_ok 189
structural constructs
  Verilog 148
subdesign synthesis 218

subport_busses
  definition 63
subports
  definition 63
Super Threading
  SGE 209
super_thread_servers
  batch 207
supported
  VHDL
    environments 141
    libraries Synergy environment 142
Synergy
  predefined VHDL libraries 142
synthesis scripts
  running 126

T

technology library
  setting the search path 103
  setting the target library 105
timing constraints 183
timing models 117
  reasons for a cell to be marked as timing model 117
timing reports
  generation 273
timing units 183

U

ungrouping 190
  automatic 192
units
  capacitance 183
  timing 183
unusable cells
  reasons for a cell to be marked unusable 112
  use during synthesis 112
update_names -prefix 293

V

Verilog
  keep track of RTL source code 162
  version, specifying 130
VHDL
  arithmetic packages from vendors 143
  case, specifying 144
  code compliance with LRM, enforcing 143
  environment, setting 141
  predefined
    environments 141
    Synergy libraries 142
  show mapping between libraries and directory 143

W

wire-load models
  definition 71
write_hdl -generic 304
write_sdc 305
write_sdf 306

X

xm_protect_version 316