Genus Synthesis Flows Guide

Product Version 19.1
November 2019

Contents

Preface

About This Manual

This manual describes various synthesis flows available to the user when using the Genus software with the Stylus 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.
- **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: 11>
  ```

- **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: 12> 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: 15> 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: 17> 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: 18> 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: 24> help path_group
```

This returns the syntax for the `path_group` command.

Getting Attribute Help

Type the following:

```
genus:root: 28> help attribute_name
```

For example:

```
genus:root: 31> 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 if you only know part of its name, even as little as one letter.

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

  For example:

  ```
  genus:root: 38> a <Tab>
  ```

  This returns the following commands:

  ```
  add_assign_buffer_options        add_clock_gates_obs
  add_clock_gates_test_connection  add_opcg_hold_mux
  add_tieoffs                      add_to_collection
  after                            alias
  all_clocks                       all_connected
  all_fanin                        all_fanout
  all_inputs                       all_instances
  all_outputs                      all_registers
  analyze_library_corners          analyze_scan_compressibility
  analyze_testability              append
  append_to_collection             applet
  apply                            apropos
  array                            assemble_design
  attribute_exists                 auto_execok
  auto_import                      auto_load
  auto_load_index                  auto_qualify
  ```

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

  For example:

  ```
  genus:root: 41> path_<Tab>
  ```

  This returns the following commands:

  ```
  path_group
  ```

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: xx>` 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.

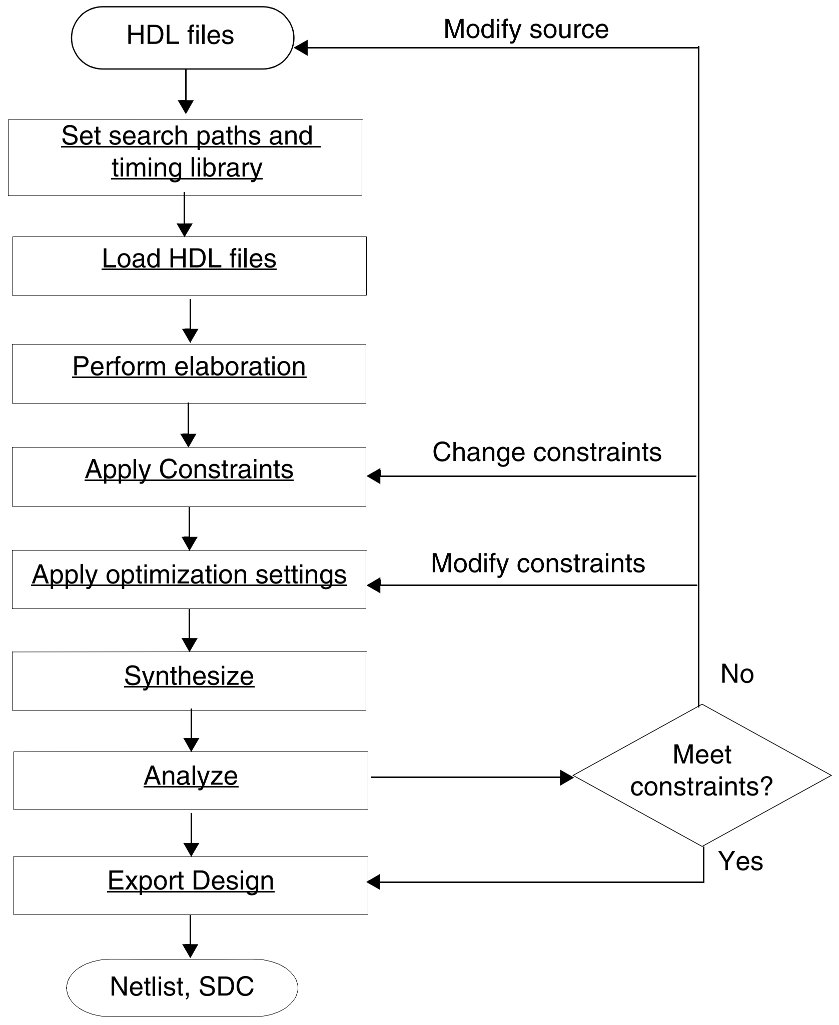
1

Getting Started with the Generic Flow

Overview

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

Figure 1-1 Generic Genus Work Flow



Tasks

Specifying Explicit Search Paths

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

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

```
genus:root: 11> set_db init_lib_search_path path
genus:root: 12> set_db script_search_path path
genus:root: 13> set_db init_hdl_search_path path
```

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

Setting the Target Technology Library

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

➤ To specify a single library:

```
genus:root: 16> set_db library lib_name.lib
```

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

```
genus:root: 21> set_db library /usr/local/files/lib_name.lib
```

➤ To specify a single library compressed with gzip:

```
genus:root: 25> set_db library lib_name.lib.gz
```

➤ To append libraries:

```
genus:root: 27> 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.

Setting the Appropriate Synthesis Mode

Genus has two modes to synthesize the design. The synthesis mode is determined by the setting of the `interconnect_mode` attribute:

- `wireload` (default) indicates to use wire-load models to drive synthesis
- `ple` indicates to use Physical Layout Estimators (PLEs) to drive synthesis

  PLE uses physical information, such as LEF libraries, to provide better closure with back-end tools.

  Note: When you read in LEF files, the `interconnect_mode` attribute is automatically set to `ple`.

Loading the HDL Files

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.

➤ To load one or more Verilog files:

- ❑ You can read the files sequentially:

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

- ❑ Or you can load the files simultaneously:

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

- ❑ You can also read the design from simulation optionfiles:

```
read_hdl -f optionfile
```

For more information on loading the design from optionfiles, see Reading Designs in Simulation Environment on page 59.

Caution

Your files may have extra, hidden characters (for example, line terminators) if they are transferred from Windows/Dos to the UNIX environment using the dos2unix command. Be sure to eliminate them because Genus will issue an error when it encounters these characters.

For more information on loading HDL files, see *Loading Files* in the *Genus User Guide*.

Performing Elaboration

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

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

Note: If there are any gate-level netlists read in with the RTL files, 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.

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.

For more information on elaborating a design, see *Elaborating the Design* in the *Genus User Guide*.

Applying Constraints

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

- Operating conditions
- Clock waveforms
- I/O timing

You can apply constraints in several ways:

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

Applying Optimization Constraints

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

With Genus, you can perform any of the following optimizations:

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

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

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

For each path group, the worst timing path drives the synthesis cost function. For more information on optimization strategies and related commands, see *Defining Optimization settings* in the *Genus User Guide*.

Performing Synthesis

After the constraints and optimizations are set for your design, you can proceed with synthesis.

➤ To synthesize your design, type the following commands:

```
genus:root: 34> syn_generic
genus:root: 35> syn_map
```

For details on the synthesis commands, see *Command Reference*.

After these two steps, you will have a technology-mapped gate-level netlist.

Analyzing the Synthesis Results

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

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

For more information on generating reports for analysis, see *Generating Reports* in the *Genus User Guide* and "Analysis and Report Commands" in the *Genus Command Reference*.

Exporting the Design

The last step in the flow involves exporting the design post synthesis to write out the gate-level netlist, design and SDC constraints.

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

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

```
genus:root: 21> write_hdl > design.v
```

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

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

```
genus:root: 25> write_script > constraints.g
```

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

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

```
genus:root: 31> write_sdc > constraints.sdc
```

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

Note: Because some place and route tools require different structures in the netlist, you may need to make some adjustments either before synthesis or before writing out the netlist. For more information about these issues, see *Interfacing to Place and Route* in the *Genus User Guide*.

➤ To export all necessary files for the Innovus System, use the following command:

```
genus:root: 34> write_design -innovus design_name
```

For more information on this topic, see *Interfacing to Place and Route* in the *Genus User Guide*.

Exiting Genus

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

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

Generic Flow

```
#general setup
#--------------
set_db init_lib_search_path path         #optional
set_db init_hdl_search_path path        #optional

#load the library
#------------------------------
set_db library library_name

#load and elaborate the design
#------------------------------
read_hdl design.v
elaborate

#specify timing and design constraints
#--------------------------------------
read_sdc sdc_file                       #optional

#add optimization constraints
#----------------------------            #optional
.....
#synthesize the design
#---------------------
syn_generic

syn_map

#analyze design                          #optional steps
------------------
report_area
report_timing
report_gates

#export design
#-------------                           #optional steps
write_hdl > dessign.vm
write_sdc > constraints.sdc
write_script > constraints.g

# export design for Innovus
#-----------------------
write_design [-basename string] [-gzip_files] [-tcf]
[-innovus] [-hierarchical] [design]
```

2
Express Flow

Introduction

The Express flow provides the capability to perform *quick synthesis* in the early stages of the design, which is important to reduce the total design time and the time to meet target constraints, such as for timing and power. While the best final performance, power, and area (PPA) are critical for the last push of timing closure through the implementation flow, the bulk of the synthesis usage across the life cycle of a design is tolerant to some PPA inefficiencies as long as the accuracy is sufficient to make feasibility trade-offs or quickly clean up the design at its inception. These clean-up and feasibility stages of the design process are the target for the Express flow.

The Express flow is based on the generic synthesis flow, but uses a few special settings (outlined further in this chapter) which allow you to

- Start synthesis with incomplete and inconsistent HDL
- Start synthesis with incomplete and buggy SDC constraints
- Run a fast synthesis with reasonable QoR which gives you a good indication of the feasibility of the design

Important

These synthesis features are limited access features. Contact your Cadence representative to qualify your usage and make sure the feature(s) meet your needs.

Starting Synthesis with Incomplete HDL

Genus supports a “prototype” clean-up mode to allow analysis of designs that are still in the early stages of development, before all of the module descriptions are fully specified and the RTL is still being written and refined. In this mode, you can synthesize full designs from multiple HDL sources, in the presence of inconsistencies, and mismatches of interfaces of the blocks in the HDL descriptions. The Express flow can be tolerant of these early design discrepancies and continue along through the synthesis by making assumptions on the user’s intent, whereas it may have erred out in a standard production flow run.

To synthesize a design with incomplete HDL, set the following root attribute:

```
set_db proto_hdl true
```

When this attribute is set to `true`, Genus relaxes error checking in the `read_hdl` and `elaborate` commands, and makes assumptions about how to implement the erroneous HDL, so that synthesis results can be generated early in the design phase. Note that this mode is for estimating results only. The netlist generated for a design with incomplete HDL might not be functionally correct.

Supported Types of Incomplete HDL

- Extra Named Instance Ports on page 32
- Port Name Case Mismatch on page 35
- Mapping Record Ports to Blasted Module Ports on page 36
- Width Mismatch on page 39
- Missing Module Descriptions on page 43

Extra Named Instance Ports

During Prototype Synthesis, Genus implements extra named ports in instantiations of hierarchical modules as follows:

- It adds the extra ports to the module as unconnected **inout** ports.
- It assumes that the size of the port is the maximum of sizes seen for that port on all instantiations of the hierarchical module.
- It flattens complex ports.

Genus does not support extra, positional ports.

Example 2-1 One module instance has extra ports

Input HDL

```
module top (in1, in2, out1, out2);
    input in1, in2;
    output out1, out2;
    mid u1 (.in1(in1), .in2(in2), .out1(out1), .out2(out2));
    mid u2 (.in1(in1), .out1(out1));
endmodule

module mid(in1,out1);
    input in1;
    output out1;
    assign out1 = in1;
endmodule
```

Netlist after elaborate

```
module mid(in1, out1, in2, out2);
    input in1;
    output out1;
    inout in2, out2;
    wire in1;
    wire out1;
    wire in2, out2;
    assign out1 = in1;
endmodule

module top(in1, in2, out1, out2);
    input in1, in2;
    output out1, out2;
    wire in1, in2;
    wire out1, out2;
    wire UNCONNECTED, UNCONNECTED0, UNCONNECTED1, UNCONNECTED2;
    mid u1(.in1 (in1), .out1 (out1), .in2 (in2), .out2 (out2));
    mid u2(.in1 (in1), .out1 (out1), .in2 (UNCONNECTED), .out2 (UNCONNECTED0));
endmodule
```

Example 2-2 Multiple module instances have extra ports

Input HDL

```
module mid(in1, out1);
    input in1;
    output out1;
    assign out1 = in1;
endmodule

module top (in1, in2, out1, out2, out3);
    input in1, in2;
    output out1, out2, out3;
    mid u1 (.in1(in1), .in2(in2), .out1(out1));
    mid u2 (.in1(in1), .out1(out3), .out2(out2));
endmodule
```

Netlist after elaborate

```
module mid(in1, out1, in2, out2);
    input in1;
    output out1;
    inout in2, out2;
    wire in1;
    wire out1;
    wire in2, out2;
    assign out1 = in1;
endmodule

module top(in1, in2, out1, out2, out3);
    input in1, in2;
    output out1, out2, out3;
    wire in1, in2;
    wire out1, out2, out3;
    wire UNCONNECTED, UNCONNECTED0, UNCONNECTED1, UNCONNECTED2;
    mid u1(.in1 (in1), .out1 (out1), .in2 (in2), .out2 (UNCONNECTED));
    mid u2(.in1 (in1), .out1 (out3), .in2 (UNCONNECTED1), .out2 (out2));
endmodule
```

Example 2-3 Module instance has an extra complex port

Input HDL

```
typedef struct packed { logic [1:0] u; logic [1:0] v;}RTR_LOC;
module top(input RTR_LOC S1, input in1, output out);
    rtr u0 (.in1(S1), .in2(in1), .out1(out));
endmodule

module rtr(input in2, output out1);
    assign out1 = in2;
endmodule
```

Netlist after elaborate

```
module rtr(in2, out1, in1);
    input in2;
    output out1;
    inout [3:0] in1;
    wire in2;
    wire out1;
    wire [3:0] in1;
    assign out1 = in2;
endmodule
module top(\S1[v] , \S1[u] , in1, out);
    input [1:0] \S1[v] , \S1[u];
    input in1;
    output out;
    wire [1:0] \S1[v] , \S1[u];
    wire in1;
    wire out;
    rtr u0(.in2 (in1), .out1 (out), .in1 ({\S1[u] , \S1[v] }));
endmodule
```

Port Name Case Mismatch

If an instance port name in a Verilog design does not match any module port name, but it does match a module port when compared case-insensitively, the instance port name is changed to match the case of the corresponding module port name.

Example 2-4 Case mismatch between port names in module and instance

Input HDL

```
module top(in1, out1);
    input in1;
    output out1;
    mid u1 (.A(in1),.B(out1));
endmodule

module mid(a, b);
    input a;
    output b;
    assign b = a;
endmodule
```

Netlist after elaborate

```
module mid(a, b);
    input a;
    output b;
    wire a;
    wire b;
    assign b = a;
endmodule

module top(in1, out1);
    input in1;
    output out1;
    wire in1;
    wire out1;
    mid u1(.a (in1), .b (out1));
endmodule
```

Mapping Record Ports to Blasted Module Ports

When a structure or record signal is connected to an instance port, and the module definition does not have a port with the same name, Genus attempts to identify a set of module ports that represent individual elements of the complex port (a *blasted* representation), and connects the elements of the complex signal to the blasted ports. This linking ability applies to instances of both hierarchical modules and technology components.

Genus uses the `hdl_record_naming_style` root attribute to identify blasted structure ports.

Note: This methodology is enabled for record/struct ports regardless of the setting of the `proto_hdl` root attribute.

Example 2-5 Mapping a structure port to a blasted port

In this example, Genus connects `in[u]` and `in[v]` of complex signal `in` to the blasted module ports `\p[u]` and `\p[v]`, respectively.

Input HDL

```
typedef struct packed {
    logic [1:0] u;
    logic [1:0] v;
    } RTR_LOC;

module top(input RTR_LOC in, output out);
    rtr u0 (.p(in), .q(out));
endmodule

module rtr(\p[v] , \p[u] , q);
    input [1:0] \p[v] , \p[u];
    output q;
    assign q = \p[v] [0];
endmodule
```

Netlist after elaborate

```
module rtr(\p[v] , \p[u] , q);
    input [1:0] \p[v] , \p[u];
    output q;
    wire [1:0] \p[v] , \p[u];
    wire q;
    assign q = \p[v] [0];
endmodule

module top(\in[v] , \in[u] , out);
    input [1:0] \in[v] , \in[u];
    output out;
    wire [1:0] \in[v] , \in[u];
    wire out;
    rtr u0(.\p[v] (\in[v] ), .\p[u] (\in[u] ), .q (out));
endmodule
```

Example 2-6 Mapping a structure port to a blasted port using hdl_record_naming_style

In this example, assuming that the `hdl_record_naming_style` root attribute is set to `"%s_%s_",` the complex port `in` of module `top` is mapped to the blasted ports `p_u_` and `p_v_` in module `rtr` based on the naming style.

Input HDL

```
typedef struct packed {
    logic [1:0] u;
    logic [1:0] v;
    } RTR_LOC;

module top(input RTR_LOC in, output out);
    rtr u0 (.p(in), .q(out));
endmodule

module rtr(p_v_, p_u_, q);
    input [1:0] p_v_, p_u_;
    output q;
    assign q = p_v_[0];
endmodule
```

Netlist after elaborate

```
module rtr(p_v_, p_u_, q);
    input [1:0] p_v_, p_u_;
    output q;
    wire [1:0] p_v_, p_u_;
    wire q;
    assign q = p_v_[0];
endmodule

module top(in_v_, in_u_, out);
    input [1:0] in_v_, in_u_;
    output out;
    wire [1:0] in_v_, in_u_;
    wire out;
    rtr u0(.p_v_ (in_v_), .p_u_ (in_u_), .q (out));
endmodule
```

Mapping Array Ports to Blasted Module Ports

When an array signal is connected to an instance port, and the module definition does not have a matching port name, Genus tries to match the connection with a set of module ports, that represent a blasted version of the array port.

Limitation: Genus can only match an array connection with blasted ports if the names of the blasted array ports conform to the Verilog array reference naming style (`%s[%d]`).

Example 2-7 Mapping of array ports to bit-blasted ports

In this example, the elements of array `in1` in module `top` are connected to ports of module `bot`: `\in1[0]`, `\in1[1]`, and `\in1[2]`.

Input HDL

```
module top(in1, out1);
    input [0:2]in1;
    output [0:3]out1;
    bot I1(.in1(in1), .out1(out1));
endmodule
module bot(\in1[0] , \in1[1] , \in1[2] , \out1[0] , \out1[1] , \out1[2]
                            , \out1[3] );
    input \in1[0] , \in1[1] , \in1[2];
    output \out1[0] , \out1[1] , \out1[2] , \out1[3] ;
    assign \out1[2]  = \in1[2] ;
    assign \out1[1]  = \in1[1] ;
    assign \out1[0]  = \in1[0] ;
endmodule // bot
```

Netlist after elaborate

```
module bot(\in1[0] , \in1[1] , \in1[2] , \out1[0] , \out1[1] , \out1[2]
                            , \out1[3] ) ;
    input \in1[0], \in1[1], \in1[2] ;
    output \out1[0] , \out1[1] , \out1[2] , \out1[3] ;
    wire \out1[0] , \out1[1] , \out1[2] , \out1[3] ;
    assign \out1[2]  = \in1[2] ;
    assign \out1[1]  = \in1[1] ;
    assign \out1[0]  = \in1[0] ;
endmodule
module top(in1, out1);
    input [0:2] in1;
    output [0:3] out1;
    wire [0:2] in1;
    wire [0:3] out1;
    bot I1(.\in1[0] (in1[0]), .\in1[1]  (in1[1]), .\in1[2]  (in1[2]),
            .\out1[0] (out1[0]), .\out1[1] (out1[1]),
            .\out1[2] (out1[2]), .\out1[3] (out1[3]));
endmodule
```

Width Mismatch

The behavior of Genus is the same for both default synthesis (`proto_hdl` attribute set to `false`) and Prototype Synthesis (`proto_hdl` attribute set to `true`) for the following width mismatch scenarios in HDL. Genus gives a warning and generates functionally correct logic as specified in the Verilog and VHDL language reference manuals.

- Port width mismatch—the width of a signal connected to an instance port differs from the width of the port. Depending on the port width and direction, Genus issues the following messages:

  ```
  Warning : Connected signal is wider than input port. [CDFG-464]
  Warning : Connected signal is wider than output port. [CDFG-465]
  Warning : Connected signal is wider than input/output port. [CDFG-466]
  Warning : Input port is wider than connected signal. [CDFG-467]
  Warning : Output port is wider than connected signal. [CDFG-468]
  Warning : Input/output port is wider than connected signal. [CDFG-469]
  ```

- Assignment width mismatch—the widths of expressions on the left- and right-hand sides of an assignment statement differ. Genus gives the following warning:

  ```
  Warning : Incompatible bitwidths in assignment. [CDFG-239]
  ```

  Note: To allow assignment width mismatch in VHDL designs, set the `hdl_vhdl_assign_width_mismatch` attribute (default: `false`) to `true`.

Example 2-8 Signal is truncated to match smaller port width

Input HDL

```
entity bot is
    port(in1 : in bit_vector(0 to 2); out1 : out bit_vector(0 to 2));
end;
architecture arch of bot is
    begin
    out1 <= in1;
end;
entity top is
    port(in1 : in bit_vector(0 to 3); out1 : out bit_vector(0 to 3));
end;
architecture arch of top is
    begin
    I1 : entity work.bot port map(in1, out1);
end;
```

Netlist after elaborate

```
module bot(in1, out1);
    input [0:2] in1;
    output [0:2] out1;
    wire [0:2] in1;
    wire [0:2] out1;
    assign out1[2] = in1[2];
    assign out1[1] = in1[1];
    assign out1[0] = in1[0];
endmodule

module top(in1, out1);
    input [0:3] in1;
    output [0:3] out1;
    wire [0:3] in1;
    wire [0:3] out1;
    bot I1(.in1 (in1[1:3]), .out1 (out1[1:3]));
endmodule
```

Example 2-9 Extra bits are unconnected when the port width is greater than the signal width

Input HDL

```
module top(in, out);
    input [0:2] in;
    output [0:2] out;
    bot u0 (.in(in),.out(out));
endmodule

module bot(in, out);
    input [0:3] in;
    output [0:3] out;
    assign out = in;
endmodule
```

Netlist after elaborate

```
module bot(in, out);
    input [0:3] in;
    output [0:3] out;
    wire [0:3] in;
    wire [0:3] out;
    assign out[3] = in[3];
    assign out[2] = in[2];
    assign out[1] = in[1];
    assign out[0] = in[0];
endmodule

module top(in, out);
    input [0:2] in;
    output [0:2] out;
    wire [0:2] in;
    wire [0:2] out;
    wire UNCONNECTED, n_5;
    bot u0(.in ({n_5, in}), .out ({UNCONNECTED, out}));
endmodule
```

Example 2-10 VHDL assignment width mismatch

Input HDL

```
entity top is
    port(in1 : in bit_vector(0 to 3);
         out1 : out bit_vector(0 to 2));
end;

architecture arch of top is
    begin
    out1 <= in1;
end;
```

Netlist after elaborate

```
module top(in1, out1);
    input [0:3] in1;
    output [0:2] out1;
    wire [0:3] in1;
    wire [0:2] out1;
    assign out1[2] = in1[3];
    assign out1[1] = in1[2];
    assign out1[0] = in1[1];
endmodule
```

Missing Module Descriptions

Early versions of a design often do not include module descriptions for all of the instantiations in the design. The behavior of Genus for missing module descriptions is the same as for default synthesis: Genus creates a black-box module description and issues a warning.

```
Warning : Creating blackbox. [CDFG-428]
```

Example 2-11 Missing Module in HDL

Input HDL

```
module top(in, out);
    input in;
    output out;
    bot I1(.in(in),.out(out));
endmodule
```

Netlist after elaborate

```
module top(in, out);
    input in;
    output out;
    wire in;
    wire out;
    bot I1(.in (in), .out (out));
endmodule
```

Unsupported Types of Incomplete HDL

- For VHDL:
  - Extra instance ports
- For Verilog:
  - Extra ports or complex ports in an array of instances
  - Extra ports, complex ports, and case-insensitive port name matching for ports that are derived from an interface or modport

Reporting Incomplete HDL Scenarios

Consider the following HDL:

```
module top(in1,in2,in3,out1);
    input [2:0] in1;
    input [2:0] in2;
    input [3:0] in3;
    output [1:0] out1;
mid u1 (.in1(in1),.in2(in2),.out1(out1));
bot u2 (.in1(in1),.in2({in2,1'b1}),.in3(in3),.out1(out1));
endmodule;
module bot(in1,in2,in3,out1);
    input [2:0] in1;
    input [3:0] in2,in3;
    output [1:0] out1;
mid I1 (.in1(in1),.in3(in3),.out1(out1));
endmodule
module mid(in1,out1);
    input [2:0] in1;
    output [1:0] out1;
    assign out1 = in1;
endmodule
```

Use the `report_prototype -hdl` command after elaborate to see a summary of the incomplete HDL constructs that Genus identifies in the design:

```
genus:root: 21> report_prototype -hdl
============================================================
  Generated by:           version
  Generated on:           date
  Technology library:     tutorial 1.1
  Operating conditions:   typical_case (balanced_tree)
  Wireload mode:          enclosed
  Area mode:              timing library
============================================================

INCOMPLETE HDL SCENARIO FOR DESIGN: design:top

PORTS SEEN IN INSTANCES BUT MISSING IN MODULE DEFINITION
========================================================

PORT NAME:     in2
INSTANCE NAME: u1
FILE NAME:     missing_port.v
MODULE NAME:   mid


INCOMPLETE HDL SCENARIO FOR SUBDESIGN: module:top/bot

PORTS SEEN IN INSTANCES BUT MISSING IN MODULE DEFINITION
========================================================

PORT NAME:     in3
INSTANCE NAME: I1
FILE NAME:     missing_port.v
MODULE NAME:   mid
```

Starting Synthesis with Incomplete SDC Constraints

During the early design stages, multiple sub-teams might be working on different sub-blocks of the full design. For a designer responsible for the integration at the chip level during feasibility, prototyping or early analysis, the SDC constraints might not be quite complete, for example some multi cycle path exceptions could be missing causing some endpoints to have a huge negative slack compared to the clock and become bottlenecks to the proper optimization of rest of the design. To allow prototype synthesis to proceed in a reasonable manner, the tool needs to ignore these huge negative slack endpoints from intensive optimization and hence the target computation and optimization focuses on the endpoints with feasible timing constraints.

To enable support for incomplete SDC constraints during global mapping, set the following root attribute:

```
set_db proto_feasible_target true
```

When the feature is enabled, the mapper analyzes the expected timing violations for all timing endpoints (for example, sequential gate inputs, primary outputs, inputs of unresolved hierarchical gates, snipped pins) at the target computation phase. If the mapper-estimated timing violation on an endpoint is greater than a threshold (`proto_feasible_target_threshold_clock_pct`), the endpoint is considered having an infeasible target, and a positive path adjust exception is added for this endpoint pin to relax the path. Such endpoints are also excluded from the computation of targets.

The path adjust constraints exclude these infeasible endpoints also from optimization allowing the tool to focus during optimization only on the feasible endpoints with realistic (feasible) timing constraints.

By default, the tool uses a threshold of 75% of the period of the smallest clock active at the endpoint.

To modify the default threshold you can

- Change the percentage of clock period value:

  `set_db proto_feasible_target_threshold_clock_pct` *`integer`*

- Specify an absolute threshold:

  `set_db proto_feasible_target_threshold` *`float`*

Following message is printed just before target message:

```
Following exceptions will be added due to attribute proto_feasible_target for endpoints with unachievable timing constraints
These endpoints will be excluded from target computation

===================================================================================================================
=========
      Exception     proto_ft_adj_1 having adjust value   1597ps is added to endpoint z
```

The mapper adds a positive path adjust value to relax these paths. Because the `path_adjust` exceptions are not cost group specific, the mapper takes the maximum adjust value among all cost groups at that end point. By default, 100% of the maximum mapper-estimated timing violation will be set as adjust value to these `path_adjust` exceptions. You can control the percentage of the slack value to be used with the `proto_feasible_target_adjust_slack_pct` root attribute.

At the end of mapping, a message will be printed.

If the mapper found some unfeasible endpoints, the message contains the number of path_adjust exceptions.

```
Warning : Mapping was done using feasible target feature due to attribute
proto_feasible_target. [PROTO_FT-1]
        : Global mapper has relaxed infeasible constraints using 1 path_adjust
exception(s) prefixed with proto_ft_adj_.
        : This option MUST NOT be used for production, but only for constraint
debugging.
```

If the mapper does not find any unfeasible endpoint, a message similar to the following is printed:

```
Warning : Mapping was done using feasible target feature due to attribute
proto_feasible_target. [PROTO_FT-1]
       : Global mapper had not detected any infeasible constraints. No path_adjust
exceptions were added.
        : This option MUST NOT be used for production, but only for constraint
debugging.
```

Path_adjust Exceptions

Exceptions added by the mapper are prefixed with `proto_ft_adj_` and they remain forever in the database unless explicitly removed. The tool does not add any new exception during global incremental optimization. These exceptions are visible to incremental optimization also.

You can analyze the exceptions after mapping, and decide whether to remove or keep them for the successive incremental optimization step.

Multi-Mode Designs

In case of multi-mode designs, the tool will only add `path_adjust` exceptions for those timing modes that show infeasible targets in that timing mode. Other timing modes (with feasible targets) do not get path adjusts.

Express Synthesis

To get a quick prediction of the feasibility of the design, run the synthesis commands with their corresponding `syn_xxx_effort` attribute set to `express`. Also, set `syn_global_effort` to `express` for setting the effort level for all the synthesize commands with a single attribute. This provides fast runtimes with reasonable QoR results. The quick runtimes allow you to experiment with RTL and floorplan changes.

The express effort relies heavily on super-threading. You should use at least 8 servers.

```
set_db super_thread_servers {machine_names}
set_db max_cpus_per_server integer
```

With the Genus-Physical-Express flow, you have always 8 threads available for free. The attribute `max_cpus_per_server` also influences the number of CPUs used in Innovus.

The license requirements for this flow are the same as the Genus-Physical and Genus-Spatial flow requirements.

Figure 2-1 Logical Express Flow

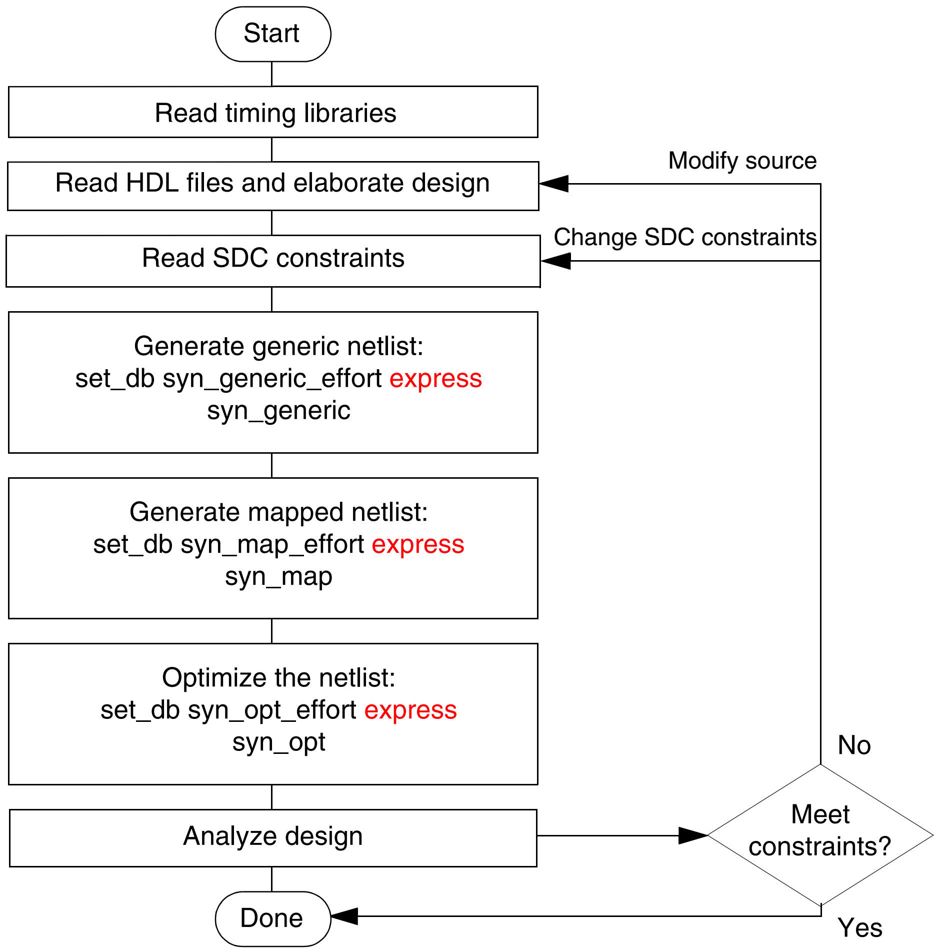
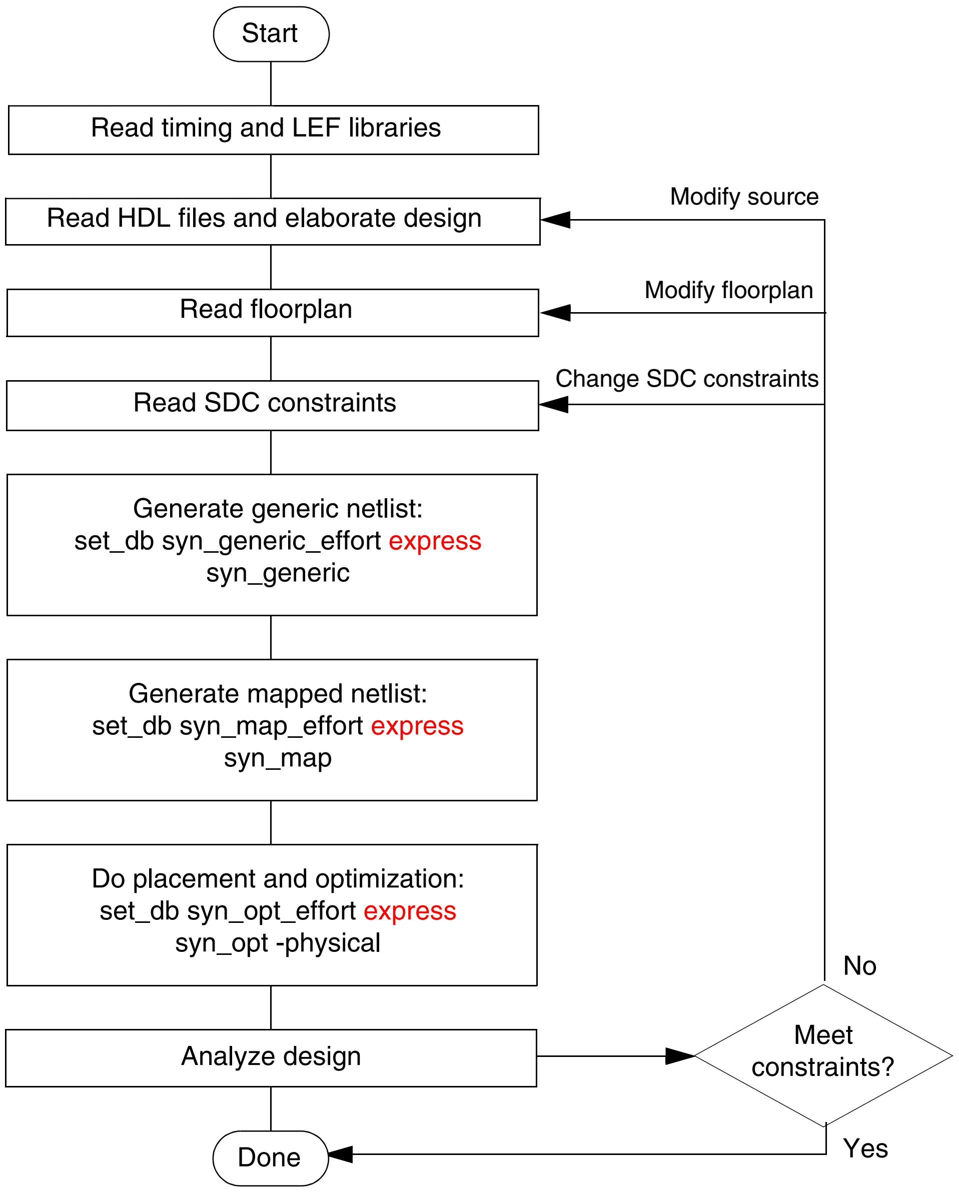


Figure 2-2 Genus-Physical Express Flow


Start
Read timing and LEF libraries
Read HDL files and elaborate design
Modify source
Read floorplan
Modify floorplan
Read SDC constraints
Change SDC constraints
Generate generic netlist:
set_db syn_generic_effort express
syn_generic
Generate mapped netlist:
set_db syn_map_effort express
syn_map
Do placement and optimization:
set_db syn_opt_effort express
syn_opt -physical
Analyze design
Meet constraints?
No
Yes
Done

Express flow can be executed only when the design contains more than 40K instances at the end of the generic synthesis (`syn_generic`). If less than 40K instances are found, a warning (MAP-300) is issued that the design is not appropriate for the express flow and the effort level is moved to the `medium` effort level during `syn_map`. The Genus-Physical express flow provides QoR results within 10% range.

Generic partitions are created during mapping with the `express` effort level. To check for the partition details, set the attribute `super_thread_debug_directory`.

Similarly, the Genus-Spatial express flow will do an express synthesis on the spatial flow for big designs (more than 40K instances). Innovus will not be invoked during the spatial flow.

Attributes Affecting the Express Flow

Attribute Name	Object	Type	Default
cap_table_file	root	string	
innovus_executable	root		default_search_order
interconnect_mode	root	enum	wireload
lef_library	root	string	
lef_stop_on_error	root	boolean	false
max_cpus_per_server	rooty	integer	8
number_of_routing_layers	design	integer	no_value
phys_ignore_nets	design	boolean	false
phys_ignore_special_nets	design	boolean	false
pqos_ignore_msv	root	boolean	false
pqos_ignore_scan_chains	root	boolean	false
qos_report_power	root	string	auto
qrc_tech_file	root	string	
scale_of_cap_per_unit_length	root	double	1.0
scale_of_res_per_unit_length	root	double	1.0
shrink_factor	root	double	no_value
super_thread_debug_directory	root	string	
super_thread_servers	root	string	
syn_generic_effort	root	enumerated type	medium
syn_global_effort	root	enumerated type	none
syn_map_effort	root	enumerated type	high
syn_opt_effort	root	enumerated type	high

Attribute Name	**Object**	**Type**	**Default**
use_area_from_lef	root	enumerated type	false
utilization	layer	double	1.0

3

HDL Features to Aid Verification

Introduction

This chapter describes HDL features that are supported by Genus which can help in verifying the synthesized design.

Preservation of Port Expressions and Port Aliases

Genus supports port expressions and port aliases for the non-ANSI style of port declarations in Verilog modules. It also preserves port expressions and alias information through elaboration and synthesis. The output of the `write_hdl` command will contain the usage of port-expressions and aliases to represent the ports of user modules. This helps in easier verification of the synthesis output netlist by Conformal® Logical Equivalence Checking.

Genus supports the following types of references inside port expressions:

- A simple identifier
- A bit-select of a vector declared within the module
- A part-select of a vector declared within the module
- A concatenation of any of the above

Named port connections in instances are not supported for modules whose definitions contain a mix of port expressions and aliases. Use positional port connections to instantiate such modules.

Port expressions and aliases are supported in the RTL elaboration flow (`read_hdl` + `elaborate`) and in the Structural Verilog flow (`read_netlist`).

Example 3-1 Handling of module with port expressions and aliases in RTL elaboration flow

Input HDL

```
typedef struct packed {bit w; bit w1; bit w2;} my_struct;

module test(a, {b, c[1]}, .x({d, e}), .y(f), .z({g[2:1]}));
input my_struct a;
input my_struct b;
input [1:0]c;
input d, e;
output my_struct f;
output [2:1]g;
assign f = b;
assign g = c;
endmodule
```

Genus output

```
module test(\a[w2] , \a[w1] , \a[w] ,  {\b[w] , \b[w1] , \b[w2] , c[1]},
  .x( {d, e} ),  .y( {\f[w] , \f[w1] , \f[w2] } ),  .z(g[2:1]));
    input [1:0] c;
    input \b[w2] , \b[w1] , \b[w] , e, d;
    output \f[w2] , \f[w1] , \f[w];
    output [2:1] g;
    wire [1:0] c;
    wire \b[w2] , \b[w1] , \b[w] , e, d;
    wire \f[w2] , \f[w1] , \f[w];
    wire [2:1] g;
    input \a[w2] , \a[w1] , \a[w];
    wire \a[w2] , \a[w1] , \a[w];
    assign g[2] = c[1];
    assign \f[w2]  = \b[w2];
    assign \f[w1]  = \b[w1];
    assign \f[w]  = \b[w];
endmodule
```

Example 3-2 Handling of module with port expressions and aliases in Structural Verilog flow

Input HDL

```
module test(.x(a), b[2], c[2:3], .y({d, e[2]}));
    input  [2:0]a;
    input  [2:0]b;
    input  [2:0]d;
    output [2:4]c;
    output [2:0]e;
    assign c[3] = b[2];
    assign e[2] = a[1];
endmodule
```

Genus output

```
module test( .x(a), b[2], c[2:3], .y( {d, e[2]} ));
    input [2:0] a, b, d;
    output [2:4] c;
    output [2:0] e;
    wire [2:0] a, b, d;
    wire [2:4] c;
    wire [2:0] e;
    assign e[2] = a[1];
    assign c[3] = b[2];
endmodule
```

4

Reading Designs in Simulation Environment

Introduction

This feature in Genus enables you to use specification files (referred to as the simulation optionfile or simply optionfile) to read all the RTL files for a Verilog or System Verilog design. This feature does not support a mixed language environment or VHDL. The optionfile is an ASCII file consisting of the file list with options that are available as old-style library specification options in Cadence's IUS (Verilog/VHDL simulator) tool. These options enforce certain rules to identify the names and locations for source files used for searching Verilog definitions for the top modules and other modules in the top module hierarchy.

If you specify the optionfile with the `-f` option of the `read_hdl` command, Genus automatically identifies the file locations and parses the definitions for the modules from the files, as per certain rules. The rules used to identify the file locations and module definitions are discussed in Use Model.

Command Description

```
read_hdl -f optionfile
```

The *`optionfile`* contains a single command line with multiple command options. The command options in the optionfile correspond to a subset of command options available with an old-style library specification scheme in IUS NCVerilog. Table 1 specifies the various switches available for use in the optionfile.

An example of an optionfile is:

```
top.v -v file1.v -y libpath1 -v file2.v -v file3.v -y libpath2/subpath1 -v design1.v
     -exclude {file5.v} -v file4.v -f optfile1 +incdir+dir1/foo+  +libext+.extv+
     +define+mac+ -recursive
```

C-style comments (single line and multi-line) are allowed in the optionfile.

Table -1 Command options available in optionfile

Switch	Description
`No switch`	Specifies the name or path of a Verilog file containing the definitions of potential top module(s). For example: top.v Files specified in optionfile without a switch are called *design files*. The top module must come from a design file.
`-v`	Specifies the name or path of a Verilog file, which has definitions of HDL library modules. Verilog files specified with –v option are called *library files*. They are read only on need basis. If the module instance hierarchy under top-module refers to a module, which is undefined in the design file. The library file containing the definition of such a module, is read to complete the definition. For example: `file1.v`, `file2.v`, `design1.v`
`-y`	Specifies the name or path of a directory that has Verilog files, containing definitions of modules. Paths specified with "-y" options are called *library directories.* A file in a library directory, is considered as a potential location of module named `foo`, if the name of the file is `foo.<ext>`. Where `<ext>` is one of the file extensions listed with `+libext+` switch. For example: `libpath1, libpath2/subpath1`
`-yd`	Specifies the name or path of a directory that has Verilog Library files.
`+define+`	Specifies the definition(s) of Verilog macros. For example: Use `+define+mac1+mac2=21+` to specify `` `define mac1 `` and `` `define mac2 21 ``
`+incdir+`	Specifies the paths to directories, which have include files. For example: `+incdir+dir1/foo` is to specify directory `dir1/foo/` as a possible location for searching `include files.
`+libext+`	Specifies the filename extensions for the Verilog files under library directories. This option can be used only once for each `read_hdl` command. For example: `+libext+.vlog+` is to specify filename extension `.vlog`
`-librescan`	Restarts the search from the beginning of the command file for each module that is searched. This command should appear only once in the command file and affects globally. Its exact place makes no difference.
`-recursive`	Enables recursive searching for design files in all the `-y` directories. This options needs to be set only once in the optionfile.

`-exclude`	If any sub-directory is specified in the exclude file list, all the files and subdirectories in the list are excluded from the parsing process. Syntax: `-exclude {`*`tcl-list of files and sub-directories to be excluded`*`}` Example: `-y dir1 -exclude {file1 subdir1} -y dir2 -exclude {file2 subdir2}`
`-f`	Specifies another optionfile. For example: `-f optfile1`.

Restrictions on the usage of `read_hdl -f`

1. `read_hdl -f` ignores the `hdl_language` attribute setting. The language option `v2001` is considered to be true for `read_hdl` even if not used explicitly. But you can change it to `sv` using `read_hdl -sv -f` *`filename`*
2. You cannot specify a design file on the command line when you use the `-f` option with the `read_hdl` command. The design file must be specified in the optionfile.

   Not correct: `read_hdl top.v -f test`

   Correct: `read_hdl -f test, where test contains top.v.`

   Also, `-f` option is nestable. That means the command options in an optionfile can further contain another `-f` option.

   Note: But, in case the same optionfile is used again for nesting, it implies a recursive loop. This situation is not handled right now. Therefore, when the `-f` option is used, take care not to specify a nested `-f` option with the same optionfile.
3. To read more than one optionfile
   a. Either include the second file in the first optionfile with `-f` switch.

      For example, `read_hdl -f top -f top2` must be replaced with `read_hdl -f top`, where `top` now contains `-f top2`.

   b. Or give another `read_hdl -f` command on the Genus prompt (that is, multiple `read_hdl -f` commands are supported).

      For example,
      ```
      read_hdl -f top
      read_hdl-f top2
      ```
4. You cannot combine the `-f` option with the `-vhdl` option. This feature only works for Verilog designs.
5. The c style single line comment (\\) should either be preceded by a blank space or a newline, or it should be at the start of the line or the file.

Use Model

Genus maintains a collection of modules, whose definitions have been read. This collection is called *module pool*. When using the `-f` option, the module pool is populated with the modules that are defined in the design files. As of now, the files that are either library files or part of library directories are not used to populate the module pool.

Next, when you specify a top-module with the elaborate command, Genus initiates a top-down traversal of the module instantiation hierarchy under the top-module. While traversing, when an instance of a hierarchical module is encountered and if it is already a part of the module-pool, then its pre-existing definition is used to define its instance. If it is not in the module pool, then Genus searches its definition according to the following rules:

- The library file options (`-v` or `-y`) are considered to be ordered (from left to right) as per the order of occurrence in the optionfile.
- Genus elaboration tries to find the definitions of the undefined modules using the order of the library file/directory options and depending on the source of the parent module, in which the undefined module's instance is encountered first during elaboration.
    - If the parent module definition comes from a design file, then the search begins from the leftmost library file or directory among the library options.
        - If the leftmost option in the optionfile is `-v`, then the tool checks the corresponding library file for the definition of next undefined module in its list of undefined modules.
        - If the leftmost switch is `-y`, then the tool searches for a file with a name `<modulename>.<ext>` in the directory path referred in the `-y` switch. Here the extension name `<ext>` is taken from the `libext` option. If the corresponding file (in library directory) does not exist or the definition is not found in the leftmost library option, then the next leftmost option is considered and so on until the definition of the undefined module is found.
    - If the parent module definition was obtained from a file in a library directory, then the starting position of the search for the undefined module is different. The search does not start from the leftmost library option. It starts from the library directory where parent module's definition was found.
    - The search continues to the next option on the right, as long as a definition of the undefined module is obtained. On reaching the rightmost library option, the search wraps around to start from the leftmost library option and ends at the library option that is immediately to the left of the starting point.

- During above search, if Genus finds the definition of a module it was looking for in a file, it adds the definition of all modules in that file, to the module pool. If it encounters a repeat definition of a module, which is already defined in the module pool, it ignores the new definition and retains the older definition for that module.
- The tool searches for the include files and instances under the given top module in top-down breadth-first manner. If module `top` has hierarchical instances of `undef1`, `undef2`, ..., `undef8` (say), the search for definitions of `undef1`, `undef2`, ..., `undef8` is done in parallel as described earlier. This means that the left-most library file (from the starting point) containing any one or more of `undef`* is read and its contents are made part of the module pool. Then the remaining modules (as yet not in the module pool) are searched.
- Search for include files (referenced Verilog files with the `` `define `` directive) is done in the following order of directory locations:
  - ❑ Current work directory.
  - ❑ Paths that are part of `+incdir+` option.
  - ❑ The directory containing the file which has `‘include` directive.
  - ❑ Paths that are part of the `-y` option.
  - ❑ Paths that are part of the `init_hdl_search_path` attribute.
- The parser in Genus honors macros in the order in which they are encountered. At the start (before reading design files), the initialization step defines the macros that are given with the `+define+` switch in the *`optionfile`* or with `-define` option of the `read_hdl` command. Then, the RTL files (design files and need-based library files) are read in the order in which the hierarchical instantiations are encountered.
- $-prefixed string references are allowed in the optionfile. String substitution is used for the $-prefixed strings. The $-prefixed string is viewed as the value of a tcl variable or a system environment variable and the value of this variable is substituted for its reference in the optionfile. There are two allowed formats for the $-prefixed references:

  `${var}` or `$var` can be used for tcl and environment variables.
  `$::env(var)` can be used for environment variables only.

  If the variable is undefined, Genus stops after issuing the error message HDL-18.

  Note: If a TCL variable is defined with the same name as an environment variable, then the TCL variable will take precedence.

Examples

Sample contents of files used in the following examples:

- top.v

```
module top (…);
   foo1 u1 (…);
   foo2 u2 (…);
endmodule // top

module foo1 (…);
   sub1 u3(…);
endmodule // foo1
```

- abc.v

```
`ifndef macro1
   module foo2 (…);
     sub3 u4(…);
endmodule // foo2
`endif

`define macro2
```

- def.v

```
module leaf1 (…);
   nand2 u4(…);
endmodule // leaf1
```

- dir1/sub_dir1/foo2.v

```
`ifdef macro2
module foo2 (…);
    sub2 u4(…);
endmodule // foo2
`endif
```

- dir1/sub1.v

```
module sub1(…);
    leaf1 u7(…);
endmodule //sub1
```

- dir1/sub2.v

```
module sub2(…)
   leaf1 u6(…);
endmodule // sub2
```

- dir2/foo2.v

```
`ifdef macro2
   module foo2 (…);
      sub2 u4(…);
   endmodule // foo2
`endif
```

- dir2/sub1.v

  ```
  module sub1(…);
     leaf1 u7(…);
  endmodule //sub1
  ```

- dir2/leaf1.v

  ```
  module leaf1 (…);
     nor2 u5(…);
  endmodule // leaf1
  ```

Example 4-1 Resolution of module definitions

```
read_hdl -f mytest
```

where `mytest` is the optionfile

- If the content of the `mytest` file is:

  ```
  top.v -v abc.v -v def.v -y dir1 +libext+.v+ -y dir2
  +define+macro1+
  ```

  In the file `mytest`, when the tool elaborates module `top`, it finds unresolved instances of `foo2` and `sub1`. To find the definition of `foo2`, the tool first reads the file `abc.v.` Since `macro1` has been defined in the optionfile, the tool does not use this definition of `foo2`, but it does set `macro2` as defined.

  Now, we need the definition of foo2. The file dir1/sub_dir1/foo2.v is not read since library directories are not searched recursively. Next, the tool reads file `dir2/foo2.v` to find the definition of `foo2`.The definition of `foo2` has an unresolved instance `sub2`.Now there are two unresolved instances `sub1` and `sub2`.

  The tool finds the definitions of `sub1` and `sub2` in the files `dir1/sub1.v` and `dir1/sub2.v`, respectively. Both modules `sub1` and `sub2` contain an unresolved instance of `leaf1`. The tool gets the definition of `leaf1` from `dir2/leaf1.v,` since the search for `leaf1` continues from `dir2.`

- If `-librescan` is added to the `mytest` file as follows:

  ```
  top.v -v abc.v -v def.v -y dir1 +libext+.v+ -y dir2
  +define+macro1+ -librescan
  ```

  The search for the definition of `sub2` and `leaf1` will start from the beginning of the optionfile, that is, `abc.v`.

Example 4-2 Usage of `exclude` and `recursive` options

```
read_hdl -f newtest
```

where `newtest` is the optionfile

The content of the `newtest` file is:

```
top.v -v abc.v -v def.v -recursive -y dir1 -exclude {sub1.v} +libext+.v+ -y dir2
     +define+macro1+
```

When Genus elaborates the module `top`, it finds unresolved instances of `foo2` and `sub1`. To find the definition of `foo2`, it first reads the file `abc.v`. Since `macro1` has been defined in the optionfile, this definition of `foo2` is not used, but it does set `macro2` as defined.

Now it reads file `dir1/sub_dir1/foo2.v` to find the definition of `foo2`. Since the optionfile contains `-recursive` option, the subdirectories (for example, `sub_dir1`) are also searched for module definitions (where it is found). The definition of `foo2` has an unresolved instance `sub2`. Now there are two unresolved instances `sub1` and `sub2`.

The file `dir1/sub1.v` will not be read since it is mentioned in the `exclude` list. The definitions of `sub1` and `sub2` are found in the files `dir2/sub1.v` and `dir1/sub2.v` respectively. Both modules, `sub1` and `sub2,` contain an unresolved instance of `leaf1`. The tool gets the definition of `leaf1` from `dir2/leaf1.v`, since the search for `leaf1` continues from `dir2`.

Example 4-3 Usage of variables

If following settings are done on the Genus command line using the `set` command:

```
    set env(PATH1) dir1
    set PATH2 $::env(PATH1)
    set PATH3 sub_dir1
    set env(PATH4) dir2
```

The optionfile may use these settings as the following:

```
top.v -v foo.v -y ${PATH2} -exclude {bar.v $PATH3} -y $::env(PATH4) +libext+.v -
recursive
```

Example 4-4 Multiple read_hdl -f commands

The following example shows the usage of multiple optionfiles along with the usage of -`define` option with `read_hdl`:

```
    read_hdl -f mytest1
    read_hdl -define macro2 example2
    elab top
```

The contents of `mytest1` file are:

```
    def.v
```

The contents of `example2` file are:

```
    top.v +libext+.v -y dir2 -y dir1
```

Here, `top.v` contains instances `foo1` and `foo2`. `foo1` will get resolved from the `top.v` file. `foo1` has `sub1` as an instance which will get resolved in `dir2/sub1.v`. `sub1` has `leaf1` as an instance which will get resolved by `def.v`.
The `foo2` instance in `top` will get resolved from `dir2/foo2.v`. `foo2` has `sub2` as an instance which will get resolved in `dir1/sub2.v`. `sub2` has instance `leaf1` which will get resolved from `def.v`.

Example 4-5 Usage of multiple read_hdl with read_hdl -f

The following example shows the usage of multiple `read_hdl` commands

```
    read_hdl def.v
    read_hdl -f example2
    elab top
```

The contents of `example2` file are:

```
    top.v +libext+.v +define+macro2+ -y dir2 -y dir1
```

The file `top.v` contains instances `foo1` and `foo2`. `foo1` will get resolved from the same t`op.v` file. `foo1` has instance `sub1` which will get resolved using `dir2/sub1.v. sub1` has `leaf1` instance which will get resolved by `def.v`.

Now `foo2` instance in `top` will get resolved from `dir2/foo2.v`. `foo2` contains `sub2` which will be resolved in `dir1/sub2.v`. `sub2` has instance `leaf1` which will get resolved in `def.v`.

5

Bottom-up Flow

Overview

To manage the complexity of modern designs, EDA designers often use modular approach for writing the RTLs, that is, by breaking the design into smaller blocks. The design or synthesis of these blocks is often given to separate teams. During synthesis of such designs, some of these blocks are synthesized and re-defined iteratively to improve QOR optimizations and to enhance their functionality. The synthesis flow needs to handle elaboration of multiple pieces of such a design with the final synthesized design, even when some of the blocks are still in the initial RTL stages, while some of the blocks are pre-elaborated or pre-synthesized. It makes the flow efficient, if the pre-synthesized or pre-elaborated blocks are not re-synthesized or re-elaborated while synthesizing the higher-level blocks. Thus, a pre-synthesized block can be used without any changes during assembling of the higher-level blocks, thereby saving time and effort for re-synthesis.

To enable this, the synthesis flow is able to read the pre-elaborated or pre-synthesized blocks in various convenient formats: as gate-level Verilog netlist, as macro-cells in liberty file, as ILM (Interface Logic Modules) modules, etc. As multiple iterations are made to assemble a large design, different pre-synthesized blocks get pulled at different (and sometimes multiple) stages based on their instantiations at different levels of hierarchies.

Issues with the bottom-up approach

- **Module-name matching** – When a sub-block is instantiated in the RTL of a parent block, the synthesis process identifies the sub-block through a name. In case of non-parameterized sub-blocks, the search for a sub-block amongst other pre-synthesized blocks is simple: the name of module in instantiation matches the name of the already synthesized sub-block. But in case of parameterized sub-blocks, the pre-synthesized sub-blocks can have different synthesized names based on the different sets of parameters passed to it during synthesis.
- **Port matching –** If a sub-block has complex ports (e.g structure port or multidimensional array port), then in its pre-synthesized netlist, these ports may be represented by multiple ports, one each for each basic leaf field of the complex port. Synthesis process needs to

be able to match each complex port name in instantiation with the corresponding multiple ports in the pre-synthesized netlist.

Flavors of Bottom-up Flows

Genus provides multiple flavors of the bottom up flows to cater to the various styles of modular designing and the preference of the module interfaces chosen for pre-synthesized blocks. Some of the bottom-up approaches available in Genus are:

- Module wrapper flow
- Sub-block Macro Cell Based Flow

Module wrapper flow

In this flow, the pre-synthesized (or pre-elaborated) sub-block is represented as a Verilog gate-level netlist, but with an extra wrapper module (in netlist) over the top-level module of the pre-synthesized sub-block. This information is required while elaborating the higher-level blocks. The features of the sub-block's pre-synthesized netlist are as follows:

- The sub-block module name in netlist is modified from its expected name.
- A wrapper SV module is included in the netlist with the same name as the sub-block, Its body consists of one instance of the pre-synthesized sub-block module using the modified name. This is done through the `write_sv_wrapper` command.
- The ports of the wrapper SV module match, in type and complexity with the original definition of the sub-block in RTL.

This extra top layer or the wrapper module matches in name and ports with the RTL definition of the sub-block; allowing Genus to easily stitch instances of the sub-block present in the RTL of a higher level block.

A detailed example:

Subblock RTL

```
typedef struct { logic f1; logic f2; } mystruct;
module subblock (input mystruct in1, output mystruct out1);
assign out1.f1 = ~in1.f1;
assign out1.f2 = ~in1.f2;
endmodule
```

After synthesis, the netlist of subblock1, will be represented as follows:

Wrapper module in file `wrapper.sv`

```
typedef struct { logic f1; logic f2; } mystruct;
module subblock (input mystruct in1, output mystruct out1);
subblock_mod u0 (\.in1[f1] (in1.f1), \in1[f2], (in1.f2), \out1[f1] (out1.f1),
\out1[f2] (out1.f2));
endmodule
```

Netlist module for sub-block obtained from synthesis (with modified block name)

Netlist in subblock_mod.v

```
module subblock_mod (input \in1[f1] , \in1[f2] , output \out1[f1] ,
                    \out1[f2] );
    input \in1[f2] , \in1[f1] ;
    output \out1[f2] , \out1[f1] ;
    wire \in1[f2] , \in1[f1] ;
    wire \out1[f2] , \out1[f1] ;
    inv1 g3(.A (\in1[f2] ), .Y (\out1[f2] ));
    inv1 g4(.A (\in1[f1] ), .Y (\out1[f1] ));
endmodule
```

Higher-level block in file `top.sv`

```
typedef struct { logic f1; logic f2; } mystruct;
module top (input mystruct in1, output mystruct out1);
subblock u0 (.in1(in1), .out1(out1));
endmodule
```

Elaborating the higher-level block:

```
set_db library <library_name>
read_hdl -language sv top.v
read_hdl -language sv wrapper.sv
read_hdl -netlist subblock_mod.v
elaborate top
```

The ports in an instance of the sub-block inside the top module can now be matched easily with the ports of the wrapper module of the sub-block. After elaboration, the design will have an extra layer of module corresponding to the wrapper sub-block.

Note: This extra layer can be eliminated using ungroup command.

To use the sub-block netlist along with its wrapper, you need to set the attribute `hdl_sv_module_wrapper` to `true` before elaboration. This signifies that you intend to make a wrapper module. Genus will take care to carry forward all the information related to the sub-block and later, this information will be used to create the wrapper. After elaboration or synthesis, you need to write out the wrapper using the `write_sv_wrapper` command.

Recommended Flow

Elaborating and synthesizing the sub-module

```
set_db library tutorial.lib
set_db hdl_sv_module_wrapper true
read_hdl subblock.v
elab
syn_gen
syn_map
...
write_sv_wrapper -module_name subblock_mod -rename_module >
               submodule_wrapper.v
write_hdl > subblock_mod.v
```

Note: Redirecting the output of the write_hdl command is mandatory as this file will be later used for stitching with the top module.

Elaborating and synthesizing the top module along with the wrapper module

```
set_db library tutorial.lib
set_db hdl_error_on_blackbox true
read_hdl -netlist subblock_mode.v
read_hdl -language sv submodule_wrapper.v top.v
elab
syn_gen
....
write_hdl > top_elaborated.v
```

It is recommended to set the `hdl_error_on_blackbox` attribute to `true` during elaboration of the parent blocks. It checks whether the stitching went well or not.

The `write_sv_wrapper` command renames the elaborated `subblock` to `subblock_mod` and also writes out the wrapper module to `stdout`. To update the name of the sub-module in the Genus design hierarchy, you need to use the `-rename_module` option of the `write_sv_wrapper` command.

Sub-block Macro Cell Based Flow

Designers, sometimes, want to represent a previously synthesized sub-block as a macro-cell (namely, a timing model) in liberty format. Then they want to elaborate the higher level blocks that contain instances (with possibly complex ports) of the sub-block, without using the RTL of the sub-block; but with the expectation that the instances of the sub-block will be replaced with the corresponding liberty based macro-cell. We refer to this style of bottom-up elaboration flow as "Sub-block macro cell based bottom-up flow".

This document explains how to stitch these liberty cells with a higher level module in Genus.

The following example illustrates how Genus elaboration supports the aforementioned stitching of macro-cell instances (in place of sub-block instances) with higher level blocks in the default flow.

A detailed example

Consider a Verilog block named `block1`:

```
package pack ;
    typedef struct packed {
    bit [1:0] f1;
    bit [2:0] f2;
    } mystruct;
endpackage

import pack::*;

module block1 (input mystruct a, output mystruct b);
    assign b = ~a;
endmodule
```

After synthesis, the netlist of `block1` will have fragmented ports as follows:

```
module block1(input [1:0] \a[f1] , input [2:0] \a[f2] , output [1:0] \b[f1] ,
       output [2:0] \b[f2] );
…
endmodule
```

This will be represented as a timing-model or macro-cell having bus ports in liberty as follows:

```
cell(block1) {
    bus (a[f1] ){
    …
    pin (a[f1][1] ) {
    direction : input
…
}

pin (a[f1][0] ) {
    direction : input ;
…
}
```

```
bus (a[f2] ){
    …
    pin (a[f2][2] ) {
    direction : input ;
    …
}
pin (a[f2][1] ) {
    direction : input ;
    …
}
…
```

This is the higher level block '`top`' that instantiates `block1`:

```
package pack ;
typedef struct packed{
    bit [1:0] f1;
    bit [2:0] f2;} mystruct;
endpackage
import pack::*;
module top(input mystruct y, output mystruct z);
    block1 b1 (.a(y),.b(z));
endmodule
```

You can now elaborate the `top` module linking the instance of `block1` with the library cell `block1`, using the following script:

```
set_db library libname.lib //The library file containing the macro-cell for block1.
read_hdl -sv top.v //The Verilog RTL file containing the definition of package pack
and module top.
elaborate
......
```

Following is the netlist generated using the `write_hdl` command:

```
module top(\y[f2] , \y[f1] , \z[f2] , \z[f1] ) ;
    input [2:0] \y[f2] ;
    input [1:0] \y[f1] ;
    output [2:0] \z[f2] ;
    output [1:0] \z[f1] ;
    wire [2:0] \y[f2] ;
    wire [1:0] \y[f1] ;
    wire [2:0] \z[f2] ;
wire [1:0] \z[f1] ;
block1 b1(.\a[f1]  (\y[f1] ), .\a[f2]  (\y[f2] ), .\b[f1]  (\z[f1] ),
            .\b[f2]  (\z[f2] ));
endmodule
```

Note: The instance of the block in its parent module(s) should be created with named port association and not positional port association. Hence, in the above example, the instance `b1` of `block1` should not be written as `block1 b1(y,z)`.

6

Mapping to Multibit Cells

Overview

A multibit cell represents a group of cells with identical functionality. Multibit cells generally have lower power, similar or better area, and are easier to use for place and route.

Genus recognizes the following style of components for multibit merging:

- Flops (non-scan and scan) with one or more of the following shared input pins:

  flop clock, async_set, async_reset, sync_set, sync_reset, sync_enable

- Latches with one or more shared control pins:

  latch gate/ enable, async_set, async_reset

- Three-state cells that share the enable pins
- Combinatorial cells (muxes, inverters, nand, nor, xor, xnor) that share all pins, which are not bundled in the Liberty description.
- State retention (SRPG) cells that share the same retention control pin(s)

Figure 6-1 on page 77 highlights the tasks you need add to the generic top-down synthesis flow to perform multibit cell mapping. Multibit cell mapping occurs during incremental optimization.

Genus supports variable bit-widths. If the library has multibit library cells with different bit widths, all sizes will be considered during multibit cell inferencing. The tool starts merging using largest bit width first.

By default, multibit mapping is QoR- driven.

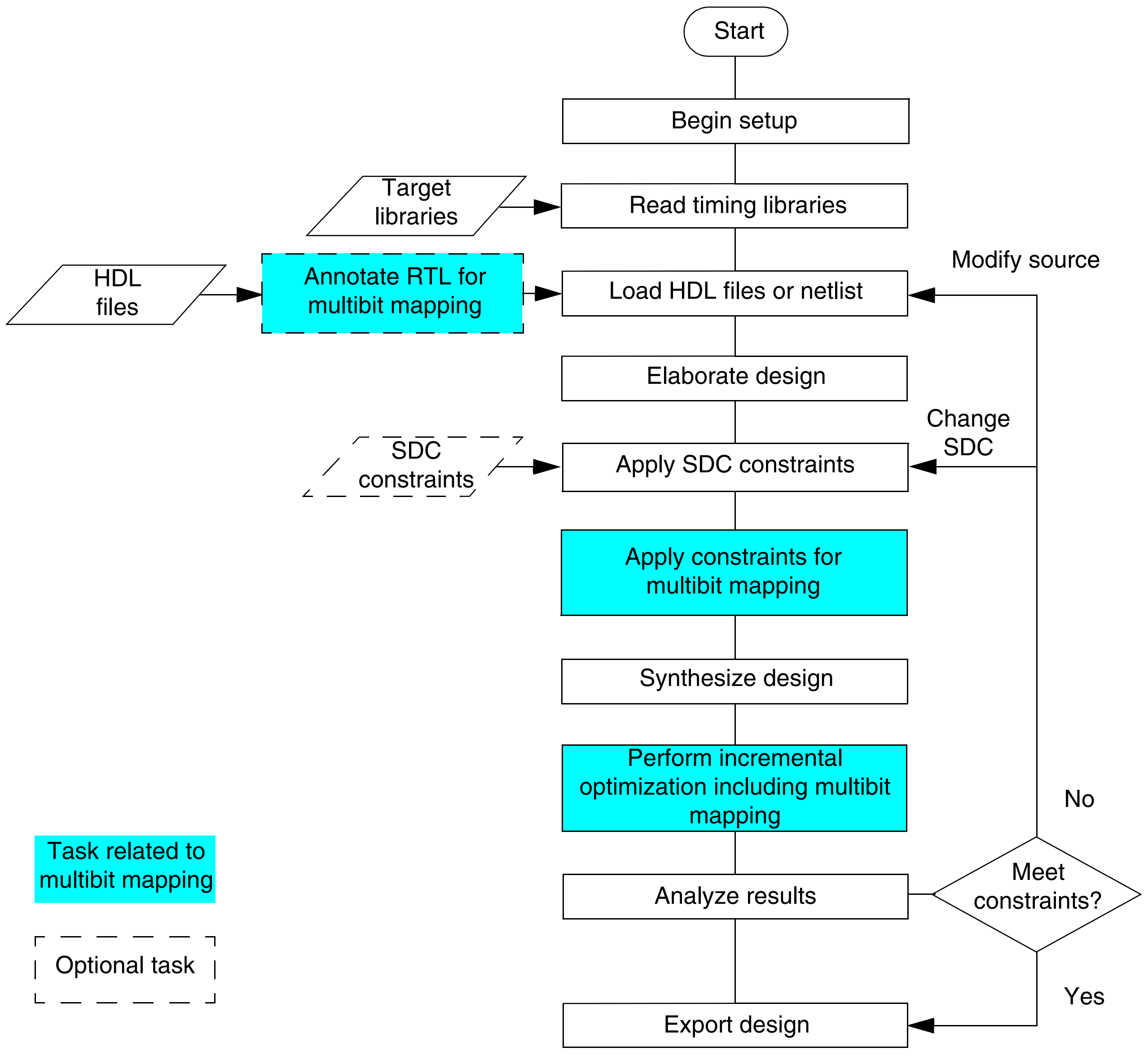
Refer to Multibit Cell Requirements in the *Genus Library Guide* for more information on the Liberty requirements for mapping to multibit cells and multibit SRPG cells.

Multibit flops can be replaced with either parallel or serial multibit scan cells in the DFT flow. For more information, refer to Mapping to Multibit Scan Cells in the *Genus Design for Test Guide.*

In addition, to map single SRPG instances to multibit SRPG cells in a CPF flow, the single SRPG instances must

- Belong to the same power domain.
- Have been inserted based on the same CPF `create_state_retention_rule` command.

Figure 6-1 Multibit Cell Mapping Flow



Tasks

The following tasks specify the constraints related to multibit cell mapping:

- Selecting Candidates for Multibit Cell Mapping on page 78
- Controlling Multibit Cell Mapping on page 79
- Controlling Naming of Multibit Instances on page 81
- Preserving Inferred Multibit Cells on page 82

Selecting Candidates for Multibit Cell Mapping

➤ To select (or exclude) candidates for (from) multibit cell mapping,

❑ Add the following pragmas in the RTL:

```
// cadence merge_multibit
// cadence dont_merge_multibit
```

For example:

```
always // cadence merge_multibit "q"
    @( posedge clk)
        q <= d;
```

Note: Pragmas can be used for combinational cells as well.

❑ Set the following attributes on instances in the netlist:

```
set_db  instance .merge_multibit {true | false}
set_db  instance .dont_merge_multibit {true | false}
```

Note: The pragmas in the RTL are also translated to these attributes during elaboration.

The following table shows when the tool considers multibit merging for the instance.

dont_merge_multibit	merge_multibit	Multibit merging is
false	false	attempted if `use_multibit_cells` is set to `true`.
false	true	attempted.
true	false	not attempted.
true	true	attempted. The `dont_merge_multibit` attribute setting is ignored.

➤ For automatic selection of candidates for multibit cell mapping, set the `use_multibit_cells` root attribute (default: `false`) prior to synthesis:

```
set_db use_multibit_cells true
```

All single-bit cell candidates are replaced with multibit cells during incremental optimization.

➤ To enable or disable the multibit mapping of *all combinational* cells, use the `use_multibit_combo_cells` root attribute (default: `false`).

➤ To enable or disable the multibit mapping of *all sequential and tristate* cells, use the `use_multibit_seq_and_tristate_cells` root attribute (default: `false`).

Controlling Multibit Cell Mapping

Genus provides several levels of controls over multibit mapping.

➤ To let the tool automatically select a suitable multibit cell in case of predefined multibit cell inferencing, set the `bank_based_multibit_inferencing` root attribute (default: `false`).

Design-Level Control

The setting of the `use_multibit_cells` root attribute, enables or disables multibit mapping for *all* candidates. This attribute controls both logical and physical-aware multibit mapping.

If the design is not placed, logical multibit mapping will occur. If the design is placed, physical-aware multibit mapping occurs. Depending on where you set the attribute in the physical flow you can control whether logical or physical-aware multibit mapping is performed.

➤ To enable or disable the multibit mapping of *all combinational* cells, use the `use_multibit_combo_cells` root attribute (default: `false`).

➤ To enable or disable the multibit mapping of *all sequential and tristate* cells, use the `use_multibit_seq_and_tristate_cells` root attribute (default: `false`).

By default, these attributes inherit the last setting of the `use_multibit_cells` attribute, but you can explicitly override this setting to enable or disable the mapping of combinational cells or sequential and tristate cells.

➤ To control whether only sequential cells with the same *basename* can be merged, use the `multibit_cells_from_different_busses` root attribute (default: `true`).

➤ To control whether asynchronous pins can be interchanged during multibit cell inferencing, use the `multibit_allow_async_phase_map` root attribute (default: `true`).

By default, phase-mapping of asynchronous pins is enabled during multibit inference by inverting the phase of async clear single-bit cells to async-preset multibit cells with inverted D input and Q output.

➤ To control which single-bit sequential instances can be merged into a multibit sequential instance during synthesis, use the `multibit_seqs_members_naming_style` root attribute. The attribute value can contain a list of naming styles of the single-bit instances for which the merging can be done.

For example, to combine single bit cells `a1_reg` and `a2_reg`, the following naming style must be part of the attribute value:

```
set_db multibit_seqs_members_naming_style "%s%d_reg"
```

Instance-Level Control

➤ To control mapping of specific instances, use the `merge_multibit` (default: `false`) or `dont_merge_multibit` (default: `false`) attributes. See the Table in Selecting Candidates for Multibit Cell Mapping for more information.

Note: The `dont_merge_multibit` attribute applies only to sequential and tristate instances.

User-Defined Multibit Cell Mapping

➤ To define a bank label for sequential instances to be considered for multibit mapping to the same bank of multibit instance, use the `map_to_multibit_bank_label` instance attribute (default: `default_bank`).

Flops belonging to different banks cannot be merged into a single multibit cell.

➤ To enable *predefined multibit cell inferencing* (MBCI) for the sequential instance and limit the multibit mapping of an instance to some specified multibit library cells, use the `map_to_multibit_register` instance attribute.

All the specified library cells must

- ❑ Have same bit width.

  A bit width mismatch implies no multibit cell inferencing for this bank-label.

- ❑ Be functionally swappable.

  A mismatch will not allow you to set the value for this attribute.

All instances with the same bank label (set through the `map_to_multibit_bank_label` attribute) and the same `map_to_multibit_register` attribute value, are considered for mapping to the same bank of multibit cells.

The following commands first define a label for a set of flops, then specify all library cells that can be used for multibit mapping of the set.

```
genus:root: 31> set_db [get_db design:* .insts *Outbus_reg\[1?\]]
.map_to_multibit_bank_label bank_1
  Setting attribute of instance 'Outbus_reg[10]': 'map_to_multibit_bank_label' =
bank_1
...
  Setting attribute of instance 'OutbusintCK_reg[19]': 'map_to_multibit_bank_label'
= bank_1
genus:root: 32> set_db [get_db design:* .insts *Outbus_reg\[1?\]]
.map_to_multibit_register HS65V_GSH_4DFPQX18
  Setting attribute of instance 'Outbus_reg[10]': 'map_to_multibit_register'
= /libraries/COR65/libcells/HS65V_4DFPQX18
...
  Setting attribute of instance 'Outbus_reg[19]': 'map_to_multibit_register' = /
libraries/COR65/libcells/HS65V_4DFPQX18
```

The next example defines two bank labels and specifies two multibit library cells.

```
set_db [get_db insts *q1_reg* ] .map_to_multibit_bank_label "bank1"
set_db [get_db insts *q2_reg* ] .map_to_multibit_bank_label "bank2"
set_db [get_db insts *q1_reg* ] .map_to_multibit_register "DFP4QX4"
set_db [get_db insts *q2_reg* ] .map_to_multibit_register "DFP4QX8"
```

Resulting netlist shows:

```
DFP4QX4 \q1_ reg[ 0_3] (. CP (clk), .D ({ a[0], a[1], a[2], a[3]}),
.Q ({ q1[0], q1[1], q1[2], q1[3]}));
DFP4QX8 \q2_ reg[ 0_3] (. CP (clk), .D ({ b[0], b[1], b[2], b[3]}), .Q ({ q2[0],
q2[1], q2[2], q2[3]}));
```

Considering Runtime, QoR, or Multibit Coverage

➤ To force merging of single-bit combinational instances into an appropriate multibit combinational instance independent of the impact on the QoR, to increase multibit *coverage*, set the `force_merge_combos_into_multibit_cells` root attribute (default: `false`).

➤ To force merging of single-bit sequential instances into an appropriate multibit sequential instance independent of the impact on the QoR, to increase multibit *coverage*, set the `force_merge_seqs_into_multibit_cells` root attribute (default: `false`).

Controlling Naming of Multibit Instances

➤ To control the naming of mapped multibit flops, latches and tristate instances, use the `multibit_seqs_instance_naming_style` root attribute (default: `concat`).

This attribute can have two values:

`auto` creates the name of the multibit instance based on busses of merged instances.

- ❑ If the instances belong to same bus, the name of the multibit instance starts with the bus name followed by the range of the merged bit-indices. For example:

  ```
  flop_bus_0_4
  ```

- ❑ If the instances belong to different busses, the name of the multibit instance contains each bus name followed by the range of corresponding bit-indices. For example:

  ```
  flop_bus_0_2_my_bus_5_7
  ```

`concat` creates the name of the multibit instance by concatenating the names of merged instances. For example:

```
flop_bus_0_flop_bus_1_flop_bus_2_flop bus_3_flop_bus_4
```

➤ To specify the prefix to be used to name the multibit instances, use the `multibit_prefix_string` root attribute (default: `CDN_MBIT_`).

The recommended value for the verification flow is `CDN_MBIT_`.

➤ To specify the separator string to be used in the name of the multibit instance created by concatenating the names of merged instances, use the `multibit_seqs_name_concat_string` root attribute (default: `_MB_`).

The recommended values for the verification flow are `_MB_` and `_mb_`.

This helps Conformal Logical Equivalence Checking to map state key points quickly.

Preserving Inferred Multibit Cells

➤ To specify whether to preserve multibit instances that are inferred during incremental optimization, use the `multibit_preserve_inferred_instances` root attribute (default: `false`).

If set to `true`, the `preserve` attribute for all multibit instances is set to `size_delete_ok`. This preserve setting prevents that any optimization step down the flow from breaking the multibit cells.

Controlling Naming of Single Bit Instances during Optimization

When the tool merges single bit instances into a multibit instances, it keeps track of the original names during the current session.

If the tool splits the multibit cell back into single bits in the same session as the merging occurred, it can reuse the original names.

If the merging and splitting occur in different sessions, the original names are not available. In that case, when the tool splits a multibit instance back into single-bit cells, it uses a delimiter with the multibit name to derive the individual names of the constituent single bit registers.

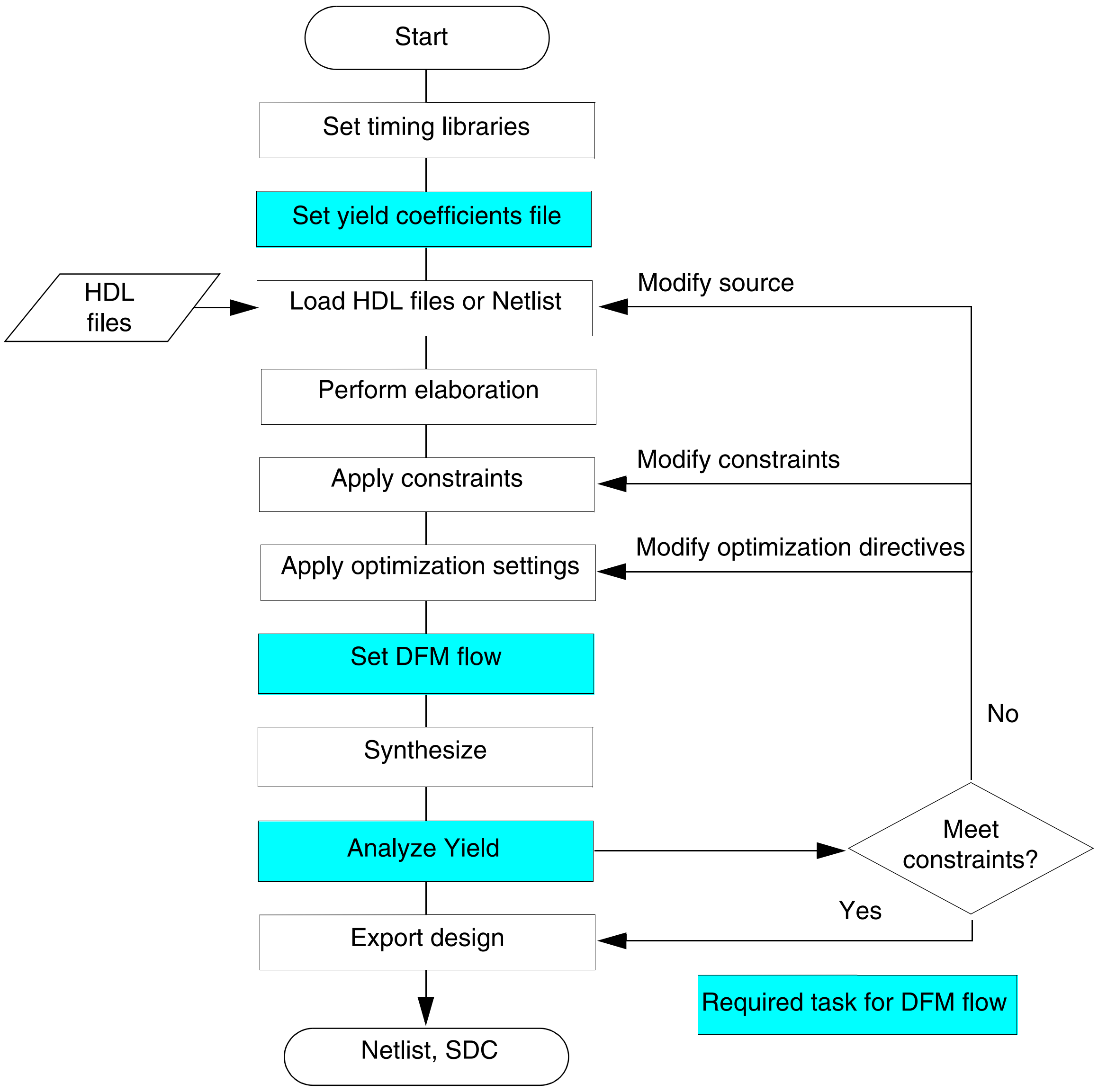
➤ To specify the string to be used as delimiter to derive the individual names of the constituent single bit registers from the multibit name, use the `multibit_split_string` root attribute (default: `_split_`).

7

Design For Manufacturing Flow

Overview

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



Tasks

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

Specifying the Yield Coefficients Information

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

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

```
genus:root: 41> read_dfm penny.dfm
```

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

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

```
genus:root: 45> read_dfm penny.dfm
genus:root: 46> read_dfm flame.dfm
```

Setting the DFM Flow

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

```
genus:root: 49> set_db optimize_yield true
```

The default value of this attribute is `false`.

Analyzing the Yield Information

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

```
genus:root: 53> report_yield

Instance      Cells   Cell Area            Cost     Yield %
-----------------------------------------------------------
cpu            470         659    1.600606e-05     99.9984
  alu1         248         283    7.606708e-06     99.9992
  pcount1       65          92    2.215669e-06     99.9998
  ireg1         33          88    1.629471e-06     99.9998
  accum1        33          88    1.629471e-06     99.9998
  decode1       50          67    1.568901e-06     99.9998
```

The command shows the defect-limited yield impact for library cell defects.

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

```
genus:root: 56> report_gates -yield

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

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

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

You must contact the fabrication facility to understand what percentage of failures are due to the cell based defect-limited yield effects. For example, `report_yield` may estimate that the yield is 90% while the true yield may only be 80% due to other effects that are not currently modeled. Cell failure rates are given as a simple failure rate per instance of the cell used.

To find the total yield for the design, use the `yield` attribute with the `get_db` command. The following example finds the yield for the design `test`:

```
genus:root: 62> get_db [get_db designs *test] .yield
0.999983594076
```

Recommended Flow

```
set_db library library_name
read_dfm coefficients_file
read_hdl filename
elaborate
read_sdc filename
set_db optimize_yield true
syn_gen
syn_map
report_yield
report_gates -yield
get_db yield [find . -design *]
```

8

Remapping a Synthesized Netlist

Overview

In cases where you need to migrate a design to a different technology but you only have the mapped netlist and constraints to start with, you will have to remap the netlist to the new target technology library. For example, a netlist mapped to a 12 tracks library, may need to be remapped to a 9 tracks library. This chapter describes the recommended flow.

Flow

1. Read in the old and new libraries.

```
set_db library {lib_12T.lib lib_9T.lib }
```

Note: You need to read in the old library to prevent unresolved references when reading in the netlist.

Important

The new technology library should have all the functional equivalents of the cells in the old library. Complex function cells or cells without functions will not be re-mapped. If you know the corresponding equivalents in the new library, you should relink the cells using the `change_link` command or edit the netlist prior to mapping.

2. Prevent mapping to the cells in the old library.

```
set_db [get_db library:lib_12T .lib_cells *] .avoid true
```

3. Read in the netlist that was mapped to the old library.

```
read_netlist netlist.v
```

4. Read in the SDC constraints.

```
read_sdc netlist.sdc
```

Note: You will need to update the SDC constraints in case they reference specific cells in the old library.

5. Unmap the design.

```
syn_generic
```

6. Map the design to the cells of the new library.

```
syn_map
```

7. Write out the new netlist.

```
write_hdl > remapped.v
```

9 Associating Dedicated Libraries with Different Blocks in the Design

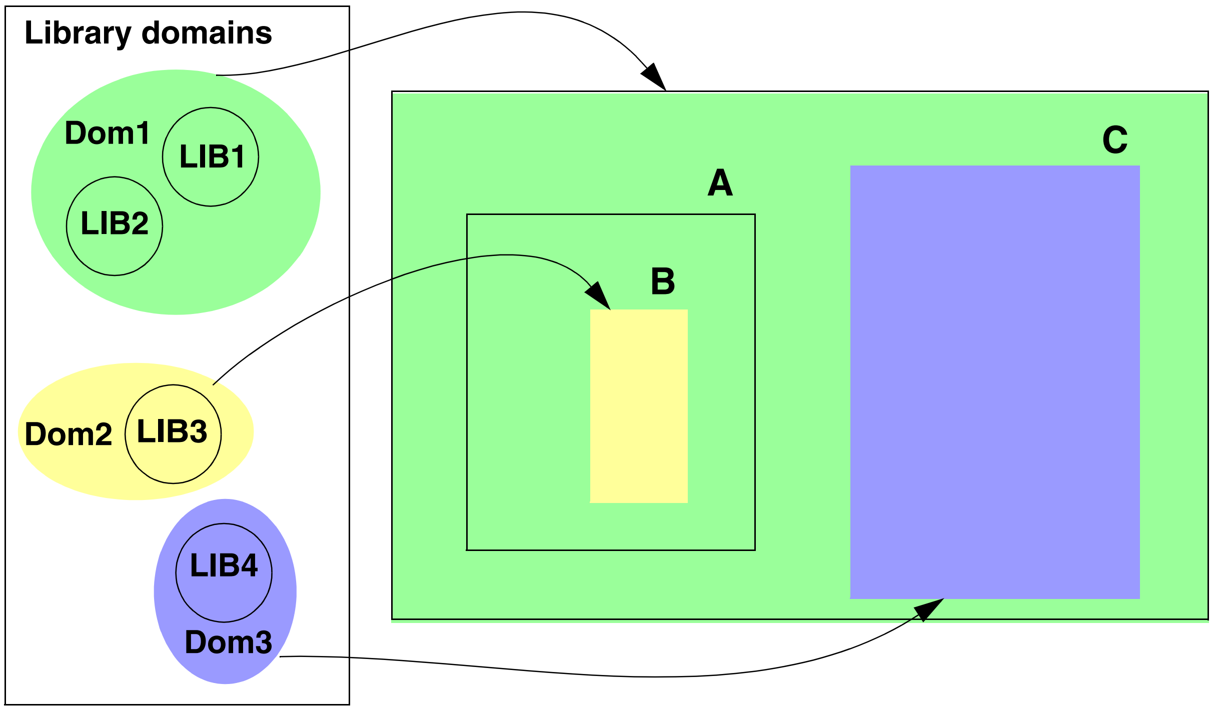
Overview

The generic synthesis flow uses the same libraries for the entire design. You must define library domains to

- **Associate Dedicated Libraries with Different Portions of the Design**

  The design in Figure 9-1 uses three different library sets. You want to use libraries `LIB1` and `LIB2` for the top-level and for block A, library `LIB3` for block B and library `LIB4` for block C.

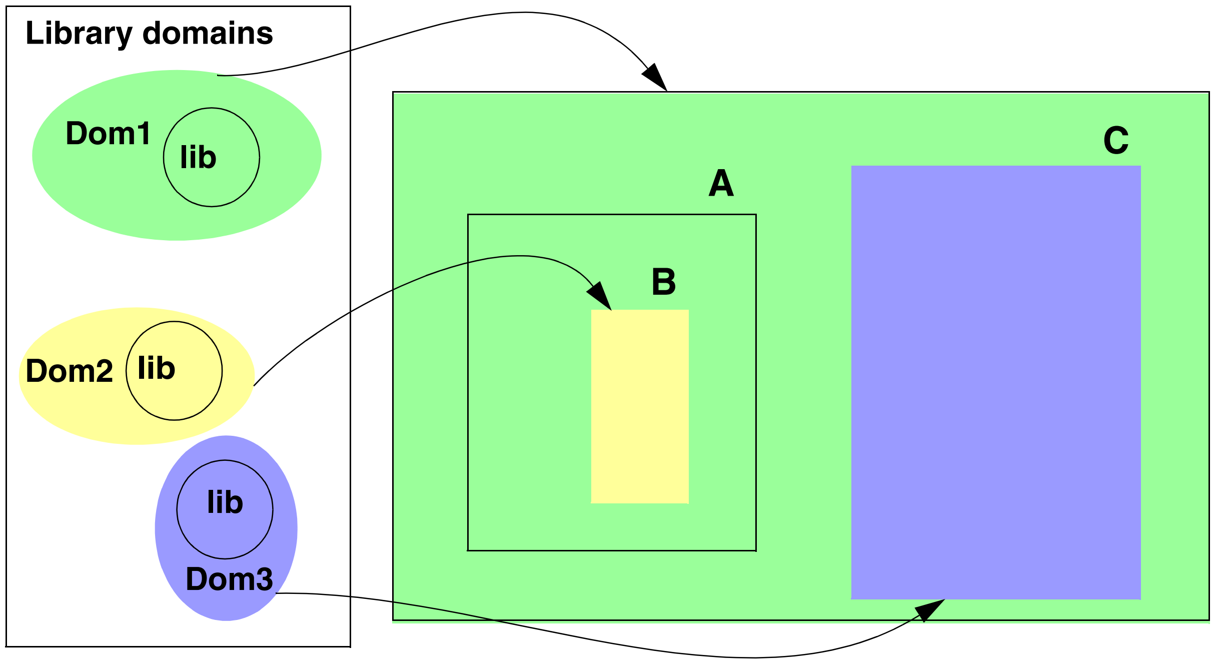
Figure 9-1 Use of Dedicated Libraries in Single Supply Voltage Design



- **Target Different Cells from Same Library for Different Portions of Design**

  The design in Figure 9-2 on page 97 uses the same library for the entire design, but you would like to use a limited set of cells to map block B and a different set of cells to map block C, and you allow the use of all cells for the top-level and block A.

Figure 9-2 Use of Targeted Cells in Single Supply Voltage Design



This chapter describes the top-down synthesis flow using multiple library domains. The flow is shown in Figure 9-3 on page 98. A script is shown in Example 9-1 on page 99.

Flow Steps describes the steps that are not part of the generic synthesis flow in detail.

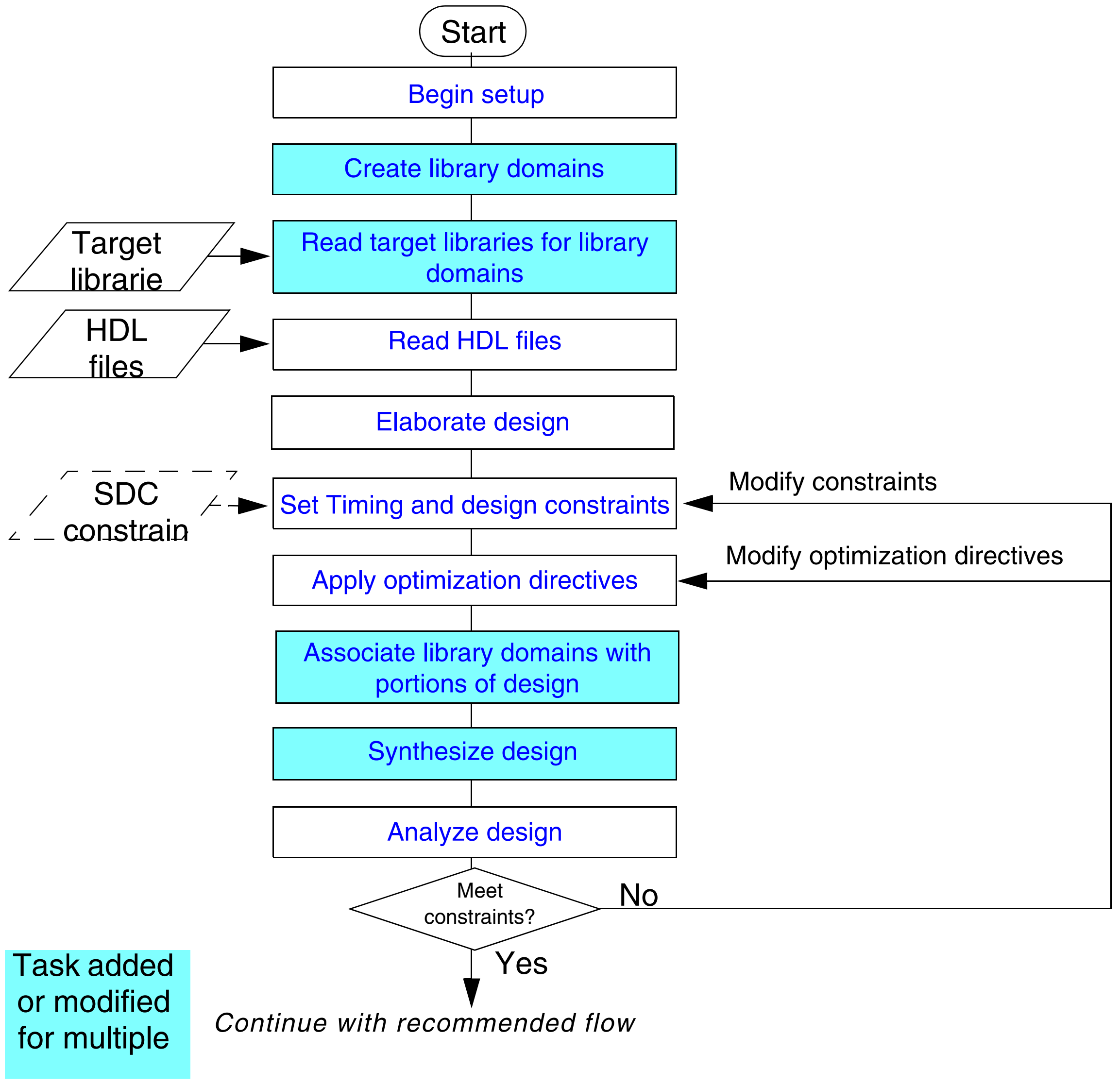
Library Domain Information in the Design Information Hierarchy shows where the library domain information is stored.

When using multiple library domains, you can also perform a what-if analysis to determine which configuration results in better timing. This analysis and some other tasks that are not part of the normal flow covered in Flow Steps are described in Additional Tasks.

Important

The flow described in this chapter assumes a *single* supply voltage design. Genus also uses library domains for *multiple* supply voltage designs.

Figure 9-3 Top-Down Synthesis Flow with Multiple Library Domains


Start
Begin setup
Create library domains
Target librarie
Read target libraries for library domains
HDL files
Read HDL files
Elaborate design
SDC constrain
Set Timing and design constraints
Modify constraints
Apply optimization directives
Modify optimization directives
Associate library domains with portions of design
Synthesize design
Analyze design
Meet constraints?
No
Yes
Continue with recommended flow
Task added or modified for multiple

Example 9-1 Script for Top-Down Synthesis Flow Using Multiple Library Domains

```
# general setup
#--------------
set_db init_lib_search_path path
set_db init_hdl_search_path path
# create library domains
#-----------------------
create_library_domain domain_list
# specify the target libraries for each library domain
#-------------------------------------------------------
set_db [get_db library_domains *domain1] .library library_list1
set_db [get_db library_domains *domain2] .library library_list2
...
# load and elaborate the design
#------------------------------
read_hdl design.v
elaborate
# specify timing and design constraints
#---------------------------------------
# specify the following constraints per library domain
#------------------------------------------------------
set_db [get_db library_domains *domain] .operating_conditions string
set_db [get_db library_domains *domain] .wireload_selection string
# set target library domain for top design
#-----------------------------------------
set_db design .library_domain library_domain
# set target library domain for blocks
#-------------------------------------
uniquify subdesign
set_db subdesign .library_domain library_domain
#synthesize the design
#---------------------
syn_generic

syn_map
# analyze design
------------------
report_timing
report_gates
# export design
#--------------
write_design [-basename string] [-gzip_files] [-tcf]
[-innovus] [-hierarchical] [design]
```

Flow Steps

Begin Setup

For more information on the setup, seeSpecifying Explicit Search Paths on page 19.

Create Library Domains

A library domain is a collection of libraries. You can use library domains to associate *dedicated* libraries or library cells with portions of the design.

➤ To create library domains, use the `create_library_domain` command:

```
create_library_domain domain_list
```

Note: There is no limitation on the number of library domains you can create.

Tip

Create as many library domains

- ❑ As there are portions in the design for which you want to use dedicated libraries.

  In the example of Figure 9-1 on page 96, you want to use three different sets of libraries, which requires you to create three library domains.

- ❑ As the number of library cell sets that you want to use.

  In the example of Figure 9-2 on page 97, you want to use three different cell sets, so you need to create three library domains.

This command returns the directory path to the library domains that it creates. You need this information when loading in the target libraries.

To get meaningful timing results, all libraries *should* have been characterized for the same nominal operating conditions. If the libraries have different operating conditions, the nominal operating conditions of the last library will be used and thus the last library also determines the voltage of the library domain. For this flow the voltage of all library domains *must* be the same.

For example, to find the active operating condition of a specific domain, use

```
get_db [get_db library_domains *domain] .active_operating_conditions
```

To find the voltage of a domain, you need use the active operating condition of the domain:

```
get_db [get_db [get_db library_domains *domain] .active_operating_conditions] \
 .voltage
```

Tip

You can always rename or remove a library domain. For more information on removing library domains, refer to Removing a Library Domain.

Read Target Libraries for Library Domains

Next, you need to associate the libraries with the library domains.

To read in the libraries for a specific library domain, set the `library` attribute for the corresponding domain:

```
set_db [get_db library_domains *domain] .library library_list
```

Note: There is no limitation on the number of libraries you can read in per domain.

Tip

When targeting different cells from the same library for different portions of design, you need to

a. Read in the same library for each library domain:

```
set_db [get_db library_domains *dom1] .library lib
set_db [get_db library_domains *dom2] .library lib
set_db [get_db library_domains *dom3] .library lib
```

b. Exclude library cells.

Because library `lib` is used in three library domains, Genus treats it as if three different libraries were read in. This allows to exclude cells in one library domain, while you allow the use of them in a different library domain.

Within each library domain, you can exclude the library cells that the mapper should not use. To exclude a cell you can use the `avoid` attribute. For example:

```
set_db /lib*/library_domains/dom2/lib/libcells/AO22X2M .avoid true
set_db [get_db library:dom2/lib .lib_cells *AO22X2M] .avoid true
```

The first library domain for which you read in the libraries, becomes the default library domain.

➤ To change the default library domain, set the following attribute:

```
set_db desired_library_domain .default true
```

For more information on the use of the default library domain, refer to Removing a Library Domain.

Read HDL Files

Read designs using `read_hdl` command. For more information on reading HDL files, see Loading Files in *Genus User Guide.*

Elaborate Design

Refer to `elaborate` command for details. For more information on elaborating a design, see Performing Elaboration in *Genus User Guide*.

Set Timing and Design Constraints

Except for the following constraints, which *should* be set per library domain, all other (SDC and Genus native) constraints are set as in the regular top-down flow.

```
set_db [get_db library_domains *domain] .operating_conditions string
set_db [get_db library_domains *domain] .wireload_selection string
```

For more information on setting design constraints, see Applying Constraints in *Genus User Guide*.

Tip

When you set the `force_wireload` attribute on a design or subdesign, make sure that the wireload model you set matches a wireload model defined for a library in the associated library domain.

For more information on optimization strategies and related commands, see Defining Optimization Settings in *Genus User Guide*.

Associate Library Domains with Different Blocks of Design

To inform Genus about the special library use, associate the library domains with the design and blocks for which you want to use the dedicated libraries or dedicated library cells.

➤ To set the target library domain for the top design, specify the `library_domain` attribute on the design:

```
set_db design .library_domain library_domain
```

➤ To set the target library domain for a subdesign, do the following

a. Uniquify the subdesign:

```
uniquify subdesign
```

See Example 9-2 on page 104 for more information.

b. Specify the `library_domain` attribute on the subdesign:

```
set_db subdesign .library_domain library_domain
```

Note: This attribute is hierarchical. It applies to all instances of the specified design or subdesign. So the order in which you specify the target domains is important. See Example 9-3 on page 104 for more information.

When you *change* the library domain for a subdesign, Genus copies all attributes from the original mapped instances to the new mapped instances.

Important

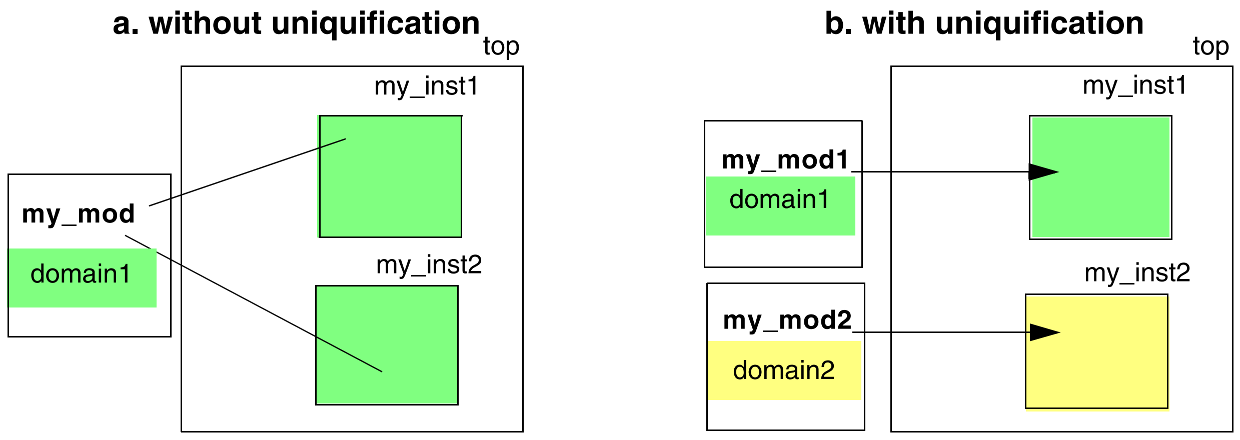
If you marked an instance *preserved*, the library domain that the instance is associated with will still be changed. However, the instance will still be marked preserved even though it will probably be pointing to another library cell in the new library domain. In other words, the `library_domain` attribute has a higher priority than the `preserve` attribute.

If Genus cannot find the corresponding cell in the libraries that belong to the new library domain, the instance becomes *unresolved*.

Example 9-2 Uniquifying a Subdesign before Associating the Library Domain

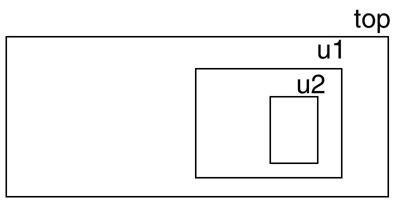
In the following design, module `top` has two instantiations of module `my_mod`. If you associate subdesign `my_mod` with library domain `domain1` before uniquifying subdesign `my_mod`, both instances `my_inst1` and `my_inst2` are associated with library domain `domain1`. This is illustrated in Figure 9-4. To associate instance `my_inst1` with library domain `domain1`, and instance `my_inst2` with library domain `domain2`, you first need to uniquify subdesign `my_mod`, and then associate both subdesigns with their domains individually.

Figure 9-4 Uniquifying a Subdesign before Associating the Library Domain



Example 9-3 Setting Target Library Domains

Assume a design `top` which has a subdesign `u1`. Subdesign `u1` has a subdesign `u2`.



Assume that all instances of u2 should be mapped using libraries from library domain `dom1`. All instances of u1, except for the instances of u2, should be mapped using libraries from library domain `dom2`. The instances in the remainder of the design should be mapped using libraries from library domain `dom1`.

To ensure the correct mapping, make the assignments in the following order:

```
set_db designs:top .library_domain [get_db library_domains *dom1]
set_db [get_db modules *u1] .library_domain [get_db library_domains *dom2]
set_db [get_db modules *u2] .library_domain [get_db library_domains *dom1]
```

Synthesize Design

After the constraints and optimizations are set for your design, you can proceed with synthesis.

➤ To synthesize your design use the `syn_generic` and `syn_map` commands.

Note: The design can be synthesized top down, without the need for manual partitioning.

The different portions of the design that are associated with different library domains will be mapped to the target libraries of those library domains and optimized.

For details on synthesizing the design, see Synthesizing your Design in *Genus User Guide*.

Analyze Design

After synthesizing the design, you can generate detailed timing and area reports using the various `report_*` commands.

For more information on generating reports for analysis, see Generating Reports in *Genus User Guide* and “Analysis and Report Commands” in the *Genus Command Reference*.

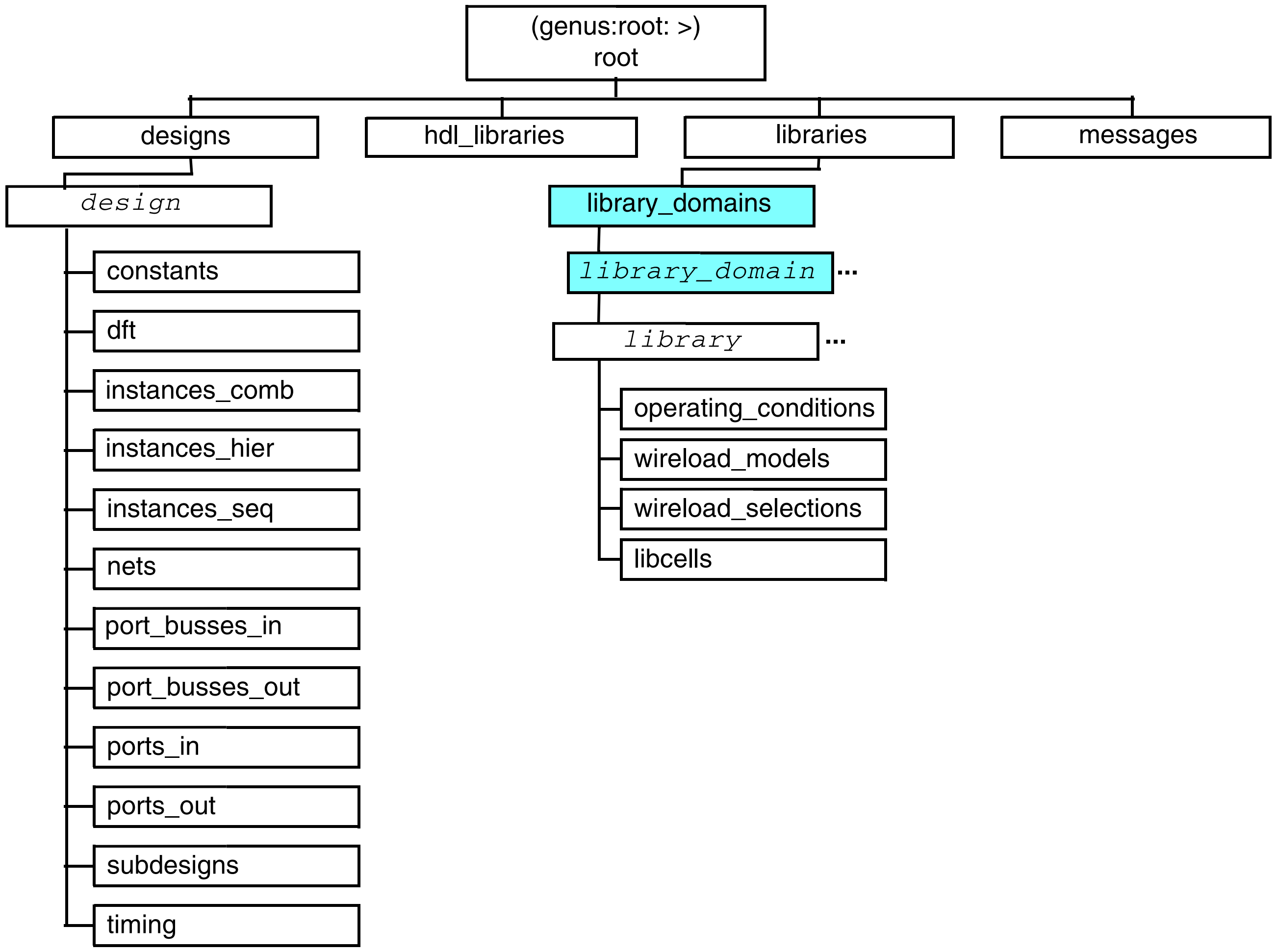
Most reports reflect information for the library domains.

In the timing report, the domain information is added to the `Type` column.

Library Domain Information in the Design Information Hierarchy

Genus stores the original design data along with additional information added by Genus in the design information hierarchy in the form of attributes. Figure 9-5 highlights where the library domain information is stored in the design information hierarchy.

Figure 9-5 Design Information Hierarchy



Additional Tasks

Removing a Library Domain

➤ To remove a library domain, use

```
delete_obj [get_db library_domains *domain]
```

When you remove a library domain, Genus removes

- The libraries that are part of that library domain
- Any level-shifter group that was referring from or to this library domain

Additional Notes

- Genus links any instances in the subdesigns associated with a library domain that is being removed to the default library domain. While relinking, Genus copies all attributes from the original mapped instance to the new mapped instance.

  Important

  If you marked an instance *preserved*, the library domain that the instance is associated with will still be removed. However, the instance will still be marked preserved even though it will probably be pointing to another library cell in the default library domain. In other words, the `library_domain` attribute has a higher priority than the `preserve` attribute.

  If Genus cannot find the corresponding cell in the libraries that belong to the default library domain, the instance becomes *unresolved*.

- You can remove the library domain that is marked the default library domain, by first setting the `default` attribute on another library domain.

  In the following example, `dom_1` is the default library domain. You can only remove library domain `dom1`, after changing the `default` for example to library domain `dom_2`,

```
genus:root: 12>set_db [get_db library_domains *dom_1] .default true
genus:root: 13>set_db [get_db library_domains *dom_2] .library typical.lib
    Setting attribute of library_domain 'dom_2': 'library' = typical.lib
genus:root: 14>set_db [get_db library_domains *dom_2] .default true
    : Set default library domain. [LBR-109]
    : The default domain changed from 'library_domain:dom1' to
     'library_domain:dom2'
    Setting attribute of library_domain 'dom2': 'default' = true
genus:root: 15> delete_obj [get_db library_domains *dom_1]
```

- You can only remove the default library domain if it is the only library domain that remains. If you remove the default library domain and a design was loaded, all instances become unresolved. In that case, none of the instances have timing, power or area information.

Saving Information

➤ To save the information for a later synthesis session, use the following commands:

```
write_hdl > design.v
write_script > design.scr
write_sdc > design.sdc
```

These commands will save specific information such as library domain-associations with portions of the design.

What If Analysis

You can check the effect on timing by using different library domains for some portions of the designs. The what-if analysis can be done before or after mapping, although you can get more meaningful results when you perform it after mapping.

Important

To do a what-if analysis, Genus expects that the libraries in the library domain you want to switch to, have the same set of cells with the same names as the libraries in the original library domain.

When you change the library domain assignment of a subdesign, Genus tries to rebind each cell in the subdesign by searching for a cell with the same name in a library in the new library domain. If Genus finds such cell, it uses the timing and power information of this cell from the new library domain to perform timing and power analysis and optimization. Otherwise, the original instance becomes an unresolved instance with the library cell name as its subdesign name.

What-If Analysis before Mapping Flow

1. Setup.
2. Create library domains.
3. Load libraries
4. Read netlist.
5. Elaborate design.
6. Set constraints.
7. Set optimization directives.
8. Assign library domains to portions of design.

What-if Analysis Steps

9. Report on timing (and power).
10. Reassign the target library domain for a portion of the design.
11. Report on timing (and power).
12. Repeat steps 10 through 11 until you are satisfied with results.

What-If Analysis after Mapping Flow

1. Setup.
2. Create library domains.
3. Load libraries
4. Read netlist.
5. Elaborate design.
6. Set constraints.
7. Set optimization directives.
8. Assign library domains to portions of design.

9. Map design.
10. Report on timing (and power).

What-if Analysis Steps

1. Save information (see Saving Information).
2. Reassign the target library domain for a portion of the design.
3. Either remap or do incremental synthesis.
4. Report on timing (and power).
5. Repeat steps 1 through 5 until you are satisfied with results.
6. Restore information depending on the what-if results.

Recommended Flow

```
# general setup
#--------------
set_db init_lib_search_path path
set_db init_hdl_search_path path
# create library domains
#-----------------------
create_library_domain domain_list
# specify the target libraries for each library domain
#-------------------------------------------------------
set_db [get_db library_domains *domain1] .library library_list1  /
set_db [get_db library_domains *domain2] .library library_list2 /
...
# load and elaborate the design
#------------------------------
read_hdl design.v
elaborate
# specify timing and design constraints
#---------------------------------------
# specify the following constraints per library domain
#------------------------------------------------------
set_db [get_db library_domains *domain] .operating_conditions string
set_db [get_db library_domains *domain] .wireload_selection string
# set target library domain for top design
#-----------------------------------------
set_db design .library_domain library_domain
# set target library domain for blocks
#-------------------------------------
uniquify subdesign
set_db subdesign .library_domain library_domain
#synthesize the design
#---------------------
syn_generic
syn_map
# analyze design
------------------
report_timing
report_gates
# export design
#--------------
write_design [-basename string] [-gzip_files] [-tcf]
[-innovus] [-hierarchical] [design]
```

10

Multi-Mode Multi-Corner Flow

Introduction

Chip design today is becoming very complicated due to integration. There are multiple sets of constraints, such as clocks, external delays, false paths, and multicycle paths, multiple PVT values, multiple library cells for some modes and multiple RC corners.

In a traditional synthesis flow, the design is synthesized in each mode and in each corner separately and the timing is closed by synthesizing all the different timing constraints. This can introduce a critical path in another view while you are trying to close timing in the current view. Multi-Mode Multi-Corner (MMMC) flow reduces extra design cycles by performing timing analysis and optimization simultaneously based on multiple SDC constraints, multiple RC corners, multiple libraries and multiple PVTs.

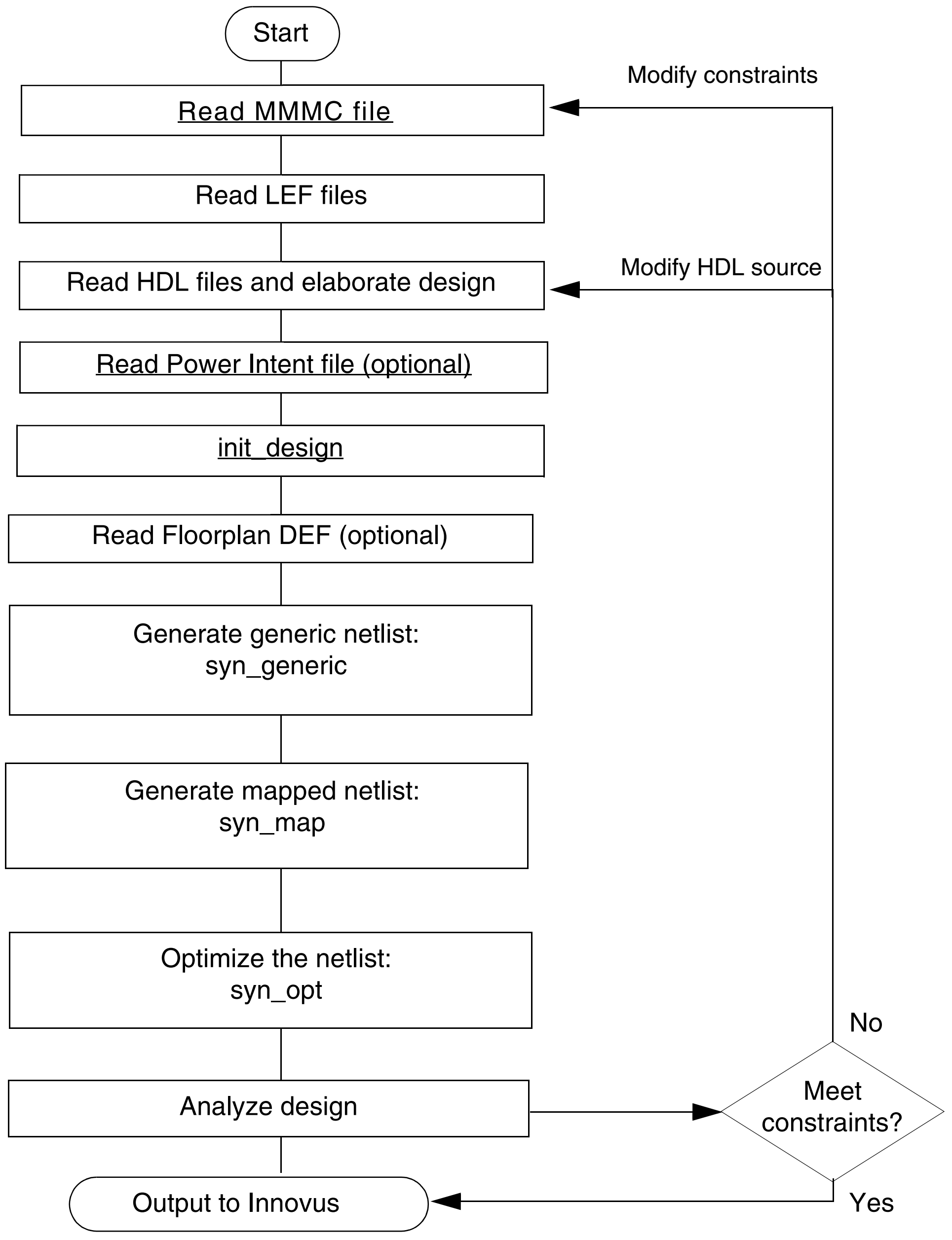
Hence, the MMMC flow allows to initiate multiple runs for different corners and modes and then collate the results.

Basic MMMC Flow

The basic MMMC flow can be broken down into following sub-groups:

- Reading MMMC file
- Reading LEF - to support physical design flows (optional)
- Reading RTL and elaborating the design
- Read power intent file
- Initializing the design
- Reading DEF (optional)
- Synthesize the design

Figure 10-1 Basic MMMC Flow

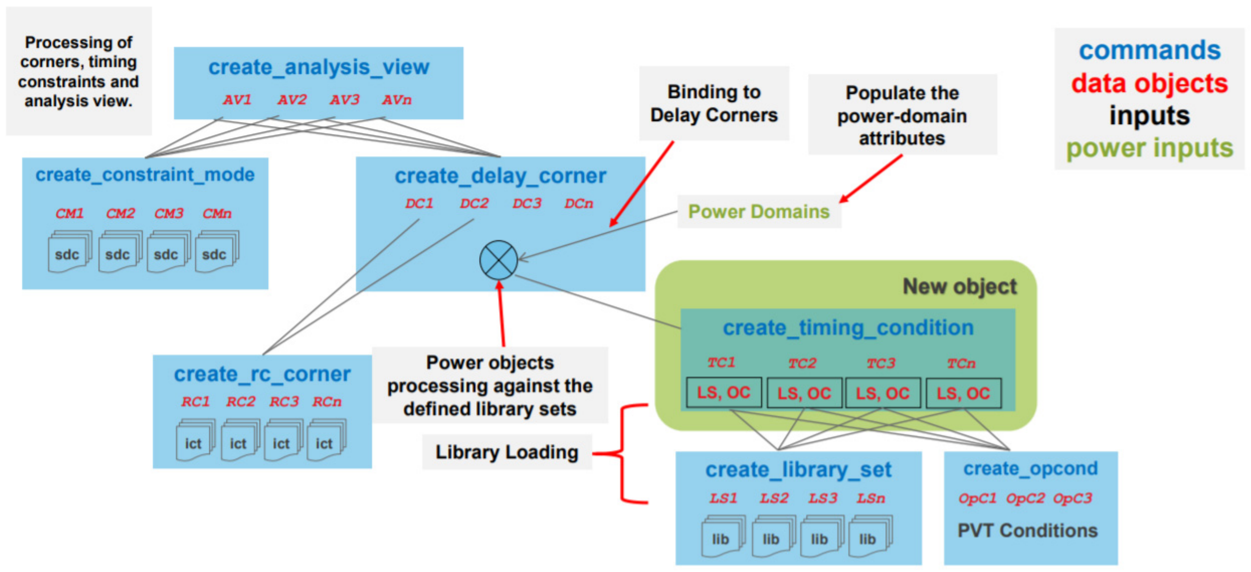

Start
Read MMMC file
Read LEF files
Read HDL files and elaborate design
Read Power Intent file (optional)
init_design
Read Floorplan DEF (optional)
Generate generic netlist: syn_generic
Generate mapped netlist: syn_map
Optimize the netlist: syn_opt
Analyze design
Meet constraints?
No
Modify constraints
Modify HDL source
Yes
Output to Innovus

Reading MMMC File

For MMMC flow, first you need to read the MMMC file. This file contains the details for the analysis views on which the design is synthesized. It is a simple TCL file containing details about libraries, operating conditions, timing conditions, delay corners and other such details.

You can read the MMMC file using `read_mmmc <filename.tcl>` command.

This is explained in detail in the following diagram.

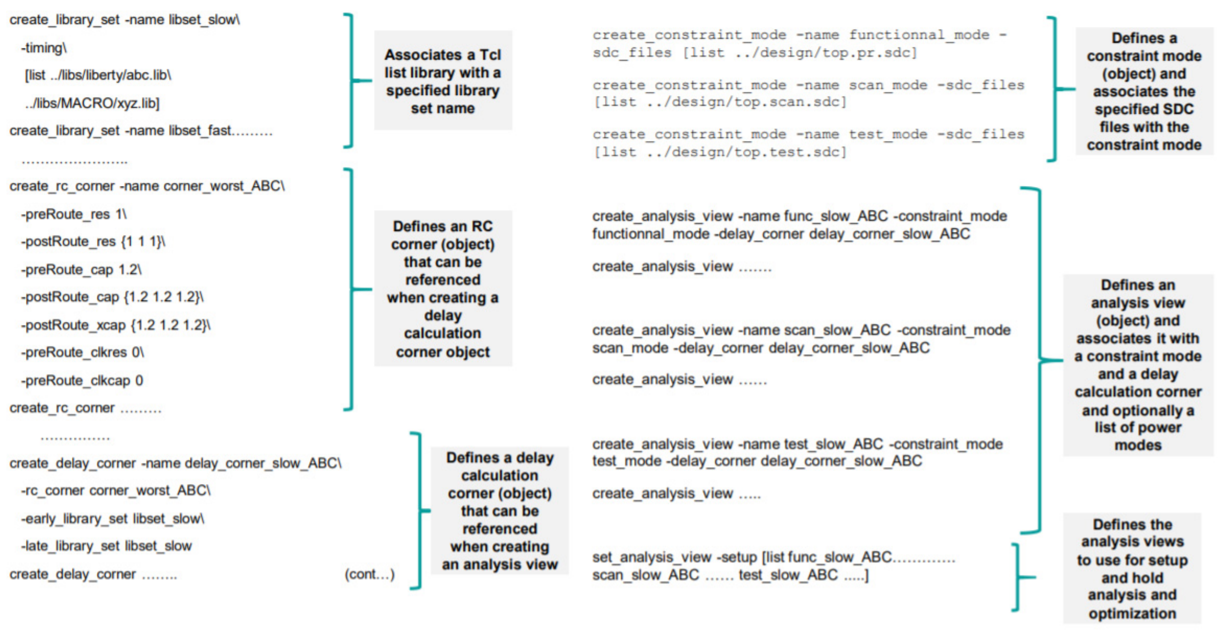


Components of the file:

1. Multi-Corners - These are used to specify the multi-corners in the design. These are specified using the `create_rc_corner` and `create_delay_corner` commands.
2. Multi-Modes - create the modes for the corners using the `create_constraint_mode` command.
3. Multi-Views - An analysis view object provides the information necessary to control a single multi-mode multi-corner analysis. These are different combinations of corners and modes which are then executed on a multi-threaded environment. `create_analysis_view` is command to create these views.
4. Set the views - Now you can decide what analysis views you want to set for running the flow. This can be done using the command `set_analysis_view`.

The components of the mmmc.tcl file are shown in the following figure:

Figure 10-2 Components of mmmc.tcl file



```
create_library_set -name libset_slow\
  -timing\
   [list ../libs/liberty/abc.lib\
   ../libs/MACRO/xyz.lib]
create_library_set -name libset_fast.........
  .......................
create_rc_corner -name corner_worst_ABC\
  -preRoute_res 1\
  -postRoute_res {1 1 1}\
  -preRoute_cap 1.2\
  -postRoute_cap {1.2 1.2 1.2}\
  -postRoute_xcap {1.2 1.2 1.2}\
  -preRoute_clkres 0\
  -preRoute_clkcap 0
create_rc_corner .........
     ...............
create_delay_corner -name delay_corner_slow_ABC\
  -rc_corner corner_worst_ABC\
  -early_library_set libset_slow\
  -late_library_set libset_slow
create_delay_corner ........                    (cont...)
```

Associates a Tcl list library with a specified library set name

Defines an RC corner (object) that can be referenced when creating a delay calculation corner object

Defines a delay calculation corner (object) that can be referenced when creating an analysis view

```
create_constraint_mode -name functionnal_mode -
sdc_files [list ../design/top.pr.sdc]

create_constraint_mode -name scan_mode -sdc_files
[list ../design/top.scan.sdc]

create_constraint_mode -name test_mode -sdc_files
[list ../design/top.test.sdc]
```

Defines a constraint mode (object) and associates the specified SDC files with the constraint mode

```
create_analysis_view -name func_slow_ABC -constraint_mode
functionnal_mode -delay_corner delay_corner_slow_ABC

create_analysis_view .......

create_analysis_view -name scan_slow_ABC -constraint_mode
scan_mode -delay_corner delay_corner_slow_ABC

create_analysis_view ......

create_analysis_view -name test_slow_ABC -constraint_mode
test_mode -delay_corner delay_corner_slow_ABC

create_analysis_view .....
```

Defines an analysis view (object) and associates it with a constraint mode and a delay calculation corner and optionally a list of power modes

```
set_analysis_view -setup [list func_slow_ABC.............
scan_slow_ABC ...... test_slow_ABC .....]
```

Defines the analysis views to use for setup and hold analysis and optimization

`read_mmmc` command helps to create the `library_set`, `rc_corner`, `timing_condition`, `constraint_mode`, and `analysis_view` objects as specified in the MMMC file. It will also load the timing libraries required by the active views from the `set_analysis_view` command in the MMMC file. Reading of files like SDC constraints is deferred until init_design on page 119. During `init_design`, only the files required by the active views from the `set_analysis_view` command are loaded. Timing libraries or constraint files for non-active views are not loaded unless those views are made active with `set_analysis_view` later in the flow.

Sample MMMC TCL script

```
## creating library_sets

create_library_set -name wcl_slow \
    -timing { slow1.lib slow2.lib \
    CDK_S128x16.lib }

create_library_set -name wcl_fast \
    -timing { fast1.lib \
            fast2.lib CDK_S128x16.lib }

create_library_set -name wcl_typical \
    -timing { typical.lib pllclk_slow.lib }

## creating operating conditions

create_opcond -name op_cond_wcl_slow -process 1.0 -voltage 1.08 -temperature 125.0

create_opcond -name op_cond_wcl_fast -process 1.0 -voltage 1.32 -temperature 0.0
create_opcond -name op_cond_wcl_typical -process 1.0 -voltage 1.2 -temperature 25.0
## creating timing_conditions
create_timing_condition -name timing_cond_wcl_slow \
    -opcond op_cond_wcl_slow \
    -library_sets { wcl_slow }
create_timing_condition -name timing_cond_wcl_fast \
    -opcond op_cond_wcl_fast \
    -library_sets { wcl_fast }
create_timing_condition -name timing_cond_wcl_typical \
    -opcond op_cond_wcl_typical \
    -library_sets { wcl_typical }
## creating rc_corners
create_rc_corner -name rc_corner \
    -cap_table typical.capTbl \
    -pre_route_res 1.0 \
    -pre_route_cap 1.0 \
    -pre_route_clock_res 0.0 \
    -pre_route_clock_cap 0.0 \
    -post_route_res {1.0 1.0 1.0} \
    -post_route_cap {1.0 1.0 1.0} \
    -post_route_cross_cap {1.0 1.0 1.0} \
    -post_route_clock_res {1.0 1.0 1.0} \
    -post_route_clock_cap {1.0 1.0 1.0}
## creating delay_corners
create_delay_corner -name delay_corner_wcl_slow \
    -early_timing_condition { timing_cond_wcl_slow } \
    -late_timing_condition { timing_cond_wcl_slow } \
    -early_rc_corner rc_corner \
    -late_rc_corner rc_corner
create_delay_corner -name delay_corner_wcl_fast \
    -early_timing_condition { timing_cond_wcl_fast } \
    -late_timing_condition { timing_cond_wcl_fast } \
    -early_rc_corner rc_corner \
    -late_rc_corner rc_corner
create_delay_corner -name delay_corner_wcl_typical \
    -early_timing_condition { timing_cond_wcl_typical } \
    -late_timing_condition { timing_cond_wcl_typical } \
    -early_rc_corner rc_corner \
    -late_rc_corner rc_corner
## setting constraint_modes
create_constraint_mode -name functional_wcl_slow \
    -sdc_files { ./functional_wcl_slow.sdc }
create_constraint_mode -name functional_wcl_fast \
    -sdc_files { ./functional_wcl_fast.sdc }
create_constraint_mode -name functional_wcl_typical \
    -sdc_files { ./functional_wcl_typical.sdc }
##creating analysis_views
create_analysis_view -name view_wcl_slow \
    -constraint_mode functional_wcl_slow \
    -delay_corner delay_corner_wcl_slow
```

```
create_analysis_view -name view_wcl_fast \
    -constraint_mode functional_wcl_fast \
    -delay_corner delay_corner_wcl_fast
create_analysis_view -name view_wcl_typical \
    -constraint_mode functional_wcl_typical \
    -delay_corner delay_corner_wcl_typical
## setting up analysis_views
set_analysis_view -setup { view_wcl_slow view_wcl_fast view_wcl_typical } \
                  -hold { view_wcl_slow view_wcl_fast view_wcl_typical }
set_analysis_view -setup { view_wcl_fast view_wcl_fast view_wcl_typical } \
                  -hold { view_wcl_fast view_wcl_fast view_wcl_typical }
```

Reading Power Intent Files

In this step, a power intent file (IEEE 1801, or CPF) is loaded. The power data is incomplete until the binding is done with the timing data during `init_design`. A power intent description is generally required for (Multi-Supply Voltage) MSV design.

init_design

It is an important step in the MMMC flow.

During execution of command `init_design`, the tool steps through the defined MMMC objects, design objects, and power requirements and builds the full design. Constraints are loaded, power data is loaded and applied to corresponding instances and operating conditions are applied to the power domains. The initialization also includes checks for completeness of data. After this step, the design is in a consistent state ready for synthesis.

Sub-steps of the `init_design` command:

1. Power domain creation - The design instances associated with a particular domain are defined in the Common Power Format (CPF) or 1801 file. This instance data is used to populate the power domain attributes. At the end of this step, the different power domains are resolved, and every instance in the design is associated with a power domain.
2. Power domain binding to domain corners - After the power domains are resolved, they can be bound to different library sets using `create_domain_corner` command. In this step, the active domain corners are processed and assigned to the appropriate library set.
3. Power domain object processing - After associating the power domains with the library sets, the power domain objects (state retention, ISO cells, and level shifters) are processed. The power object rules are processed against the defined library sets for each domain to identify the power cells to be used.

4. Loading of libraries - The library files (.lib) are loaded for each active analysis view.
5. Timing constraint loading - The timing constraints (.sdc files) are loaded for all active analysis views defined with the `set_analysis_view` command. The SDC files correspond to the constraint modes of the active analysis views. The SDC files of inactive analysis views are also loaded, to keep the timing constraints in sync with the design when it is synthesized. This enables the SDC files of inactive analysis views to be saved during `write_mmmc` in sync with the design changes. The RC corners defined by the required domain_corner are also processed and loaded.

After a design has been initialized with `init_design`, the design is ready for execution. Execution can include incremental updates to timing, synthesis and optimization.

Refer MMMC chapter in *Genus Attribute Reference* for all attributes related to MMMC flow in Genus.

Sample script

```
##Set all library paths
set_db init_lib_search_path {. ../}
set_db script_search_path { . }
set_db init_hdl_search_path {. ../}

set_db max_cpus_per_server 8
read_mmmc mmmc.tcl

read_physical -lef { \
      example1.lef example2.lef }

## Reading hdl files and elaborating them
read_hdl example.v
elaborate example
read_def example.def

##Initialize the design
init_design
time_info init_design
check_design -unresolved
## Set the innovus executable to be used for placement and routing
set_db innovus_executable path_to_innovus

## Synthesize the design
```

```
syn_generic -physical
syn_map -physical
syn_opt -physical

## generate reports to save the Innovus state
write_snapshot -innovus -directory ./output_dir -tag syn_opt_physical
report_summary -directory ./output_reports
puts "Runtime & Memory after 'syn_opt -physical'"

## write out the final database
write_db -to_file example1.db
```