Design Compiler®
User Guide

Version H-2013.03, March 2013

SYNOPSYS®

Copyright Notice and Proprietary Information

Destination Control Statement

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

Disclaimer

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

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at http://www.synopsys.com/Company/Pages/Trademarks.aspx.
All other product or company names may be trademarks of their respective owners.

Synopsys, Inc.
700 E. Middlefield Road
Mountain View, CA 94043
www.synopsys.com

Copyright Statement for the Command-Line Editing Feature

Copyright Statement for the Line-Editing Library

Contents

11. Using a Milkyway Database

12. Analyzing and Resolving Design Problems

Preface

This preface includes the following sections:

About This Manual

The *Design Compiler User Guide* provides basic synthesis information for users of the Design Compiler tools. This manual describes synthesis concepts and commands, and presents examples for basic synthesis strategies.

This manual does not cover asynchronous design, I/O pad synthesis, test synthesis, simulation, or back-annotation of physical design information.

The information presented here supplements the Synopsys synthesis reference manuals but does not replace them. See other Synopsys documentation for details about topics not covered in this manual.

This manual supports the Synopsys synthesis tools, whether they are running under the UNIX operating system or the Linux operating system. The main text of this manual describes UNIX operation.

Audience

This manual is intended for logic designers and engineers who use the Synopsys synthesis tools with the VHDL or Verilog hardware description language (HDL). Before using this manual, you should be familiar with the following topics:

- High-level design techniques
- ASIC design principles
- Timing analysis principles
- Functional partitioning techniques

Related Publications

For additional information about Design Compiler, see the documentation on SolvNet at the following address:

https://solvnet.synopsys.com/DocsOnWeb

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

- Design Vision
- DesignWare components
- DFT Compiler and DFTMAX
- DC Explorer

- PrimeTime
- Power Compiler
- HDL Compiler
- IC Compiler

Release Notes

Information about new features, changes, enhancements, known limitations, and resolved Synopsys Technical Action Requests (STARs) is available in the *Design Compiler Release Notes* in SolvNet.

To see the *Design Compiler Release Notes*,

1. Go to the Download Center on SolvNet located at the following address:

   https://solvnet.synopsys.com/DownloadCenter

2. Select Design Compiler, and then select a release in the list that appears.

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
`Courier`	Indicates syntax, such as `write_file`.
`Courier italic`	Indicates a user-defined value in syntax, such as `write_file` *`design_list`*.
`Courier bold`	Indicates user input—text you type verbatim—in examples, such as `prompt>` **`write_file top`**
[ ]	Denotes optional arguments in syntax, such as `write_file [-format` *`fmt`*`]`
...	Indicates that arguments can be repeated as many times as needed, such as *`pin1 pin2 ... pinN`*
|	Indicates a choice among alternatives, such as `low | medium | high`
Ctrl+C	Indicates a keyboard combination, such as holding down the Ctrl key and pressing C.
\	Indicates a continuation of a command line.
/	Indicates levels of directory structure.
Edit > Copy	Indicates a path to a menu command, such as opening the Edit menu and choosing Copy.

Customer Support

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

Accessing SolvNet

SolvNet includes a knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. SolvNet also gives you access to a wide range of Synopsys online services including software downloads, documentation, and technical support.

To access SolvNet, go to the following address:

https://solvnet.synopsys.com

If prompted, enter your user name and password. If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.

If you need help using SolvNet, click HELP in the top-right menu bar.

Contacting the Synopsys Technical Support Center

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

- Open a support case to your local support center online by signing in to SolvNet at https://solvnet.synopsys.com, clicking Support, and then clicking “Open A Support Case.”
- Send an e-mail message to your local support center.
  - E-mail support_center@synopsys.com from within North America.
  - Find other local support center e-mail addresses at http://www.synopsys.com/Support/GlobalSupportCenters/Pages
- Telephone your local support center.
  - Call (800) 245-8005 from within North America.
  - Find other local support center telephone numbers at http://www.synopsys.com/Support/GlobalSupportCenters/Pages

1

Introduction to Design Compiler

The Design Compiler tool is the core of the Synopsys synthesis products. Design Compiler optimizes designs to provide the smallest and fastest logical representation of a given function. It comprises tools that synthesize your HDL designs into optimized, technology-dependent, gate-level designs. It supports a wide range of flat and hierarchical design styles and can optimize both combinational and sequential designs for speed, area, and power.

Design Compiler also provides topographical technology, which allows you to accurately predict post-layout timing, area, and power during RTL synthesis without the need for timing approximations based on wire load models. It uses Synopsys placement and optimization technologies to drive accurate timing prediction within synthesis, ensuring better correlation with the final physical design.

In addition, Design Compiler provides the Design Compiler Graphical tool, which optimizes multicorner-multimode designs and allows you to create and modify floorplans using floorplan exploration. The tool also reduces routing congestion, and it improves area correlation with IC Compiler and runtime in IC Compiler.

For an overview of Design Compiler, see

- Design Compiler in the Design Flow
- High-Level Design Flow Tasks
- Design Compiler Family

Design Compiler in the Design Flow

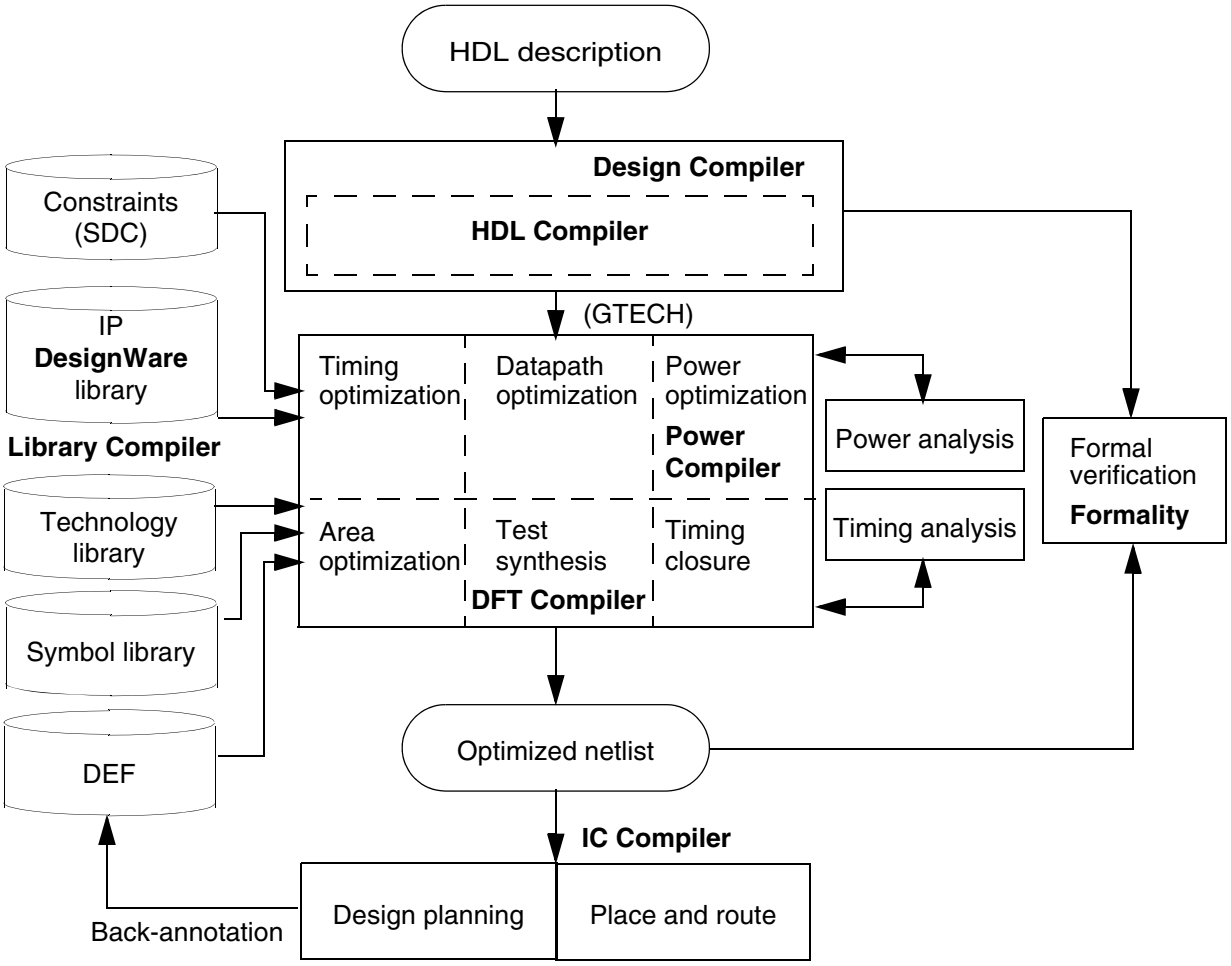
You use Design Compiler for logic synthesis, which is the process of converting a design description written in a hardware description language, such as Verilog or VHDL, into an optimized gate-level netlist mapped to a specific logic library. When the synthesized design meets functionality, timing, power, and other design goals, you can pass the design to IC Compiler for physical implementation.

Even though the following terms have slightly different meanings, they are often used synonymously in the Design Compiler documentation:

- *Synthesis* is the process that generates a gate-level netlist for an IC design that has been defined with a hardware description language (HDL). Synthesis includes reading the HDL source code and optimizing the design created from that description.
- *Optimization* is the step in the synthesis process that implements a combination of library cells that best meet the functional, timing, area, and power requirements of the design.
- *Compile* is the Design Compiler process that executes the synthesis and optimization steps. After you read in the design and perform other necessary tasks, you run the `compile_ultra` or `compile` command to generate a gate-level netlist for the design.

Figure 1-1 shows an overview of how Design Compiler fits into the design flow.

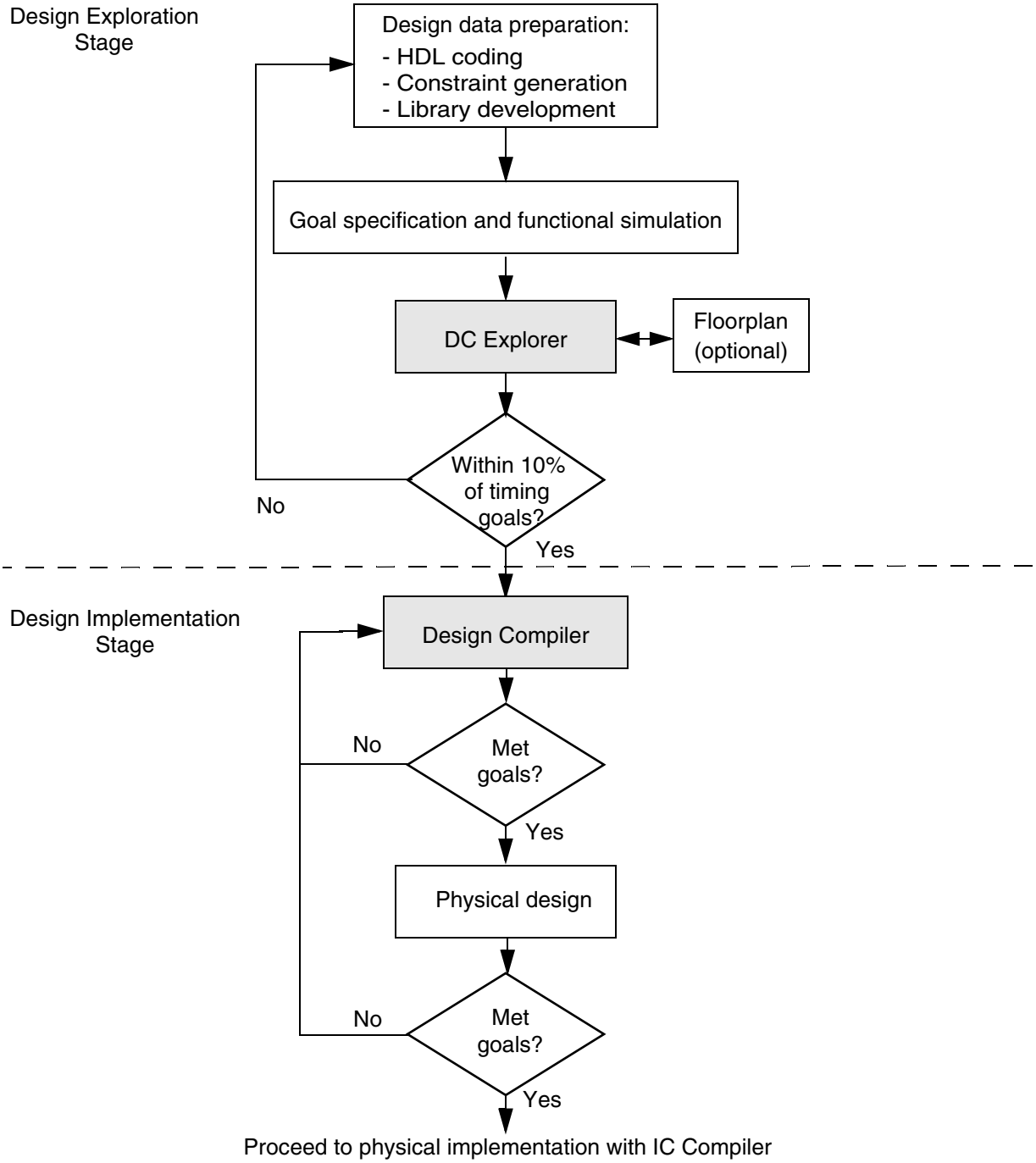
Figure 1-1 Design Compiler in the Design Flow



High-Level Design Flow Tasks

Figure 1-2 shows the high-level design flow from HDL coding to physical implementation in IC Compiler. The shaded areas indicate where the design exploration and synthesis tasks occur in the flow.

Figure 1-2 High-Level Design Flow



The synthesis design flow consists of the design exploration stage and the final design implementation stage. In the design exploration stage, you use DC Explorer to perform what-if analyses of various design configurations early in the design cycle to speed the

development of high-quality RTL and constraints and drive a faster, more convergent design flow. In the design implementation stage, you use the full power of Design Compiler to synthesize the design.

Using the high-level design flow shown in Figure 1-2, you perform the following tasks:

1. Write an HDL description of your design in Verilog or VHDL. Use good coding practices to facilitate successful Design Compiler synthesis of the design.
2. Perform design exploration and functional simulation in parallel.
   - In design exploration, use DC Explorer to (a) implement specific design goals, such as design rules and optimization constraints, (b) detect mismatches and missing constraints, and (c) resolve mismatches and design data inconsistencies.

     You can also create and modify floorplans early in the design cycle with floorplan exploration.
   - If design exploration fails to meet timing goals by more than 10 percent, modify your design goals and constraints, or improve the HDL code. Then repeat both design exploration and functional simulation.
   - In functional simulation, determine whether the design performs the desired functions by using an appropriate simulation tool.
   - If the design does not function as required, you must modify the HDL code and repeat both design exploration and functional simulation.
   - Continue performing design exploration and functional simulation until the design is functioning correctly and is within 10 percent of the timing goals.
3. Perform design implementation synthesis by using Design Compiler to meet design goals.

   After synthesizing the design into a gate-level netlist, verify that the design meets your goals. If the design does not meet your goals, generate and analyze various reports to determine the techniques you might use to correct the problems.
4. After the design meets functionality, timing, power, and other design goals, proceed to the physical implementation stage in IC Compiler.
5. Analyze the physical design's performance by using back-annotated data. If the results do not meet design goals, resolve them in IC Compiler or return to step 3. If the results meet your design goals, you are finished with the design cycle.

See Also

- Design Compiler in the Design Flow
- Running a Synthesis Flow

Design Compiler Family

The Design Compiler family provides an integrated RTL synthesis solution to address today's challenging IC designs. Using Design Compiler tools, you can

- Produce fast, area- and power-efficient IC designs using advanced optimizations and shared technology with IC Compiler place and route
- Predict, visualize, and alleviate routing congestion
- Perform floorplan exploration to create and modify design floorplans
- Explore design tradeoffs involving design constraints, such as timing, area, and power, under various loading, temperature, and voltage conditions

To learn about the Design Compiler family of products, see

- About DC Ultra
- About Design Compiler Graphical
- About DC Expert
- About DC Explorer
- About Design Vision
- About DesignWare Library
- About DFT Compiler and DFTMAX
- About Library Compiler
- About Power Compiler

About DC Ultra

At the core of the Synopsys RTL synthesis solution is DC Ultra. DC Ultra provides concurrent optimization of timing, area, power, and test for today's high performance designs. DC Ultra includes topographical technology, which allows you to accurately predict post-layout timing, area, and power, ensuring better correlation with the final physical design.

DC Ultra provides the following features:

- Placement and optimization technologies that are shared with IC Compiler place and route to drive accurate timing and area prediction within synthesis, ensuring a better starting point for physical implementation
- Advanced delay optimization algorithms
- Advanced arithmetic optimization
- Integrated datapath partitioning and synthesis capabilities
- Advanced critical path resynthesis
- Register retiming, the process by which the tool moves registers through combinational gates to improve timing
- Advanced timing analysis
- Support for multivoltage and multiple supply designs
- Infrastructure to support multicore execution for faster runtimes
- Support for hierarchical compile (top down or bottom up)
- Full and incremental compile techniques
- Sequential optimization for complex flip-flops and latches
- Command-line interface and graphical user interface

See Also

- Overview of Topographical Technology

About Design Compiler Graphical

In addition to DC Ultra capabilities, Design Compiler Graphical provides the following features:

- Optimization for multicorner-multimode designs
- Reduction of routing congestion during synthesis
- Improved area and timing correlation with IC Compiler
- Improved runtime and routability in IC Compiler
- Physical guidance technology, which includes enhanced placement and the capability to pass seed placement to IC Compiler to improve quality of results (QoR), correlation, and routability
- Ability to create and modify floorplans using floorplan exploration

See Also

- Using the Design Compiler Graphical Tool

About DC Expert

DC Expert provides optimization for area, timing, and power using wire load models for delay estimation.

DC Expert provides the following features:

- Hierarchical compile (top down or bottom up)
- Full and incremental compile techniques
- Sequential optimization for complex flip-flops and latches
- Time borrowing for latch-based designs
- Timing analysis
- Command-line interface and graphical user interface

About DC Explorer

Developing new RTL and integrating it with third-party IP and many legacy RTL blocks can be a time-consuming process when designers lack a fast and efficient way to explore and improve the data, fix design issues, and create a better starting point for RTL synthesis. DC Explorer overcomes these problems by allowing you to perform early RTL exploration, leading to a better starting point for RTL synthesis and accelerating design implementation.

DC Explorer provides the following features:

- Efficiently performs what-if analyses of various design configurations early in the design cycle, even with incomplete design data, to speed the development of high quality RTL and constraints and drive a faster, more convergent design flow
- Generates an early netlist, which can be used to begin physical exploration in IC Compiler
- Creates and modifies floorplans very early in the design cycle with access to IC Compiler design planning
- Performs preliminary synthesis using only a small fraction of the time needed for full synthesis, yet gives you timing and area results typically within ten percent of the final results produced by Design Compiler in topographical mode

See Also

- The *DC Explorer User Guide*

About Design Vision

The Design Vision tool is the graphical user interface (GUI) for the Synopsys logic synthesis environment and provides analysis tools for viewing and analyzing designs at the generic technology (GTECH) level and gate level. The Design Vision main window provides menus and dialog boxes for running frequently used Design Compiler commands. It also provides graphical displays, such as histograms and schematics for visual analysis.

When you start Design Vision in topographical mode, the Design Vision layout window lets you analyze physical constraints, timing, and congestion in your floorplan. A layout view displays floorplan constraints, critical timing paths, and congested areas in a single, flat view of the physical design. This information can help you to guide later optimization operations in Design Compiler and other Synopsys tools.

See Also

- The *Design Vision User Guide*
- Design Vision Help

About HDL Compiler

HDL Compiler translates Verilog or VHDL hardware language descriptions into a generic technology (GTECH) netlist, which is used by Design Compiler to create an optimized netlist.

See Also

- HDL Coding for Synthesis
- The HDL Compiler documentation

About DesignWare Library

A DesignWare library is a collection of reusable circuit-design building blocks, which are tightly integrated into the Synopsys synthesis environment. During synthesis, Design Compiler selects the right component with the best speed and area optimization from the DesignWare Library.

See Also

- The DesignWare Library documentation

About DFT Compiler and DFTMAX

The DFT Compiler tool is the Synopsys advanced test synthesis solution. It enables transparent implementation of design-for-test capabilities into the Synopsys synthesis flow without interfering with functional, timing, signal integrity, or power requirements.

DFTMAX compression provides synthesis-based adaptive scan technology to lower the cost of testing complex designs, particularly when fabricated with advanced process technologies. These deep-submicron (DSM) designs can have subtle manufacturing defects that are only detected by applying DSM tests, such as at-speed and bridging tests, in addition to stuck-at tests. The extra patterns needed to achieve high test quality for these designs can increase both the test time and the test data, resulting in higher test costs. DFTMAX reduces these costs by delivering 10-100x test data and test time reduction with very low silicon area overhead. DFTMAX uniquely enables adaptive scan compression synthesis in Design Compiler and adaptive scan pattern generation in TetraMAX ATPG.

See Also

- The DFT Compiler and DFTMAX documentation

About Library Compiler

Library Compiler reads the description of an ASIC library from a text file and compiles the description into either an internal database (.db file format) or into VHDL libraries. The compiled database supports synthesis tools. The VHDL libraries support VHDL simulation tools.

See Also

- The Library Compiler documentation

About Power Compiler

The Power Compiler tool offers a complete methodology for power, including analyzing and optimizing designs for static and dynamic power consumption.

See Also

- Power Optimization in Topographical Mode
- The *Power Compiler User Guide*

2

Design Compiler Basics

Design Compiler offers two interfaces for synthesis and timing analysis: the dc_shell command-line interface (or shell) and the Design Vision graphical user interface (GUI). The dc_shell command-line interface is a text-only environment in which you enter commands at the command-line prompt. Design Vision is the GUI for the Synopsys synthesis environment; use it for visualizing design data and analyzing results.

To learn the standard tasks for working in the Design Compiler environment and running a synthesis flow, see

Running Design Compiler

To learn basic information for running Design Compiler using dc_shell and the GUI, see

- Starting Design Compiler
- Entering dc_shell Commands
- Interrupting or Terminating Command Processing
- The Setup Files
- Using the GUI
- Finding Session Information in the Log Files
- Using Script Files
- Saving Designs and Exiting Design Compiler

Starting Design Compiler

Design Compiler operates in the X windows environment on UNIX or Linux. Before starting Design Compiler, make sure your `$SYNOPSYS` variable is set, and the path to the bin directory is included in your `$PATH` variable.

To start Design Compiler, enter `dc_shell` in a UNIX or Linux shell:

```
% dc_shell
```

If you are using Design Compiler in topographical mode, specify the `-topographical_mode` option:

```
% dc_shell -topographical_mode
```

By default, these commands start the tool in the command-line interface (dc_shell).

When you start the command-line interface, the dc_shell prompt appears in the UNIX or Linux shell. If you specified Design Compiler in topographical mode, the resulting command prompt is

```
dc_shell-topo>
```

You can open the GUI when you start the tool by using the `-gui` option. For example, you can enter either of the following commands:

```
% dc_shell -gui
% dc_shell -topographical_mode -gui
```

Note:
You must have a Design Vision license to use the GUI. Before opening the GUI, make sure your `$DISPLAY` environment variable is set to the name of your UNIX or Linux system display.

When you open the GUI, the Design Vision main window appears. For more information about working in the GUI, see Using the GUI.

You can also include numerous options when you start the tool, such as

- `-f` to execute a script file before displaying the initial dc_shell prompt.
- `-x` to include a dc_shell statement that is executed at startup.
- `-checkout` to access licensed features in addition to the default features checked out by the program.
- `-wait` to set a wait time limit for checking out any additional licenses.
- `-no_init` to prevent dc_shell from executing any .synopsys_dc.setup startup files. Use this option only when you want to include a command log or other script file to reproduce a previous dc_shell session.
- `-no_home_init` to prevent dc_shell from executing any home .synopsys_dc.setup startup files.
- `-no_local_init` to prevent dc_shell from executing any local .synopsys_dc.setup startup files.
- `-64bit` to invoke the 64-bit executable of the Design Compiler command shell.

At startup, dc_shell does the following tasks:

1. Creates a command log file.
2. Reads and executes the .synopsys_dc.setup files, as described in The Setup Files.
3. Executes any script files or commands specified by the `-f` and `-x` options, respectively, on the command line.
4. Displays the program header and dc_shell prompt in the window from which you invoked dc_shell. The program header lists all features for which your site is licensed.

It is important that you specify the absolute path, as shown, when invoking dc_shell on the command line to indicate the Synopsys root containing the Design Compiler installation:

```
/tools/synopsys/2013.03/bin/dc_shell
```

If you use a relative path (../), as shown, Design Compiler cannot access the libraries that are located in the root directory:

```
../../2013.03/bin/dc_shell
```

See Also

- Saving Designs and Exiting Design Compiler

Entering dc_shell Commands

You interact with the Design Compiler shell by using dc_shell commands, which are based on the tool command language (Tcl) and include certain command extensions needed to implement specific Design Compiler functionality. The Design Compiler command language provides capabilities similar to UNIX command shells, including variables, conditional execution of commands, and control flow commands. You can

- Enter individual commands interactively at the dc_shell prompt
- Run one or more Tcl command scripts, which are text files that contain dc_shell commands

You enter commands in dc_shell the same way you enter commands in a standard UNIX or Linux shell.

To enter a command in dc_shell,

1. Type the command on the command line.
2. Press Enter.

When the GUI is open, you can enter commands on the console command line and use the commands available through the menu interface.

To enter a command on the console command line,

1. Click anywhere on the console to make sure the command line is active.
2. Type the command.
3. Click the dc_shell prompt button or press Enter.

Design Compiler echoes the command output, including processing messages and any warnings or error messages, in both dc_shell and the console log view.

When entering a command, option, or file name, you can minimize your typing by pressing the Tab key when you have typed enough characters to specify a unique name; Design Compiler completes the remaining characters. If the characters you typed could be used for more than one name, Design Compiler lists the qualifying names from which you can select by using the arrow keys and the Enter key.

To reuse a command from the output for a command-line interface, copy and paste the portion by selecting it, moving the pointer to the dc_shell command line, and clicking the middle mouse button.

In the GUI, you can select commands in the console history view and either rerun them or copy them to the command line, where you can edit them. For more information, see the "Viewing the Command History" topic in Design Vision Help.

See Also

- *Using Tcl With Synopsys Tools*

Interrupting or Terminating Command Processing

If you enter the wrong options for a command or enter the wrong command, you can interrupt command processing and remain in dc_shell. To interrupt or terminate a command, press Ctrl+C.

Some commands and processes, such as the `update_timing` command, cannot be interrupted. To stop these commands or processes, you must terminate dc_shell at the system level. When you terminate a process or the shell, no data is saved.

When you press Ctrl+C, remember the following points:

- If a script file is being processed and you interrupt one of its commands, the script processing is interrupted and no further script commands are processed.
- If you press Ctrl+C three times before a command responds to your interrupt, dc_shell is interrupted and exits with the following message:

  ```
  Information: Process terminated by interrupt.
  ```

This behavior has a few exceptions, which are documented in the man pages for the applicable commands.

The Setup Files

When you invoke Design Compiler, it automatically executes commands in three setup files. These files have the same file name, .synopsys_dc.setup, but reside in different directories. The files can contain commands that initialize parameters and variables, declare design libraries, and so forth.

When you open the GUI, the tool reads another set of setup files named .synopsys_dv_gui.tcl. You can use these files to perform GUI-specific setup tasks. Settings from the .synopsys_dv_gui.tcl files override settings from the .synopsys_dc.setup files.

Design Compiler reads the .synopsys_dc.setup and .synopsys_dv_gui.tcl files from three directories in the following order:

1. The Synopsys root directory ($SYNOPSYS/admin/setup)

   These system-wide setup files contain system variables defined by Synopsys and general Design Compiler setup information for all users at your site. Only the system administrator can modify these files.

2. Your home directory

   These user-defined setup files can contain variables that define your preferences for the Design Compiler working environment. The variables in these files override the corresponding variables in the system-wide setup files.

3. The current working directory (the directory from which you start Design Compiler)

   These design-specific setup files can contain project-specific or design-specific variables that affect the optimizations of all designs in this directory. To use the files, you must invoke Design Compiler from this directory. Variables defined in these files override the corresponding variables in the user-defined and system-wide setup files.

Example 2-1 shows a .synopsys_dc.setup file.

Example 2-1 .synopsys_dc.setup File

```
# Define the target logic library, symbol library,
# and link libraries
set_app_var target_library lsi_10k.db
set_app_var symbol_library lsi_10k.sdb
set_app_var synthetic_library dw_foundation.sldb
set_app_var link_library "* $target_library $synthetic library"
set_app_var search_path [concat $search_path ./src]
set_app_var designer "Your Name"

# Define aliases
alias h history
alias rc "report_constraint -all_violators"
```

Using the GUI

The GUI allows you to use the same design methodology and scripts that you use in dc_shell and to extend your methodology with visual analysis. Many Design Compiler commands are available on Design Vision menus. All Design Compiler functions are available through the command-line interface on the console. You must have a Design Vision license to use the GUI from a dc_shell session.

To learn basic tasks for using the GUI, see

- Opening and Closing the GUI
- Saving Window Images in the GUI
- Saving Designs and Exiting Design Compiler From the GUI

Opening and Closing the GUI

You can open the GUI at any time from the Design Compiler command-line interface (dc_shell). You can open or close the GUI multiple times during a session. Before opening the GUI, make sure your `$DISPLAY` environment variable is set to the name of your UNIX or Linux system display.

To open the GUI, enter the `gui_start` command at the dc_shell prompt:

```
dc_shell> gui_start
```

Note:

You can also open the GUI when you start Design Compiler by specifying the `-gui` option with the `dc_shell` command. For more information, see Starting Design Compiler.

You can close the GUI without exiting Design Compiler. For example, if you need to save system resources, you can close the GUI and leave Design Compiler running as a command-line interface.

To close the GUI, do one of the following:

- Choose File > Close GUI.
- Enter the `gui_stop` command:

  ```
  dc_shell> gui_stop
  ```

When you are ready to use the GUI again, enter the `gui_start` command on the command line interface.

When you open the GUI, the tool reads a set of .synopsys_dv_gui.tcl setup files. You can use these files to perform GUI-specific setup tasks. Settings from the .synopsys_dv_gui.tcl

files override settings from the .synopsys_dc.setup files. For more information about the .synopsys_dv_gui.tcl files, see the *Design Vision User Guide*.

In addition to reading the setup files, the tool loads preferences and view settings from a file named .synopsys_dv_prefs.tcl in your home directory. You should not edit this file. The default system preferences are set for optimal tool operation and work well for most designs. However, if necessary, you can change GUI preferences during the session by using the Application Preferences dialog box. For more information about GUI preferences, see the "Setting GUI Preferences" topic in Design Vision Help.

Saving Window Images in the GUI

To save window images in the GUI, use the `gui_write_window_image` command or choose View > Save Screenshot As and set options in the Save Screenshot As dialog box. The `gui_write_window_image` command saves an image of a GUI window or a view window in the specified image file. You can specify the format of the image as BMP, JPEG, XPM, or PNG. The default format is PNG.

You can also save window images by entering the `gui_write_window_image` command on the command line or by using it in a batch script. For more details, see the "Saving an Image of a Window or View" topic in Design Vision Help.

Saving Designs and Exiting Design Compiler From the GUI

The tool does not automatically save the designs loaded in memory. To save the current design and each of its subdesigns in separate .ddc format files named *design_name*.ddc, where *design_name* is the name of the design, choose File > Save.

To save the current design and all of its subdesigns in a single file with a different file name or file format, choose File > Save As, enter or select a file name, select a file format, and click OK. For more information, see the "Saving the Design" topic in Design Vision Help.

To exit dc_shell from the GUI, you can do any of the following:

- Choose File > Exit, and then click OK in the message box that appears.
- Enter `exit` or `quit` on the command line.
- Press Ctrl+C three times in the UNIX or Linux shell.

See Also

- Using the Man Page Viewer
- Using Design Vision Help
- The *Design Vision User Guide*
- Design Vision Help

Finding Session Information in the Log Files

You can find session information, such as the dc_shell commands that were processed and the files that were accessed, in the following log files:

- Command Log Files
- Compile Log Files
- File Name Log Files

Command Log Files

The command log file records the dc_shell commands processed by Design Compiler, including setup file commands and variable assignments. By default, Design Compiler writes the command log to a file called command.log in the directory from which you invoked dc_shell.

You can change the name of the command.log file by setting the `sh_command_log_file` variable in the .synopsys_dc.setup file. You should make any changes to this variable before you start Design Compiler. If your user-defined or project-specific .synopsys_dc.setup file does not define the variable, Design Compiler automatically creates the command.log file.

Each Design Compiler session overwrites the existing command log file. To save a command log file, move or rename it. You can use the command log file to

- Produce a script for a particular synthesis strategy
- Record the design exploration process
- Document any problems you are having

Compile Log Files

Each time you compile a design, Design Compiler creates the following compile log files:

- ASCII log file

  The log displays the output, such as the commands that are processed and the error messages for each Design Compiler run, on the screen for quick viewing and debugging.

- HTML log file

  The file resides under the current working directory. It contains the complete contents of the ASCII log but in HTML format. At the end of this file, a summary table provides an overview of all occurrences of messages grouped by the message ID. You can click the message ID link to display the message.

To generate the HTML log file in dc_shell, set the `html_log_enable` variable to `true` before reading in the design. For example,

```
dc_shell> set_app_var html_log_enable true
```

By default, the `html_log_enable` variable is set to `false`, and only the ASCII log file is generated. The contents of the ASCII and HTML log files are identical. However, the way the contents are displayed is different. In the HTML log file, you can control the level of detail that is displayed by clicking the plus (+) or minus (-) buttons to expand or collapse multiple lines of messages.

By default, the HTML file name is default.html. To specify a different file name, set the `html_log_filename` variable. For example,

```
dc_shell> set_app_var html_log_filename my_HTML_log.html
```

Important:

You must have the Python programming language installed to generate an HTML log file. For download information, go to the following address:

http://www.python.org/download

File Name Log Files

By default, Design Compiler lists the names of the files that it has read to the file name log file in the directory from which you invoked dc_shell. You can use the file name log file to identify data files needed to reproduce an error if Design Compiler terminates abnormally. To specify the name of the file name log file, set the `filename_log_file` variable in the .synopsys_dc.setup file.

Using Script Files

Designers often use scripts to accomplish routine repetitive tasks such as setting constraints or defining other design attributes. You can use your existing Tcl scripts in the Design Compiler command-line interface and GUI.

You can create a script file by placing a sequence of dc_shell commands in a text file. You can also define scripts in your setup files. Any dc_shell command can be executed within a script file.

In Tcl, a pound sign (#) at the beginning of a line denotes a comment. For example,

```
# This is a comment.
```

To execute a script file, use the `source` command at the command prompt and specify the file name, as shown:

```
dc_shell-topo> source file_name
```

To execute a script file in the GUI, choose File > Execute Script and specify the file name in the Execute File dialog box.

See Also

- *Using Tcl With Synopsys Tools*
- The “Using dctcl Scripts” topic in Design Vision Help.

Saving Designs and Exiting Design Compiler

You can exit Design Compiler at any time and return to the operating system. By default, dc_shell saves the session information in the command.log file. However, if you change the name of the log file using the `sh_command_log_file` variable after you start the tool, session information might be lost.

Also, dc_shell does not automatically save the designs loaded in memory. To save these designs before exiting, use the `write_file` command. For example,

```
dc_shell> write_file -format ddc -hierarchy -output my_design.ddc
```

To exit dc_shell, do one of the following:

- If you are in the command-line interface, Enter `quit` or `exit`.
- If you are using the GUI, choose File > Exit.
- If you are running Design Compiler in interactive mode, and the tool is busy, press Ctrl+D.

See Also

- Saving Designs and Exiting Design Compiler From the GUI

Getting Help in Design Compiler

Design Compiler provides a variety of user-assistance tools. To learn about getting help in Design Compiler, see

- Getting Help on the Command Line
- Using the Man Page Viewer
- Using Design Vision Help

Getting Help on the Command Line

The following online information resources are available while you are using the Design Compiler tool:

- Command help, which is a list of options and arguments used with a specified dc_shell command, displayed in the Design Compiler shell and in the console log view when the GUI is open
- Man pages displayed in the Design Compiler shell and in the console log view when the GUI is open
- A man page viewer that displays command, variable, and error message man pages that you request while using the GUI

To get a list of all dc_shell commands, enter the command:

```
dc_shell> help
```

To get information about the options available for a specific dc_shell command, enter the command name with the `-help` option:

```
dc_shell> command_name -help
```

To get the man page for a specific dc_shell command or variable, enter

```
dc_shell> man command_or_variable_name
```

Using the Man Page Viewer

In the GUI, you can use an HTML-based browser window to view, search, and print man pages for commands, variables, and error messages. You can browse back and forth between pages you previously viewed the same way you browse Web pages in a Web browser.

To view man pages interactively in the man page viewer,

1. Choose Help > Man Pages.

   The man page viewer appears. The home page displays a list of links for the different man page categories.

2. Select the type of man pages you want to view: Commands, Variables, or Messages.

   A list of man pages appears.

3. Select the man page you want to view.

You can also display man pages in the man page viewer by using the `man` command or the `gui_show_man_page` command on the console command line.

Note:

If you enter the `gui_show_man_page` command in dc_shell when the GUI is closed, the tool automatically opens the GUI and displays the man page in the man page viewer.

See Also

- The “Viewing Man Pages” topic in Design Vision Help

Using Design Vision Help

Design Vision Help is available in the GUI. You can access the Design Vision Help from the Help menu. The Help system contains topics that explain the details of tasks that you can perform. For example, if you need help performing a step in a procedure presented in the user guide, you can find the information you need in Design Vision Help.

Information in Design Vision Help is grouped in the following categories:

- Feature topics – Overviews of Design Vision window components and tools.
- How-to topics – Procedures for accomplishing synthesis and analysis tasks.
- Reference topics – Explanations of views, toolbar buttons, menu commands, and dialog box options.

Note:

Before you can access Design Vision Help from within Design Vision, the Web browser executable file must be listed in your UNIX or Linux path variable.

Design Vision Help is a browser-based HTML Help system designed for viewing in the Firefox and Mozilla Web browsers.

To access online Design Vision Help,

1. Choose Help > Online Help.

   The Web browser appears and displays the Welcome topic for the Design Vision Help.

2. Use the navigation frame (leftmost frame) to find the information you need in one of the following ways:

   - Find the topic in the hierarchical organization of the Help system by clicking Contents and expanding the appropriate books until you find the information you need.
   - Find the topic by its subject by clicking Index and looking for the subject in the alphabetical listing.
   - Search for keywords found in the topic by clicking Search and entering the keywords.

If more than one topic has the words you are searching for, you must select the appropriate topic from a list of topics.

You can view Design Vision Help as a standalone Help system in your Web browser by opening the file named index.html in the online Help directory: $SYNOPSYS/doc/syn/html/dvoh/enhanced.

Design Vision Help makes extensive use of JavaScript and cascading style sheets (CSS). If your browser encounters problems displaying Design Vision Help, open the browser preferences and make sure that JavaScript and style sheets are enabled and that JavaScript is not blocked by your security preferences.

Note:

If you reset browser preferences while the Help system is open, you might need to click the Reload button on the browser's navigation toolbar after you reset the preferences.

You can use the following browsers to view Design Vision Help:

- Firefox
- Mozilla

The default Help browser is Firefox. You can use the `gui_online_browser` variable to control which browser Design Vision uses to display Design Vision Help.

To set the browser for the current GUI session,

- Enter the following command after you start the GUI:

  ```
  dc_shell> set gui_online_browser "browser_name"
  ```

  The *browser_name* can be `firefox` or `mozilla`.

Alternatively, you can use a setup file to set a default Help browser for any GUI session. Create a file named .synopsys_dv_gui.tcl in your home directory, and enter the `set` command in the file:

```
set gui_online_browser "browser_name"
```

Working With Licenses

You need to determine which licenses are in use and know how to obtain and release licenses. You can use dc_shell commands or the GUI to view, check out, and release licenses.

To learn about working with licenses, see

- Listing the Licenses in Use
- Checking Out Licenses
- Enabling License Queuing
- Releasing Licenses

Listing the Licenses in Use

To view the licenses that you currently have checked out, use the `list_licenses` command. For example,

```
dc_shell-topo> list_licences

Licenses in use:
         DC-Expert (3)
         DC-Ultra-Features (3)
         DC-Ultra-Opt (3)
         Design-Compiler
         DesignWare
1
```

To display which licenses are already checked out, use the `license_users` command. For example,

```
dc_shell-topo> license_users

bill@eng1 Design-Compiler
matt@eng2 Design-Compiler, DC-Ultra-Opt
2 users listed.
1
```

You can also view the licenses that are currently checked out by using the GUI. To display current license information,

1. Choose File > Licenses.

   The Application Licenses dialog box appears. The Allocated Licenses list shows the licenses you are currently using. The Available Licenses list shows the licenses you can use.

2. When you finish viewing the licenses information, click Close to close the dialog box.

Checking Out Licenses

When you invoke Design Compiler, the Synopsys Common Licensing software automatically checks out the appropriate license. For example, if you read in an HDL design description, Synopsys Common Licensing checks out a license for the appropriate HDL compiler.

If you know the tools and interfaces you need, you can use the `get_license` command to check out those licenses. This ensures that each license is available when you are ready to use it. By default, only one license is checked out for each feature. After a license is checked out, it remains checked out until you release it or exit dc_shell.

If multiple licenses are required for a multicore run, use the `-quantity` option to specify the total number of licenses needed. If licenses have already been checked out, Design Compiler acquires only the additional licenses needed to bring the total to the specified quantity.

In the following example, Design Compiler checks out a license for the multivoltage feature:

```
dc_shell-topo> get_license Galaxy-MV
```

In the following example, Design Compiler checks out two of the licenses required to run a multicore `compile_ultra` command:

```
dc_shell-topo> get_license -quantity 2 \
               {DC-Expert DC-Ultra-Opt DC-Ultra-Features}
```

You can also check out additional licenses by using the GUI. To check out an additional license,

1. Choose File > Licenses.
2. Select a license in the Available Licenses list.
3. Click Allocate.

The tool checks out a copy of the license if one is available or displays an error message if all the licenses are already taken.

Enabling License Queuing

Design Compiler has a license queuing functionality that allows your application to wait for licenses to become available if all licenses are in use. To enable this functionality, set the SNPSLMD_QUEUE environment variable to true.

When you enable license queuing, Design Compiler displays the following message:

```
Information: License queuing is enabled. (DCSH-18)
```

When you have enabled the license queuing functionality, you might run into a situation where you hold license L1 while waiting for license L2, and another user holds license L2 while waiting for license L1. In that case, both you and the other user might wait forever, unless more licenses become available.

To prevent such situations, use the SNPS_MAX_WAITTIME and the SNPS_MAX_QUEUETIME environment variables. You must set the SNPSLMD_QUEUE environment variable to true before using these two variables.

- The SNPS_MAX_WAITTIME variable specifies the maximum wait time in seconds for the first license that you require, for example, starting dc_shell.
- The SNPS_MAX_QUEUETIME variable specifies the maximum wait time in seconds for checking out subsequent licenses within the same dc_shell process. You use this variable after you have successfully checked out the first license to start dc_shell.

  Consider the following scenario: You have already started Design Compiler and are running a command that requires a DC-Ultra-Features license. The queuing functionality attempts to check out the license within the specified wait time. The default is 28,800 seconds (or eight hours). If the license is still not available after the predefined time, you might see a message similar to the following:

  ```
  Information: Timeout while waiting for feature 'DC-Ultra-Features.'
  (DCSH-17)
  ```

When you run your design through the synthesis flow, the queuing functionality might display other status messages as follows:

```
Information: Started queuing for feature 'HDL-Compiler'. (DCSH-15)
Information: Still waiting for feature 'HDL-Compiler'. (DCSH-16)
Information: Successfully checked out feature 'HDL-Compiler'. (DCSH-14)
```

Releasing Licenses

To release a license that is checked out to you, use the remove_license command. For example,

```
dc_shell> remove_license HDL-Compiler
```

For multicore runs, where multiple licenses might be required for certain features, use the `-keep` option to specify how many licenses should be retained for each feature after the command has completed.

The following example removes some, but not all, of the licenses required for a multicore `compile_ultra` run in Design Compiler:

```
dc_shell-topo> list_licenses
Licenses in use:
        DC-Expert (4)
        DC-Ultra-Features (4)
        DC-Ultra-Opt (4)
        Design-Compiler
1
dc_shell-topo> remove_license -keep 2 \
               {DC-Expert DC-Ultra-Opt DC-Ultra-Features}
dc_shell-topo> list_licenses
Licenses in use:
        DC-Expert (2)
        DC-Ultra-Features (2)
        DC-Ultra-Opt (2)
        Design-Compiler
1
```

To release a license using the GUI,

1. Choose File > Licenses.
2. Select a license in the Allocated Licenses list.
3. Click Release.

Running a Synthesis Flow

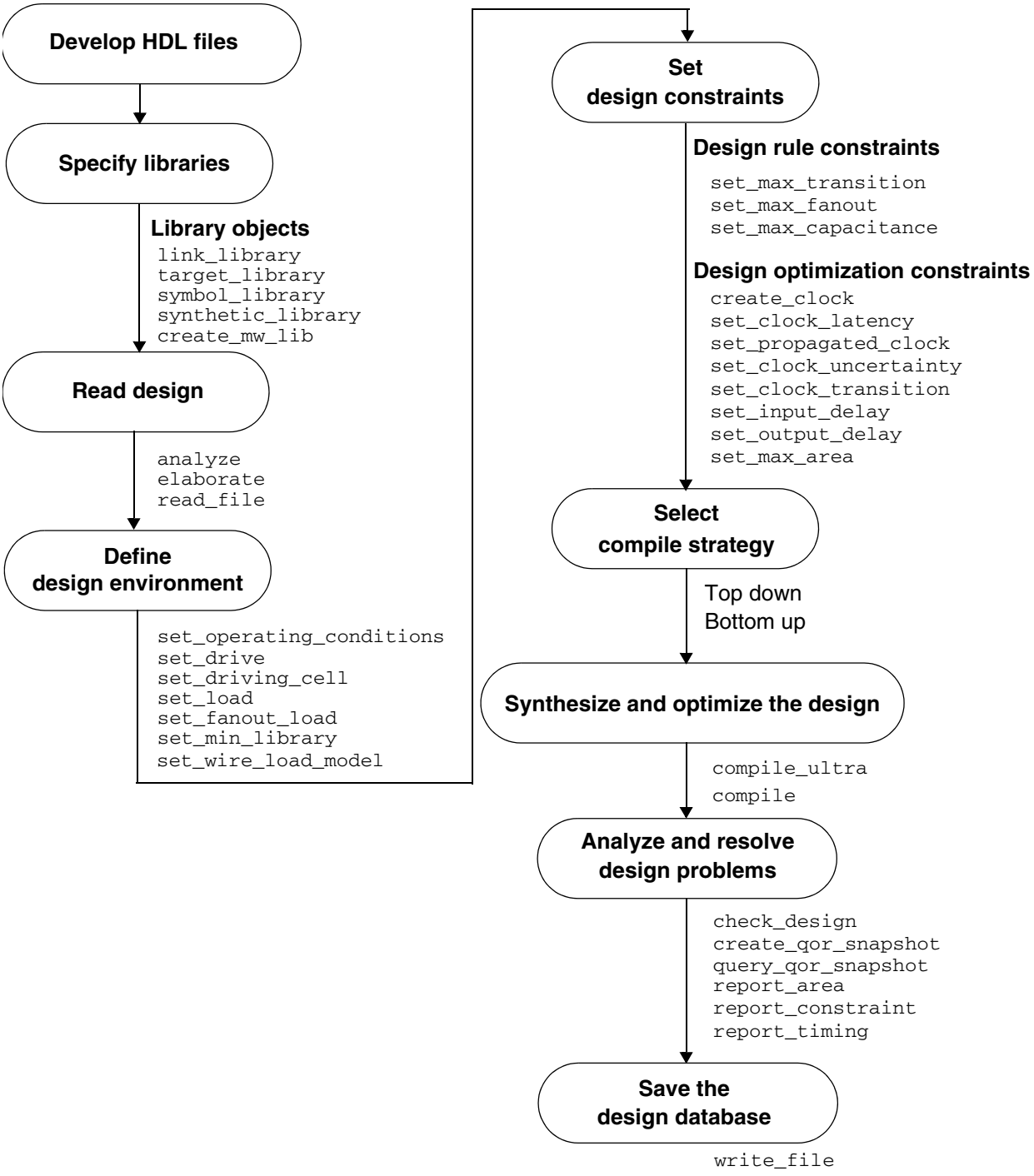
Figure 2-1 shows a basic synthesis flow. You can use this synthesis flow in both the design exploration and design implementation stages of the high-level design flow discussed in Design Compiler in the Design Flow.

The figure lists basic DC Expert and DC Ultra commands that are commonly used in each step of the flow. All the commands shown in the figure can take options, but no options are shown in the figure.

Note:

In the “Select Compile Strategy” step, top down and bottom up are not commands. They refer to two commonly used compile strategies that use different combinations of commands.

Figure 2-1 Basic Synthesis Flow



The following steps provide an overview of a basic synthesis flow:

1. Develop the HDL files.

   The input design files for Design Compiler are often written using a hardware description language (HDL) such as Verilog or VHDL. These design descriptions need to be written carefully to achieve the best synthesis results possible. When writing HDL code, you need to consider design data management, design partitioning, and your HDL coding style. Partitioning and coding style directly affect the synthesis and optimization processes.

   Note:
   This step is included in the flow, but it is not actually a Design Compiler step. You do not create HDL files with the Design Compiler tools.

   For details, see Preparing Design Files for Synthesis.

2. Specify the libraries.

   Specify the link, target, symbol, and synthetic libraries for Design Compiler by using the `link_library`, `target_library`, `symbol_library`, and `synthetic_library` commands. In addition, you must specify any specially licensed DesignWare libraries by using the `synthetic_library` command. You do not need to specify the standard DesignWare library.

   In topographical mode, Design Compiler also requires physical libraries. Use the Milkyway design library to specify physical libraries. You can create a Milkyway design library by using the `create_mw_lib` command.

   For details, see Working With Libraries.

3. Read the design.

   Design Compiler can read both RTL designs and gate-level netlists. Design Compiler uses HDL Compiler to read Verilog and VHDL RTL designs. It also has a specialized netlist reader for reading Verilog and VHDL gate-level netlists, which reads netlists faster and uses less memory than HDL Compiler.

   Design Compiler provides the following ways to read design files:

   - The `analyze` and `elaborate` commands
   - The `read_file` command
   - The `read_vhdl` and `read_verilog` commands

     These commands are derived from the `read_file -format VHDL` and `read_file -format verilog` commands.

   - The `-autoread` option

     When you use the `-autoread` option with the `analyze` or `read_file` command, the tool automatically analyzes and elaborates designs and any files dependent on them

in the correct order. The `read_file -autoread` command analyzes and elaborates the top-level design; the `analyze -autoread` command analyzes the design but does not perform elaboration.

For details, see Working With Designs in Memory.

For information about the recommended reading methods, see the HDL Compiler documentation.

4. Define the design environment.

   Design Compiler requires that you model the environment of the design to be synthesized. This model comprises the external operating conditions (manufacturing process, temperature, and voltage), loads, drives, fanouts, and so on. It directly influences design synthesis and optimization results. If you are not using topographical mode, you need to specify wire load models to estimate the effect of wire length on design performance.

   For details, see Defining the Design Environment.

   In topographical mode,

   - Do not specify wire load models. If you do, Design Compiler ignores them.
   - You can specify power constraints, as described in Power Optimization in Topographical Mode.
   - You can specify test configuration, as described in Test Synthesis in Topographical Mode.
   - You can specify multivoltage designs, as described in Multivoltage Designs.

   For more information about using topographical mode, see Using Design Compiler Topographical Technology.

   Sometimes Design Compiler in topographical mode and IC Compiler have different environment settings. These differences can lead to correlation problems. To help fix correlation issues between Design Compiler in topographical mode and IC Compiler, use the `write_environment` command. For more information, see Comparing Design Compiler Topographical and IC Compiler Environments.

5. Set the design constraints.

   Design Compiler uses design rules and optimization constraints to control the synthesis of the design. Mandatory design rules are provided in the vendor logic library to ensure that the product meets specifications and works as intended. You can also specify additional design rules. Typical design rules constrain transition times, fanout loads, and capacitances. These rules specify technology requirements that you cannot violate. You can, however, specify stricter constraints.

   Optimization constraints define the design goals for timing (clocks, clock skews, input delays, and output delays) and area (maximum area). During the optimization process,

Design Compiler attempts to meet these goals, but no design rules are violated by the process. You define these constraints by using commands such as those listed under this step in Figure 2-1. To optimize a design correctly, you must set realistic constraints.

Note:

Design constraint settings are influenced by the compile strategy you choose. Flow steps 5 and 6 are interdependent. Compile strategies are discussed in step 6.

For details, see Defining Design Constraints.

6. Select the compile strategy.

   The two basic compile strategies that you can use to optimize hierarchical designs are referred to as top down and bottom up.

   In the top-down strategy, the top-level design and all its subdesigns are compiled together. All environment and constraint settings are defined with respect to the top-level design. Although this strategy automatically takes care of interblock dependencies, the method is not practical for large designs because all designs must reside in memory at the same time.

   In the bottom-up strategy, individual subdesigns are constrained and compiled separately. After successful compilation, the designs are assigned the `dont_touch` attribute to prevent further changes to them during subsequent compile phases. Then the compiled subdesigns are assembled to compose the designs of the next higher level of the hierarchy (any higher-level design can also incorporate unmapped logic), and these designs are compiled. This compilation process is continued up through the hierarchy until the top-level design is synthesized. This method lets you compile large designs because Design Compiler does not need to load all the uncompiled subdesigns into memory at the same time. At each stage, however, you must estimate the interblock constraints, and typically you must iterate the compilations, improving these estimates, until all subdesign interfaces are stable.

   Each strategy has its advantages and disadvantages, depending on your particular designs and design goals. You can use either strategy to process the entire design, or you can use a mix of strategies, using the most appropriate strategy for each subdesign.

   Note:

   The compile strategy you choose affects your choice of design constraints and the values you set. Flow steps 5 and 6 are interdependent. Design constraints are discussed in step 5.

   For details, see Optimizing the Design.

7. (Optional) Provide the floorplan information.

   In topographical mode, you can provide a floorplan or floorplan constraints to improve timing correlation with the post-place-and-route tools, such as IC Compiler, by considering floorplanning information during optimization.

If you do not specify a floorplan, Design Compiler creates one for you. However, the principal reason for using floorplan constraints in topographical mode is to accurately represent the placement area and to improve timing correlation with the post-place-and-route design. You can provide high-level physical constraints that determine core area and shape, port location, macro location and orientation, voltage areas, placement blockages, and placement bounds. These physical constraints can be derived from IC Compiler floorplan data, extracted from an existing Design Exchange Format (DEF) file, or created manually.

For details, see Using Floorplan Physical Constraints.

If you provide floorplan information, you can use the GUI layout window to verify that your pre-synthesis floorplan is laid out according to your expectations. The layout view automatically displays floorplan constraints read in with the `extract_physical_constraints` command or with Tcl floorplanning commands. You need to link all applicable designs and libraries to obtain an accurate floorplan.

Before synthesizing your design, check that all designs and libraries have the necessary data for the compilation to run successfully.

8. Check the design with the `compile_ultra -check_only` command.

   If you are using topographical mode, check all the designs and libraries to make sure they have the necessary data for the compilation to run successfully by using the `-check_only` option with the `compile_ultra` command. The `-check_only` option reports potential problems that could cause the tool to stop during the `compile_ultra` run or to produce unsatisfactory correlation with your physical implementation.

   For details, see Checking Designs and Libraries Before Synthesis.

9. Synthesize and optimize the design.

   Use the DC Ultra `compile_ultra` command or the DC Expert `compile` command to run Design Compiler synthesis and optimization. Use the `compile_ultra` command for designs that have significantly tight timing constraints. The `compile_ultra` command is ideal for timing-critical, high performance designs and encapsulates DC Ultra strategies into a single command for better quality of results (QoR).

   Several compile options are available with the `compile_ultra` and `compile` commands. Use the `-scan` option to perform test-ready compilation. During test-ready compilation, the tool employs the scan insertion design-for-test technique, which replaces regular flip-flops with flip-flops that contain logic for testability. Test-ready compilation reduces iterations and design time by accounting for the impact of the scan implementation during the logic optimization process.

   In topographical mode, use the `-timing_high_effort_script` option with the `compile_ultra` command to improve the delay of the design. The strategy can make changes to variables or constraints that modify `compile_ultra` behavior and perform additional passes to achieve better delay. Use the `-gate_clock` option to implement clock gating in the design according to options set by the `set_clock_gating_style`

command. For more information about the `set_clock_gating_style` command, see the *Power Compiler User Guide*.

For details, see Optimizing the Design and Using Design Compiler Topographical Technology.

10. Insert scan chains.

    Run the `insert_dft` command to insert scan chains.

    For details, see the *DFT Compiler Scan User Guide*.

11. Perform incremental synthesis.

    Perform an incremental compile by using the `compile_ultra -incremental -scan` command or the `compile -incremental -scan` command.

    The main goal for `compile_ultra -incremental` is to enable topographical-based optimization for post-topographical-based synthesis flows such as retiming, design-for-test (DFT), DFTMAX, and minor netlist edits.

    For details, see Performing an Incremental Compile.

12. (Optional) Visually inspect the floorplan and placement results.

    In topographical mode, use the GUI layout window to verify your floorplan and placement results. You can

    - Examine the placement and orientation of objects such as macro cells, port locations, and physical constraints
    - Examine the placement of critical timing path objects
    - Analyze floorplan-related congestion and identify the causes of congestion hotspots

    For details, see the *Design Vision User Guide* or Design Vision Help.

13. Analyze and resolve design problems.

    Design Compiler can generate numerous reports, such as area, constraint, and timing reports, on the synthesis and optimization results. You use reports to analyze and resolve any design problems or to improve synthesis results. You can use the `check_design` command to check the synthesized design for consistency. Other `check_` commands are available.

    You can also create a categorized timing report in HTML format by using the `create_qor_snapshot` and `query_qor_snapshot` commands. The report lets you quickly find paths with certain problems, such as large fanouts or transition degradation. You can then modify the constraints and generate a new report based on the constraints you specified.

    For details, see Analyzing and Resolving Design Problems.

14. Save the design.

    Use the `write_file` command to save the synthesized design. Design Compiler does not automatically save designs before exiting. You can write out the design in .ddc, Milkyway, or Verilog format. For details, see Saving Designs and Exiting Design Compiler.

    If you are using topographical mode, see Inputs and Outputs in Design Compiler Topographical Mode.

    You can also save the design attributes and constraints used during synthesis in a script file. Script files are ideal for managing your design attributes and constraints. For details, see the information about using script files in *Using Tcl With Synopsys Tools*.

See Also

- Compile Flows in Topographical Mode

A Design Compiler Session Example

Example 2-2 shows a simple Tcl script that performs a top-down compile run. It uses the basic synthesis flow in topographical mode. The script contains comments that identify each of the steps in the flow. Some of the script command options and arguments have not yet been explained in this manual. Nevertheless, from the previous discussion of the basic synthesis flow, you can begin to understand this example of a top-down compile.

Note:

Only the `set_driving_cell` command is not discussed in the section on basic synthesis design flow. The `set_driving_cell` command is an alternative way to set the external drives on the ports of the design to be synthesized.

Example 2-2 Top-Down Compile Script

```
# Specify the libraries
set_app_var search_path "$search_path ./libraries"
set_app_var link_library "* max_lib.db"
set_app_var target_library "max_lib.db"
create_mw_lib -technology $mw_tech_file \
-mw_reference_library $mw_reference_library $mw_lib_name
open_mw_lib $mw_lib_name

# Read the design
read_verilog rtl.v

# Define the design environment
set_load 2.2 sout
set_load 1.5 cout
set_driving_cell -lib_cell FD1 [all_inputs]
```

```
# Set the optimization constraints
create_clock clk -period 10
set_input_delay -max 1.35 -clock clk {ain bin}
set_input_delay -max 3.5 -clock clk cin
set_output_delay -max 2.4 -clock clk cout
extract_physical_constraints def_file_name

# Map and optimize the design
compile_ultra

# Analyze and debug the design
report_timing

change_names -rules verilog -hierarchy

# Save the design database
write_file -format ddc -hierarchy -output top_synthesized.ddc
write_file -format verilog -hierarchy -output netlist.v
write_sdf sdf_file_name
write_parasitics -output parasitics_file_name
write_sdc sdc_file_name
write_floorplan -all phys_cstr_file_name.tcl
```

You can execute these commands in any of the following ways:

- Enter dc_shell and type each command in the order shown in the example.
- Enter dc_shell and execute a script file by using the `source` command.

  For example, if you are running Design Compiler and the script is in a file called run.scr, you can execute the script file by entering the following command:

  ```
  dc_shell-topo> source run.scr
  ```

- Run the script from the UNIX command line by using the `dc_shell` command with the `-f` option.

  For example, the following command invokes Design Compiler in topographical mode and executes the run.scr script file from the UNIX prompt:

  ```
  % dc_shell -topographical_mode -f run.scr
  ```

Using Multicore Technology

The multicore technology in Design Compiler allows you to use multiple cores to improve the tool runtime. During synthesis, multicore functionality can divide large optimization tasks into smaller tasks for processing on multiple cores.

Multicore technology is only supported in Design Compiler in topographical mode and Design Compiler Graphical. It is not supported in DC Expert.

Enabling Multicore Functionality

To enable multicore functionality in Design Compiler, use the `set_host_options` command. For example, the following command enables the tool to use six cores to run your processes:

```
dc_shell> set_host_options -max_cores 6
```

All `compile_ultra` command options support the use of multiple cores for optimization.

Note:

There might be a slowdown in the performance when the number of available cores is less than the number you specify with `set_host_options -max_cores` because predicting machine load is difficult.

If you are in multicore mode, the log file contains an information message similar to the following message:

```
Information: Running optimization using a maximum of 8 cores. (OPT-1500)
```

Measuring Runtime

When you measure the runtime speedup using multicore optimization, use the wall clock time of the process. The CPU time does not correctly account for multicore runtime speedup.

To report the overall compile wall clock time, run the `report_qor` command, as shown in the following example. The command reports the combined wall clock time, combining the `compile_ultra` and `compile_ultra -incremental` command runs.

```
dc_shell > report_qor

****************************************
Report : qor
...
****************************************
...

  Hostname: machine
  Compile CPU Statistics
  ----------------------------------------
  Resource Sharing:                  21.54
  Logic Optimization:               182.63
  Mapping Optimization:             230.79
  ----------------------------------------
  Overall Compile Time:             631.32
  Overall Compile Wall Clock Time:  288.11
```

You can also check the wall clock time using the clock commands shown in the following example:

```
dc_shell> set_host_options -max_cores 2
dc_shell> set pre_compile_clock [clock seconds]
dc_shell> compile_ultra
dc_shell> set post_compile_clock [clock seconds]
dc_shell> set diff_clock [expr $post_compile_clock - $pre_compile_clock]
```

For additional information about multicore technology, see the `set_host_options` man page.

For information about the `report_qor` command, see Displaying Quality of Results.

Multicorner-Multimode Designs

Designs are often required to operate under multiple modes, such as test or standby mode, and under multiple operating conditions, sometimes referred to as corners. Such designs are known as multicorner-multimode designs. Design Compiler Graphical can analyze and optimize across multiple modes and corners concurrently. The multicorner-multimode feature in Design Compiler Graphical provides compatibility between flows in Design Compiler and IC Compiler.

To define your modes and corners, use the `create_scenario` command. A scenario definition usually includes commands that specify the TLUPlus libraries, operating conditions, and constraints. For details about setting up multicorner-multimode analysis, see Optimizing Multicorner-Multimode Designs in Design Compiler Graphical.

3

Preparing Design Files for Synthesis

Designs (design descriptions) are stored in design files. Design files must have unique names. If a design is hierarchical, each subdesign refers to another design file, which must also have a unique name. However, different design files can contain subdesigns with identical names.

To learn about preparing design files for synthesis, see

- Managing the Design Data
- Partitioning for Synthesis
- HDL Coding for Synthesis

Managing the Design Data

Use systematic organizational methods to manage the design data. Design data control and data organization, as described in the following sections, are two basic elements of managing design data.

- Controlling the Design Data
- Organizing the Design Data

Controlling the Design Data

As new versions of your design are created, you must maintain some archival and record keeping method that provides a history of the design evolution and that lets you restart the design process if data is lost. Establishing controls for data creation, maintenance, overwriting, and deletion is a fundamental design management issue. Establishing file-naming conventions is one of the most important rules for data creation.

Table 3-1 lists the recommended file name extensions for each design data type.

Table 3-1 File Name Extensions

Design data type	Extension	Description
Design source code	.v	Verilog
	.vhd	VHDL
Synthesis scripts	.con	Constraints
	.scr	Script
Reports and logs	.rpt	Report
	.log	Log
Design database	.ddc	Synopsys internal database format

Organizing the Design Data

Establishing and adhering to a method of organizing data is more important than the method you choose. After you place the essential design data under a consistent set of controls, you can organize the data in a meaningful way. To simplify data exchanges and data searches, you should adhere to this data organization system.

You can use a hierarchical directory structure to address data organization issues. Your compile strategy will influence your directory structure. Figure 3-1 shows directory structures based on the top-down compile strategy, and Figure 3-2 shows the bottom-up compile strategy.

Figure 3-1 Top-Down Compile Directory Structure

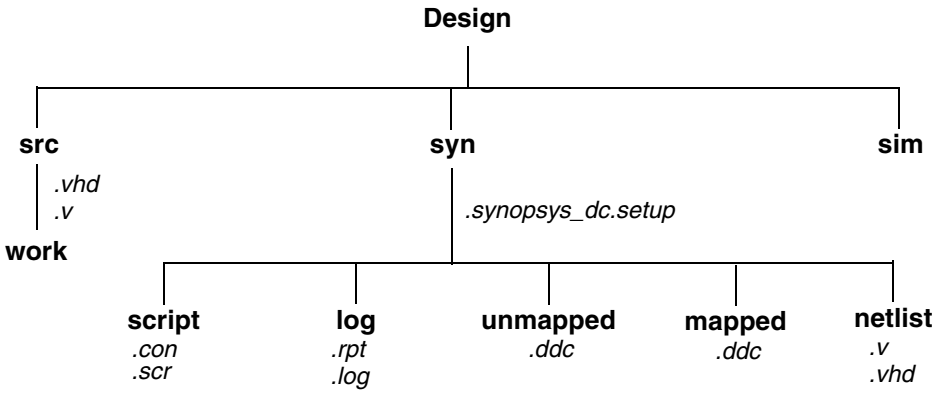
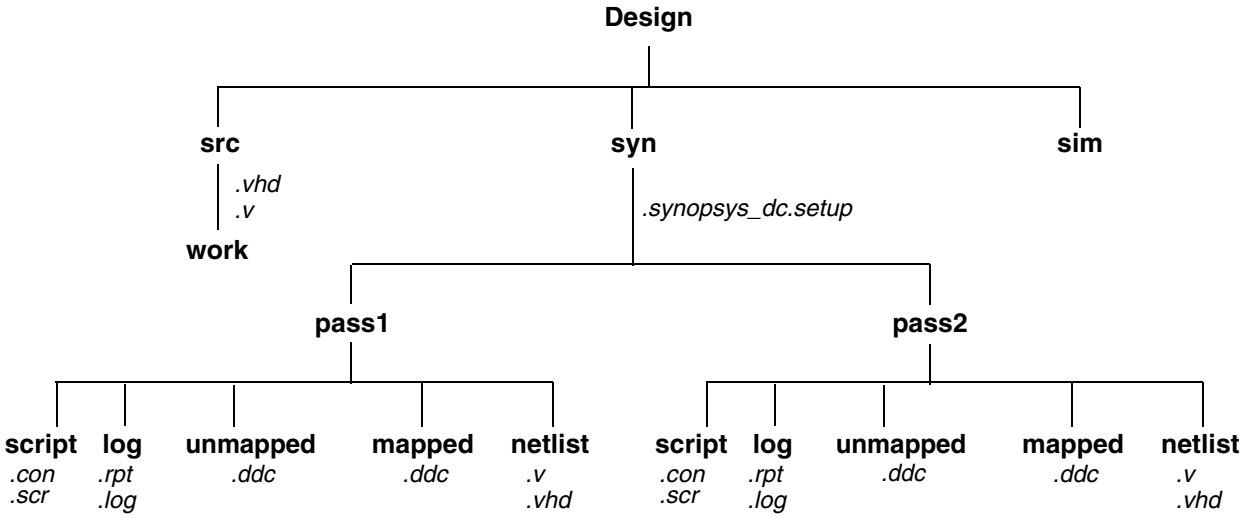


Figure 3-2 Bottom-Up Compile Directory Structure



For details about compile strategies, see Selecting and Using a Compile Strategy.

Partitioning for Synthesis

Partitioning a design effectively can enhance the synthesis results, reduce compile time, and simplify the constraint and script files.

Partitioning affects block size, and although Design Compiler has no inherent block size limit, you should be careful to control block size. If you make blocks too small, you can create artificial boundaries that restrict effective optimization. If you create very large blocks, compile runtimes can be lengthy.

Use the following strategies to partition your design and improve optimization and runtimes:

- Partitioning for Design Reuse
- Keeping Related Combinational Logic Together
- Registering Block Outputs
- Partitioning by Design Goal
- Partitioning by Compile Technique
- Keeping Sharable Resources Together
- Keeping User-Defined Resources With the Logic They Drive
- Isolating Special Functions

Partitioning for Design Reuse

Design reuse decreases time-to-market by reducing the design, integration, and testing effort. When reusing existing designs, partition the design to enable instantiation of the designs.

To enable designs to be reused, follow these guidelines during partitioning and block design:

- Thoroughly define and document the design interface.
- Standardize interfaces whenever possible.
- Parameterize the HDL code.

Keeping Related Combinational Logic Together

By default, Design Compiler cannot move logic across hierarchical boundaries. Dividing related combinational logic into separate blocks introduces artificial barriers that restrict logic optimization.

For best results, apply these strategies:

- Group related combinational logic and its destination register together.

  When working with the complete combinational path, Design Compiler has the flexibility to merge logic, resulting in a smaller, faster design. Grouping combinational logic with its destination register also simplifies the timing constraints and enables sequential optimization.

- Eliminate glue logic.

  Glue logic is the combinational logic that connects blocks. Moving this logic into one of the blocks improves synthesis results by providing Design Compiler with additional flexibility. Eliminating glue logic also reduces compile time, because Design Compiler has fewer logic levels to optimize.

For example, assume that you have a design containing three combinational clouds on or near the critical path. Figure 3-3 shows poor partitioning of this design. Each of the combinational clouds occurs in a separate block, so Design Compiler cannot fully exploit its combinational optimization techniques.

Figure 3-3 Poor Partitioning of Related Logic

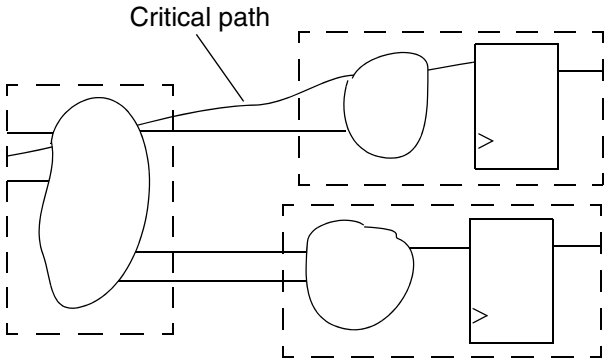
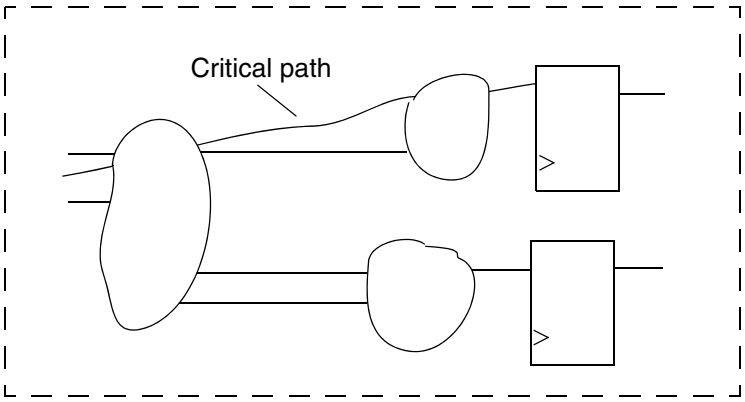


Figure 3-4 shows the same design with no artificial boundaries. In this design, Design Compiler has the flexibility to combine related functions in the combinational clouds.

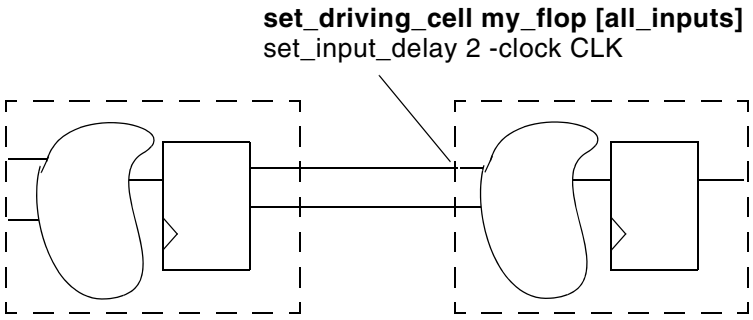
Figure 3-4 Keeping Related Logic in the Same Block



Registering Block Outputs

To simplify the constraint definitions, make sure that registers drive the block outputs, as shown in Figure 3-5.

Figure 3-5 Registering All Outputs



This method enables you to constrain each block easily because

- The drive strength on the inputs to an individual block always equals the drive strength of the average input drive
- The input delays from the previous block always equal the path delay through the flip-flop

Because no combinational-only paths exist when all outputs are registered, time budgeting the design and using the `set_output_delay` command are easier. Given that one clock cycle occurs within each module, the constraints are simple and identical for each module.

This partitioning method can improve simulation performance. With all outputs registered, a module can be described with only edge-triggered processes. The sensitivity list contains only the clock and, perhaps, a reset pin. A limited sensitivity list speeds simulation by having the process triggered only one time in each clock cycle.

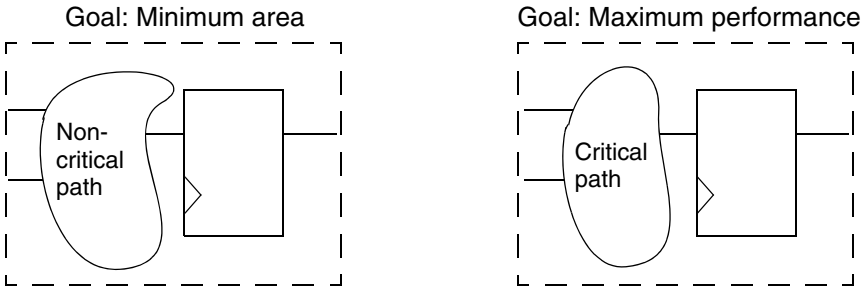
Partitioning by Design Goal

Partition logic with different design goals into separate blocks. Use this method when certain parts of a design are more area and timing critical than other parts.

To achieve the best synthesis results, isolate the noncritical speed constraint logic from the critical speed constraint logic. By isolating the noncritical logic, you can apply different constraints, such as a maximum area constraint, on the block.

Figure 3-6 shows how to separate logic with different design goals.

Figure 3-6 Blocks With Different Constraints



Partitioning by Compile Technique

Partition logic that requires different compile techniques into separate blocks. Use this method when the design contains highly structured logic along with random logic.

- Highly structured logic, such as error detection circuitry, which usually contains large exclusive OR trees, is better suited to structuring.
- Random logic is better suited to flattening.

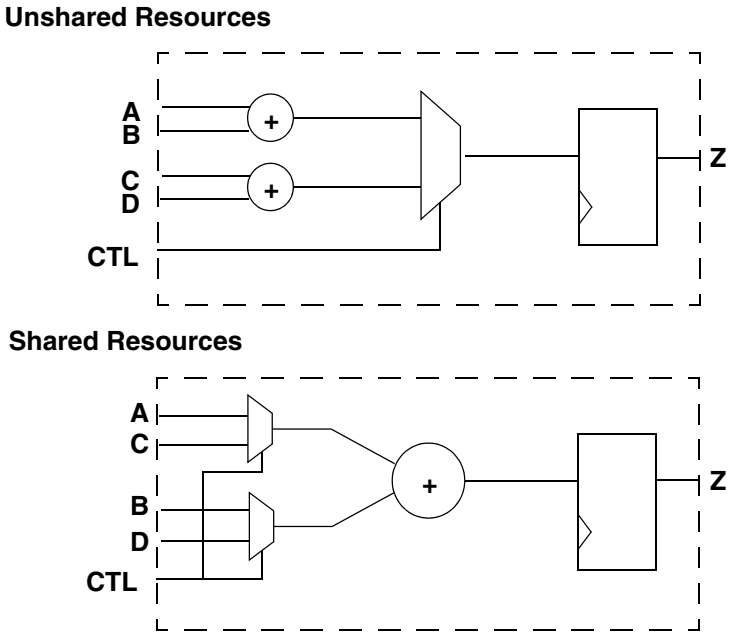
For more information about structuring and flattening, see Logic-Level Optimization.

Keeping Sharable Resources Together

Design Compiler can share large resources, such as adders or multipliers, but resource sharing can occur only if the resources belong to the same VHDL process or Verilog always block.

For example, if two separate adders have the same destination path and have multiplexed outputs to that path, keep the adders in one VHDL process or Verilog always block. This approach allows Design Compiler to share resources (using one adder instead of two) if the constraints allow sharing. Figure 3-7 shows possible implementations of a logic example.

Figure 3-7 Keeping Sharable Resources in the Same Process



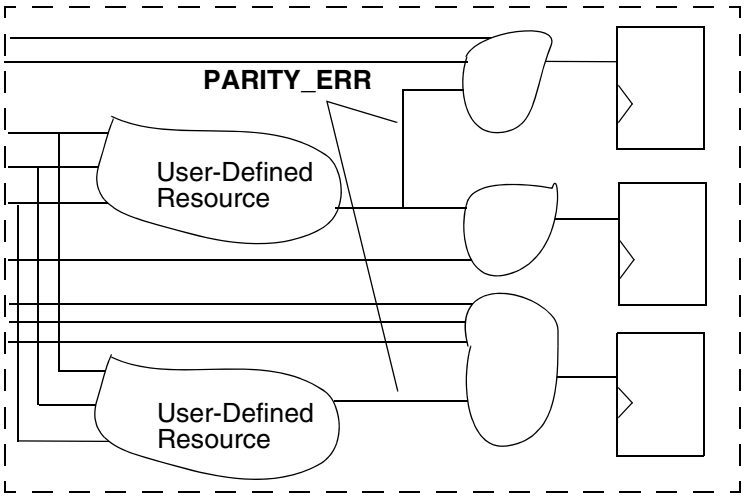
For more information about resource sharing, see the HDL Compiler documentation.

Keeping User-Defined Resources With the Logic They Drive

User-defined resources are user-defined functions, procedures, or macro cells, or user-created DesignWare components. Design Compiler cannot automatically share or create multiple instances of user-defined resources. Keeping these resources with the logic they drive, however, gives you the flexibility to split the load by manually inserting multiple instantiations of a user-defined resource if timing goals cannot be achieved with a single instantiation.

Figure 3-8 illustrates splitting the load by multiple instantiation when the load on the signal PARITY_ERR is too heavy to meet constraints.

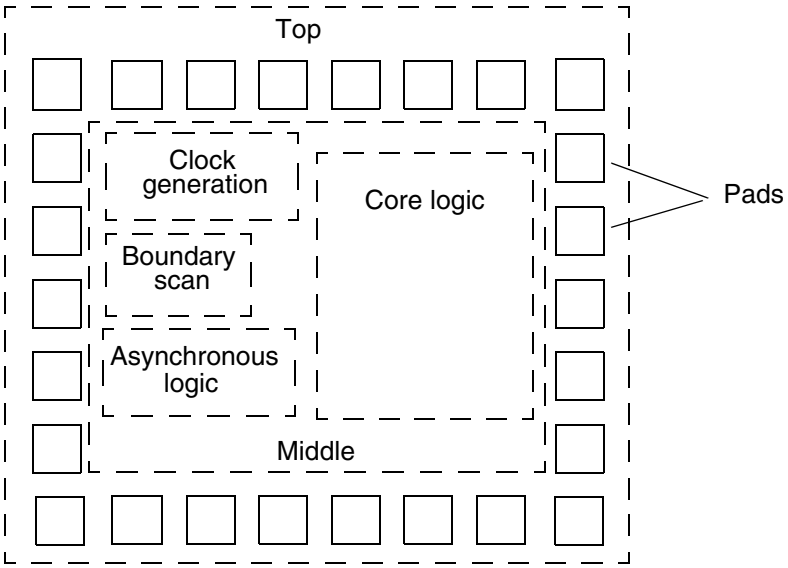
Figure 3-8 Duplicating User-Defined Resources



Isolating Special Functions

Isolate special functions, such as I/O pads, clock generation circuitry, boundary-scan logic, and asynchronous logic from the core logic. Figure 3-9 shows the recommended partitioning for the top level of the design.

Figure 3-9 Recommended Top-Level Partitioning



The top level of the design contains the I/O pad ring and a middle level of hierarchy that contains submodules for the boundary-scan logic, the clock generation circuitry, the asynchronous logic, and the core logic. The middle level of hierarchy exists to allow the flexibility to instantiate I/O pads. Isolation of the clock generation circuitry enables instantiation and careful simulation of this module. Isolation of the asynchronous logic helps confine testability problems and static timing analysis problems to a small area.

HDL Coding for Synthesis

HDL coding is the foundation for synthesis because it implies the initial structure of the design. When writing your HDL source code, always consider the hardware implications of the code. A good coding style can generate smaller and faster designs.

To learn how to write efficient code so that you can achieve your design target in the shortest possible time, see

- Writing Technology-Independent HDL
- Using HDL Constructs
- Writing Effective Code

Writing Technology-Independent HDL

The goal of high-level design that uses a completely automatic synthesis process is to have no instantiated gates or flip-flops. If you meet this goal, you will have readable, concise, and portable high-level HDL code that can be transferred to other vendors or to future processes.

In some cases, the HDL Compiler tool requires compiler directives to provide implementation information while still maintaining technology independence. In Verilog, compiler directives begin with the characters // or /*. In VHDL, compiler directives begin with the two hyphens (--) followed by pragma or synopsys. For more information, see the HDL Compiler documentation.

To learn various methods for keeping your HDL code technology independent, see

- Inferring Components
- Designing State Machines

Inferring Components

HDL Compiler provides the capability to infer the following components:

- Multiplexers
- Registers
- Three-state drivers
- Multibit components

To learn about these inference capabilities, see

- Inferring Multiplexers
- Inferring Registers
- Mixing Register Types
- Inferring Registers Without Control Signals
- Inferring Registers With Control Signals
- Inferring Three-State Drivers
- Inferring Multibit Components

See Also

- The HDL Compiler documentation

Inferring Multiplexers

HDL Compiler can infer a generic multiplexer cell (MUX_OP) from case statements in your HDL code. If your target logic library contains at least a 2-to-1 multiplexer cell, Design Compiler maps the inferred MUX_OPs to multiplexer cells in the target logic library. Design Compiler determines the MUX_OP implementation during compile based on the design constraints. For information about how Design Compiler maps MUX_OPs to multiplexers, see the *Design Compiler Optimization Reference Manual*.

Use the `infer_mux` compiler directive to control multiplexer inference. When attached to a block, the `infer_mux` directive forces multiplexer inference for all case statements in the block. When attached to a case statement, the `infer_mux` directive forces multiplexer inference for that specific case statement.

Inferring Registers

Register inference allows you to specify technology-independent sequential logic in your designs. A register is a simple, 1-bit memory device, either a latch or a flip-flop. A latch is a level-sensitive memory device. A flip-flop is an edge-triggered memory device.

HDL Compiler infers a D latch whenever you do not specify the resulting value for an output under all conditions, as in an incompletely specified if or case statement. HDL Compiler can also infer SR latches and master-slave latches.

HDL Compiler infers a D flip-flop whenever the sensitivity list of a Verilog always block or VHDL process includes an edge expression (a test for the rising or falling edge of a signal). HDL Compiler can also infer JK flip-flops and toggle flip-flops.

Mixing Register Types

For best results, restrict each Verilog always block or VHDL process to a single type of register inferencing: latch, latch with asynchronous set or reset, flip-flop, flip-flop with asynchronous set or reset, or flip-flop with synchronous set or reset.

Be careful when mixing rising- and falling-edge-triggered flip-flops in your design. If a module infers both rising- and falling-edge-triggered flip-flops and the target logic library does not contain a falling-edge-triggered flip-flop, Design Compiler generates an inverter in the clock tree for the falling-edge clock.

Inferring Registers Without Control Signals

For inferring registers without control signals, make the data and clock pins controllable from the input ports or through combinational logic. If a gate-level simulator cannot control the data or clock pins from the input ports or through combinational logic, the simulator cannot initialize the circuit, and the simulation fails.

Inferring Registers With Control Signals

You can initialize or control the state of a flip-flop by using either an asynchronous or a synchronous control signal.

For inferring asynchronous control signals on latches, use the `async_set_reset` compiler directive (attribute in VHDL) to identify the asynchronous control signals. HDL Compiler automatically identifies asynchronous control signals when inferring flip-flops.

For inferring synchronous resets, use the `sync_set_reset` compiler directive (attribute in VHDL) to identify the synchronous controls.

Inferring Three-State Drivers

Assign the high-impedance value (1’bz in Verilog, 'Z' in VHDL) to the output pin to have Design Compiler infer three-state gates. Three-state logic reduces the testability of the design and makes debugging difficult. Where possible, replace three-state buffers with a multiplexer.

Never use high-impedance values in a conditional expression. HDL Compiler always evaluates expressions compared to high-impedance values as false, which can cause the gate-level implementation to behave differently from the RTL description.

For additional information about three-state inference, see the HDL Compiler documentation.

Inferring Multibit Components

Multibit inference allows you to map multiplexers, registers, and three-state drivers to regularly structured logic or multibit library cells. Using multibit components can have the following results:

- Smaller area and delay, due to shared transistors and optimized transistor-level layout
- Reduced clock skew in sequential gates
- Lower power consumption by the clock in sequential banked components
- Improved regular layout of the data path

Multibit components might not be efficient in the following instances:

- As state machine registers
- In small bused logic that would benefit from single-bit design

You must weigh the benefits of multibit components against the loss of optimization flexibility when deciding whether to map to multibit or single-bit components.

Attach the `infer_multibit` compiler directive to bused signals to infer multibit components. You can also change between a single-bit and a multibit implementation after optimization by using the `create_multibit` and `remove_multibit` commands.

For more information about how Design Compiler handles multibit components, see the *Design Compiler Optimization Reference Manual*.

Designing State Machines

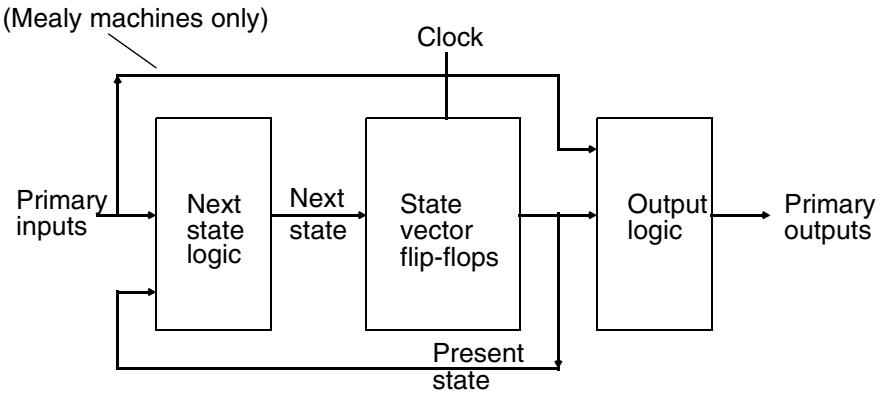
You can specify a state machine by using several different formats:

- Verilog
- VHDL
- State table
- PLA

If you use the `state_vector` and `enum` compiler directives in your HDL code, Design Compiler can extract the state table from a netlist. In the state table format, Design Compiler does not retain the casex, casez, and parallel_case information. Design Compiler does not optimize invalid input combinations and mutually exclusive inputs.

Figure 3-10 shows the architecture for a finite state machine.

Figure 3-10 Finite State Machine Architecture



Using an extracted state table provides the following benefits:

- State minimization can be performed.
- Tradeoffs between different encoding styles can be made.
- Don’t care conditions can be used without flattening the design.
- Don’t care state codes are automatically derived.

For information about extracting state machines and changing encoding styles, see the *Design Compiler Optimization Reference Manual*.

Using HDL Constructs

For information and guidelines about HDL constructs, see

- General HDL Constructs
- Using Verilog Macro Definitions
- Using VHDL Port Definitions

General HDL Constructs

The information in this section applies to both Verilog and VHDL.

Sensitivity Lists

You should completely specify the sensitivity list for each Verilog always block or VHDL process. Incomplete sensitivity lists (shown in the following examples) can result in simulation mismatches between the HDL and the gate-level design.

Example 3-1 Incomplete Sensitivity List (Verilog)

```
always @ (A)
   C <= A | B;
```

Example 3-2 Incomplete Sensitivity List (VHDL)

```
process (A)
   C <= A or B;
```

Value Assignments

Both Verilog and VHDL support the use of immediate and delayed value assignments in the RTL code. The hardware generated by immediate value assignments—implemented by Verilog blocking assignments (=) and VHDL variables (:=)—is dependent on the ordering of the assignments. The hardware generated by delayed value assignments—implemented by Verilog nonblocking assignments (<=) and VHDL signals (<=)—is independent of the ordering of the assignments.

For correct simulation results,

- Use delayed (nonblocking) assignments within sequential Verilog always blocks or VHDL processes
- Use immediate (blocking) assignments within combinational Verilog always blocks or VHDL processes

if Statements

When an if statement used in a Verilog always block or VHDL process as part of a continuous assignment does not include an else clause, Design Compiler creates a latch. The following examples show if statements that generate latches during synthesis.

Example 3-3 Incorrect if Statement (Verilog)

```
if ((a == 1) && (b == 1))
   z = 1;
```

Example 3-4 Incorrect if Statement (VHDL)

```
if (a = '1' and b = '1') then
   z <= '1';
end if;
```

case Statements

If your if statement contains more than three conditions, consider using the case statement to improve the parallelism of your design and the clarity of your code. The following examples use the case statement to implement a 3-bit decoder.

Example 3-5 Using the case Statement (Verilog)

```
case ({a, b, c})
   3'b000: z = 8'b00000001;
   3'b001: z = 8'b00000010;
   3'b010: z = 8'b00000100;
   3'b011: z = 8'b00001000;
   3'b100: z = 8'b00010000;
   3'b101: z = 8'b00100000;
   3'b110: z = 8'b01000000;
   3'b111: z = 8'b10000000;
   default: z = 8'b00000000;
endcase
```

Example 3-6 Using the case Statement (VHDL)

```
case_value :=  a & b & c;
CASE case_value IS
   WHEN "000" =>
      z <= "00000001";
   WHEN "001" =>
      z <= "00000010";
   WHEN "010" =>
      z <= "00000100";
   WHEN "011" =>
      z <= "00001000";
   WHEN "100" =>
      z <= "00010000";
   WHEN "101" =>
      z <= "00100000";
   WHEN "110" =>
      z <= "01000000";
   WHEN "111" =>
      z <= "10000000";
   WHEN OTHERS =>
      z <= "00000000";
END CASE;
```

An incomplete case statement results in the creation of a latch. VHDL does not support incomplete case statements. In Verilog you can avoid latch inference by using either the default clause or the full_case compiler directive.

Although both the full_case directive and the default clause prevent latch inference, they have different meanings. The full_case directive asserts that all valid input values have been specified and no default clause is necessary. The default clause specifies the output for any undefined input values.

For best results, use the default clause instead of the full_case directive. If the unspecified input values are don't care conditions, using the default clause with an output value of x can generate a smaller implementation.

If you use the full_case directive, the gate-level simulation might not match the RTL simulation whenever the case expression evaluates to an unspecified input value. If you use the default clause, simulation mismatches can occur only if you specified don't care conditions and the case expression evaluates to an unspecified input value.

Constant Definitions

Use the Verilog `` `define `` statement or the VHDL constant statement to define global constants. Keep global constant definitions in a separate file. Use parameters (Verilog) or generics (VHDL) to define local constants.

Example 3-7 shows a Verilog code fragment that includes a global `` `define `` statement and a local parameter. Example 3-8 shows a VHDL code fragment that includes a global constant and a local generic.

Example 3-7 Using Macros and Parameters (Verilog)

```
// Define global constant in def_macro.v
`define WIDTH 128

// Use global constant in reg128.v
reg regfile[WIDTH-1:0];

// Define and use local constant in module my_module
module my_module (a, b, c);
   parameter WIDTH=128;
   input [WIDTH-1:0] a, b;
   output [WIDTH-1:0] c;
```

Example 3-8 Using Global Constants and Generics (VHDL)

```
-- Define global constant in synthesis_def.vhd
constant WIDTH : INTEGER := 128;

-- Include global constants
library my_lib;
USE my_lib.synthesis_def.all;

-- Use global constant in entity my_design
entity my_design is
   port (a,b : in std_logic_vector(WIDTH-1 downto 0);
         c: out std_logic_vector(WIDTH-1 downto 0));
end my_design;

-- Define and use local constant in entity my_design
entity my_design is
   generic (WIDTH_VAR : INTEGER := 128);
   port (a,b : in std_logic_vector(WIDTH-1 downto 0);
         c: out std_logic_vector(WIDTH-1 downto 0));
end my_design;
```

Using Verilog Macro Definitions

In Verilog, macros are implemented using the `` `define `` statement. Follow these guidelines for `` `define `` statements:

- Use `` `define `` statements only to declare constants.
- Keep `` `define `` statements in a separate file.
- Do not use nested `` `define `` statements.

  Reading a macro that is nested more than twice is difficult. To make your code readable, do not use nested `` `define `` statements.
- Do not use `` `define `` inside module definitions.

  When you use a `` `define `` statement inside a module definition, the local macro and the global macro have the same reference name but different values. Use parameters to define local constants.

Using VHDL Port Definitions

When defining ports in VHDL source code, observe these guidelines:

- Use the STD_LOGIC and STD_LOGIC_VECTOR packages.

  By using STD_LOGIC, you avoid the need for type conversion functions on the synthesized design.

- Do not use the buffer port mode.

  When you declare a port as a buffer, the port must be used as a buffer throughout the hierarchy. To simplify synthesis, declare the port as an output, then define an internal signal that drives the output port.

Writing Effective Code

For guidelines for writing efficient, readable HDL source code for synthesis, see

- Guidelines for Identifiers
- Guidelines for Expressions
- Guidelines for Functions
- Guidelines for Modules

Guidelines for Identifiers

A good identifier name conveys the meaning of the signal, the value of a variable, or the function of a module; without this information, the hardware descriptions are difficult to read.

Observe the following naming guidelines to improve the readability of your HDL source code:

- Ensure that the signal name conveys the meaning of the signal or the value of a variable without being verbose.

  For example, assume that you have a variable that represents the floating point opcode for rs1. A short name, such as frs1, does not convey the meaning to the reader. A long name, such as floating_pt_opcode_rs1, conveys the meaning, but its length might make the source code difficult to read. Use a name such as fpop_rs1, which meets both goals.

- Use a consistent naming style for capitalization and to distinguish separate words in the name.

  Commonly used styles include C, Pascal, and Modula.

  - C style uses lowercase names and separates words with an underscore, for example, packet_addr, data_in, and first_grant_enable.

  - Pascal style capitalizes the first letter of the name and first letter of each word, for example, PacketAddr, DataIn, and FirstGrantEnable.
  - Modula style uses a lowercase letter for the first letter of the name and capitalizes the first letter of subsequent words, for example, packetAddr, dataIn, and firstGrantEnable.

    Choose one convention and apply it consistently.

- Avoid confusing characters.

  Some characters (letters and numbers) look similar and are easily confused, for example, O and 0 (zero); l and 1 (one).

- Avoid reserved words.
- Use the noun or noun followed by verb form for names, for example, AddrDecode, DataGrant, PCI_interrupt.
- Add a suffix to clarify the meaning of the name.

  Table 3-2 shows common suffixes and their meanings.

Table 3-2 Signal Name Suffixes and Their Meanings

Suffix	Meaning
_clk	Clock signal
_next	Signal before being registered
_n	Active low signal
_z	Signal that connects to a three-state output
_f	Register that uses an active falling edge
_xi	Primary chip input
_xo	Primary chip output
_xod	Primary chip open drain output
_xz	Primary chip three-state output
_xbio	Primary chip bidirectional I/O

Guidelines for Expressions

Observe the following guidelines for expressions:

- Use parentheses to indicate precedence.

  Expression operator precedence rules are confusing, so you should use parentheses to make your expression easy to read. Unless you are using DesignWare resources, parentheses have little effect on the generated logic. An example of a logic expression without parentheses that is difficult to read is

  ```
  bus_select = a ^ b & c~^d|b^~e&^f[1:0];
  ```

- Replace repetitive expressions with function calls or continuous assignments.

  If you use a particular expression more than two or three times, consider replacing the expression with a function or a continuous assignment that implements the expression.

Guidelines for Functions

Observe these guidelines for functions:

- Do not use global references within a function.

  In procedural code, a function is evaluated when it is called. In a continuous assignment, a function is evaluated when any of its declared inputs changes.

  Avoid using references to nonlocal names within a function because the function might not be reevaluated if the nonlocal value changes. This can cause a simulation mismatch between the HDL description and the gate-level netlist.

  For example, the following Verilog function references the nonlocal name byte_sel:

  ```
  function byte_compare;
     input [15:0] vector1, vector2;
     input [7:0] length;

     begin
        if (byte_sel)
           // compare the upper byte
        else
           // compare the lower byte
        ...
     end
  endfunction // byte_compare
  ```

- Be aware that the local storage for tasks and functions is static.

  Formal parameters, outputs, and local variables retain their values after a function has returned. The local storage is reused each time the function is called. This storage can be useful for debugging, but storage reuse also means that functions and tasks cannot be called recursively.

- Be careful when using component implication.

  You can map a function to a specific implementation by using the map_to_module and return_port_name compiler directives. Simulation uses the contents of the function. Synthesis uses the gate-level module in place of the function. When you are using component implication, the RTL model and the gate-level model might be different. Therefore, the design cannot be fully verified until simulation is run on the gate-level design.

  The following functionality might require component instantiation or functional implication:

  - Clock-gating circuitry for power savings
  - Asynchronous logic with potential hazards

    This functionality includes asynchronous logic and asynchronous signals that are valid during certain states.

  - Data-path circuitry

    This functionality includes large multiplexers; instantiated wide banks of multiplexers; memory elements, such as RAM or ROM; and black box macro cells.

    For more information about component implication, see the HDL Compiler documentation.

Guidelines for Modules

Observe these guidelines for modules:

- Avoid using logic expressions when you pass a value through ports.

  The port list can include expressions, but expressions complicate debugging. In addition, isolating a problem related to the bit field is difficult, particularly if that bit field leads to internal port quantities that differ from external port quantities.

- Define local references as generics (VHDL) or parameters (Verilog). Do not pass generics or parameters into modules.

4

Working With Libraries

Design Compiler uses logic, symbol, and synthetic or DesignWare libraries to implement synthesis and to display synthesis results graphically. In topographical mode, Design Compiler also uses Milkyway reference libraries and technology files to obtain physical library information.

To learn about these libraries, how to set up the libraries, and how to carry out simple library commands so that Design Compiler uses the library data correctly, see

Selecting a Semiconductor Vendor

One of the first things you must do when designing a chip is to select the semiconductor vendor and technology you want to use. Consider the following issues during the selection process:

- Maximum frequency of operation
- Physical restrictions
- Power restrictions
- Packaging restrictions
- Clock tree implementation
- Floorplanning
- Back-annotation support
- Design support for libraries, megacells, and RAMs
- Available IP cores
- Available test methods and scan styles

About the Libraries

In wire load mode, Design Compiler uses logic, symbol, and DesignWare libraries. In topographical mode, Design Compiler also uses physical libraries. To learn about these libraries, see

- Logic Libraries
- Symbol Libraries
- DesignWare Libraries
- Physical Libraries

Logic Libraries

Logic libraries contain information about the characteristics and functions of each cell, such as cell names, pin names, area, delay arcs, and pin loading. They also define the conditions that must be met, for example, the maximum transition time for nets. These conditions are called design rule constraints. In addition, a logic library specifies the operating conditions and wire load models for a specific technology.

Design Compiler supports logic libraries that use nonlinear delay models (NLDMs), Composite Current Source (CCS) models (either compact or noncompact), or both NLDM and CCS models. Design Compiler automatically selects the type of timing model to use based on the contents of the logic library. If a library contains both NLDM and CCS models, Design Compiler uses the CCS models. During logic synthesis and preroute optimization, the tool might not use all the available CCS data to save runtime.

Design Compiler requires the logic libraries to be in .db format. In most cases, your semiconductor vendor provides you with .db formatted libraries. If you are provided with only library source code, see the Library Compiler documentation for information about generating logic libraries in .db format.

Design Compiler uses logic libraries for the following purposes:

- Resolving cell references

  The logic libraries that Design Compiler uses to resolve cell references are called link libraries. Link libraries contain the descriptions of library cells and subdesigns in a mapped netlist and can also contain design files. Link libraries include local link libraries defined in the `local_link_library` attribute and system link libraries specified by the `link_library` variable.

- Implementing the design function

  The logic libraries that Design Compiler maps to during optimization are called target libraries. Target libraries contain the cells used to generate the netlist and definitions for the design's operating conditions. The target libraries are the subset of the link libraries that are used to compile or translate a design. Design Compiler saves this information in the design's `local_link_library` attribute.

- Calculating timing values and path delays

  Link libraries define the delay models that are used to calculate timing values and path delays. For information about the various delay models, see the Library Compiler documentation.

- Calculating power consumption

  For information about calculating power consumption, see the *Power Compiler User Guide*.

Link Libraries

For a design to be complete, all cell instances in the design must be linked to the library components and designs that are referenced. This process is called linking the design or resolving references. To resolve references, Design Compiler uses the link libraries set by the following variables and attribute:

- The `link_library` application variable lists the libraries and design files that Design Compiler uses to resolve references.

  Design Compiler searches the files listed in the `link_library` variable from left to right, and it stops searching when it finds a reference. Specifying an asterisk in the `link_library` variable means that Design Compiler searches loaded libraries in memory for the reference. For example, if you set the `link_library` variable to {"*" lsi_10k.db}, Design Compiler searches for the reference in memory first and then in the lsi_10k library.

- The `local_link_library` attribute lists the design files and libraries added to the beginning of the `link_library` variable during the link process. Design Compiler searches files in the `local_link_library` attribute first when it resolves references. You can set this attribute by using the `set_local_link_library` command.

- The `search_path` variable specifies a list of directory paths that the tool uses to find logic libraries and other files when you specify a plain file name without a path. It also sets the paths where Design Compiler can continue the search for unresolved references after it searches the link libraries.

  If Design Compiler does not find the reference in the link libraries, it searches in the directories specified by the `search_path` variable, as described in Specifying a Library Search Path.

Target Libraries

Design Compiler selects functionally correct gates from the target libraries to build a circuit during mapping. It also calculates the timing of the circuit by using the vendor-supplied timing data for these gates.

To specify the target libraries, use the `target_library` variable. Your should specify only the standard cell libraries that you want Design Compiler to use for mapping the standard cells in your design, such as combinational logic and registers. You should not specify any DesignWare libraries or macro libraries, such as pads or memories.

For information about specifying target libraries, see Specifying Logic Libraries.

The Main Library

Design Compiler uses the first logic library found in the `link_library` variable as the main library. It uses the main library to obtain default values and settings used in the absence of explicit specifications for operating conditions, wire load selection group, wire load mode, and net delay calculation. If other libraries have measurement units different from the main library units, Design Compiler converts all units to those specified in the main library. Design Compiler obtains the following default values and settings from the main library:

- Unit definitions
- Operating conditions
- K-factors
- Wire load model selection
- Input and output voltage
- Timing ranges
- RC slew trip points
- Net transition time degradation tables

Symbol Libraries

Symbol libraries contain definitions of the graphic symbols that represent library cells in design schematics. Semiconductor vendors maintain and distribute the symbol libraries.

Design Compiler uses symbol libraries to generate schematic views. You must use Design Vision to view the schematic.

When you generate a schematic, Design Compiler performs a one-to-one mapping of cells in the netlist to cells in the symbol library.

DesignWare Libraries

A DesignWare library is a collection of reusable circuit-design building blocks (components) that are tightly integrated into the Synopsys synthesis environment.

DesignWare components that implement many of the built-in HDL operators are provided by Synopsys. These operators include +, -, *, <, >, <=, >=, and the operations defined by if and case statements.

You can develop additional DesignWare libraries at your site by using DesignWare Developer, or you can license DesignWare libraries from Synopsys or from third parties. To use licensed DesignWare components, you need a license key.

Physical Libraries

In topographical mode, Design Compiler requires physical libraries. The Milkyway database stores design data in the Milkyway design library and physical library data in the Milkyway reference library. You use the Milkyway design library to specify physical libraries and save designs in Milkyway format. The inputs required to create a Milkyway design library are the Milkyway reference library and the Milkyway technology file.

The Milkyway reference library contains the physical representation of standard cells and macros. The Milkyway reference library uses the FRAM abstract view to store information. The reference library also defines the placement unit tile (the width and height of the smallest placeable instance and the routing directions).

The Milkyway technology file (.tf), contains technology-specific information required to route a design. Design Compiler automatically derives routing layer directions if your Milkyway library file is missing this information. Derived routing layer directions are saved in the .ddc file. You can override the derived routing layer direction by using the `set_preferred_routing_direction` command. To report all routing directions, use the `report_preferred_routing_direction` command.

Note:

Because multivoltage designs use power domains, these designs usually require that certain library cells are marked as always-on cells and certain library cell pins are marked as always-on library cell pins. These always-on attributes are necessary to establish any always-on relationships between power domains.

For more information, see the *Power Compiler User Guide*.

Specifying the Libraries

Use the variables listed in Table 4-1 to specify the libraries used by Design Compiler. The table includes the library type and the default file name. The table does not include physical libraries. To specify physical libraries, use the `create_mw_lib` command.

Table 4-1 Library Variables

Library type	Variable	Default	File extension
Link library	`link_library`	`{* your_library.db}`	.db
Target library	`target_library`	`{your_library.db}`	.db
Symbol library	`symbol_library`	`{your_library.sdb}`	.sdb
DesignWare library	`synthetic_library`	`" "`	.sldb

You can also use the Application Setup dialog box in the Design Vision GUI to view, set, or change the library and search path variables for the current session. For more information, see the *Design Vision User Guide* and the “Setting Library Locations” topic in the Design Vision Help.

To learn how to specify libraries, see the following sections:

- Specifying Logic Libraries
- Specifying a Library Search Path
- Specifying DesignWare Libraries
- Specifying Physical Libraries

Specifying Logic Libraries

To specify logic libraries, you must specify the target library and link library. The target library is a subset of the link library. In the following example, the target library is the first link library. The example includes the `additional_link_lib_files` user-defined variable for libraries such as pads and macros.

```
dc_shell-topo> set_app_var target_library [list_of_standard_cell_libraries]
dc_shell-topo> set_app_var synthetic_library [list_of_sldb_files_for_DesignWare]
dc_shell-topo> set additional_link_lib_files [additional_libraries]
dc_shell-topo> set_app_var link_library [list * $target_library \
                 $additional_link_lib_files $synthetic_library]
```

If you are performing technology translation, add the standard cell library for the existing mapped gates to the link libraries and the standard cell library being translated to the target library.

Setting Minimum Timing Libraries

If you are performing simultaneous minimum and maximum timing analysis, the logic libraries specified by the `link_library` variable are used for both maximum and minimum timing information. To specify a separate minimum timing library, use the `set_min_library` command. The `set_min_library` command associates minimum timing libraries with the maximum timing libraries specified in the `link_library` variable. For example,

```
dc_shell-topo> set_app_var link_library {* maxlib.db}
dc_shell-topo> set_min_library maxlib.db -min_version minlib.db
```

To find out which libraries have been set to be the minimum and maximum libraries, use the `list_libs` command. In the generated report, the lowercase letter m appears next to the minimum library and the uppercase letter M appears next to the maximum library.

Specifying a Library Search Path

You can specify the library location by using either the complete path or only the file name. If you specify only the file name, Design Compiler uses the search path defined in the `search_path` variable to locate the library files. By default, the search path includes the current working directory and $SYNOPSYS/libraries/syn, where $SYNOPSYS is the environmental variable that defines the path to the installation directory. Design Compiler looks for the library files, starting with the leftmost directory specified in the `search_path` variable, and uses the first matching library file it finds.

For example, assume that you have logic libraries named my_lib.db in both the lib directory and the vhdl directory. Design Compiler uses the my_lib.db file found in the lib directory because it finds the lib directory first:

```
dc_shell-topo> set_app_var search_path "lib vhdl default"
```

To see the order of the library files as they are found by Design Compiler, use the `which` command:

```
dc_shell-topo> which my_lib.db
/usr/lib/my_lib.db, /usr/vhdl/my_lib.db
```

You can also use the Application Setup dialog box in the Design Vision GUI to view, set, or change the library search path for the current session.

For more information about using the GUI, see the *Design Vision User Guide* and the Design Vision Help.

Specifying DesignWare Libraries

You do not need to specify the standard synthetic library, standard.sldb, which implements the built-in HDL operators. The software automatically uses this library. If you are using additional DesignWare libraries, you must specify these libraries by using the `synthetic_library` variable for optimization and the `link_library` variable for cell references.

For more information about using DesignWare libraries, see the DesignWare documentation.

Specifying Physical Libraries

You use the Milkyway design library to specify physical libraries and save designs in the Milkyway format. The inputs required to create a Milkyway design library are the Milkyway reference library and the Milkyway technology file.

To create a Milkyway design library,

1. Create the Milkyway design library by using the `create_mw_lib` command.

   For example,

   ```
   dc_shell-topo> create_mw_lib -technology $mw_tech_file \
    -mw_reference_library $mw_reference_library $mw_design_library_name
   ```

2. Open the Milkyway library that you created by using the `open_mw_lib` command.

   For example,

   ```
   dc_shell-topo> open_mw_lib $mw_design_library_name
   ```

3. (Optional) Attach the TLUPlus files by using the `set_tlu_plus_files` command.

   For example,

   ```
   dc_shell-topo> set_tlu_plus_files -max_tluplus $max_tlu_file \
      -min_tluplus $min_tlu_file -tech2itf_map $prs_map_file
   ```

4. In subsequent sessions, use the `open_mw_lib` command to open the Milkyway library. If you are using the TLUPlus files for RC estimation, use the `set_tlu_plus_files` command to attach these files.

   For example,

   ```
   dc_shell-topo> open_mw_lib $mw_design_library_name
   dc_shell-topo> set_tlu_plus_files -max_tluplus $max_tlu_file \
      -min_tluplus $min_tlu_file -tech2itf_map $prs_map_file
   ```

The following Milkyway library commands are also supported: `copy_mw_lib`, `close_mw_lib`, `report_mw_lib`, `current_mw_lib`, and `check_tlu_plus_files`.

For more information about using TLUPlus files to provide more accurate capacitance and resistance data, see Using TLUPlus Files for RC Estimation.

For guidelines for using the Milkyway database, see Using a Milkyway Database.

Working With the Libraries

You can perform the following tasks by using simple library commands:

- Loading Libraries
- Listing Libraries
- Reporting Library Contents
- Specifying Library Objects
- Excluding Cells From the Target Libraries
- Verifying Library Consistency
- Removing Libraries From Memory

Loading Libraries

Design Compiler uses binary libraries (.db format for logic libraries and .sdb format for symbol libraries) and automatically loads these libraries when needed. To manually load a binary library, use the `read_file` command:

```
dc_shell-topo> read_file my_lib.db
dc_shell-topo> read_file my_lib.sdb
```

If your library is not in the appropriate binary format, use the `read_lib` command to compile the library source. The `read_lib` command requires a Library-Compiler license.

Listing Libraries

Design Compiler refers to a library loaded in memory by its name. The library statement in the library source defines the library name. To list the names of the libraries loaded in memory, use the `list_libs` command:

```
dc_shell-topo> list_libs
Logical Libraries:
Library          File          Path
-------          ----          ----
my_lib           my_lib.db     /synopsys/libraries
my_symbol_lib    my_lib.sdb    /synopsys/libraries
```

Reporting Library Contents

To report the contents of a library, use the `report_lib` command. The command reports the following information:

- Library units
- Operating conditions
- Wire load models
- Cells (including cell exclusions, preferences, and other attributes)

Specifying Library Objects

Library objects are the vendor-specific cells and their pins. To specify library objects, use the following naming convention:

[*file*:]*library*/*cell*[/*pin*]

where *file* is the file name of a logic library, *library* is the name of a library loaded in memory, *cell* is a library cell, and *pin* is a cell's pin. If you have multiple libraries loaded in memory with the same name, you must specify the file name.

For example, to set the `dont_use` attribute on the AND4 cell in the my_lib library, enter

```
dc_shell-topo> set_dont_use my_lib/AND4
```

To set the `disable_timing` attribute on the Z pin of the AND4 cell in the my_lib library, enter

```
dc_shell-topo> set_disable_timing [get_pins my_lib/AND4/Z]
```

Excluding Cells From the Target Libraries

When Design Compiler maps a design to a logic library, it selects library cells from this library. To specify cells in the target library to be excluded during optimization, use the `set_dont_use` command. For example, to prevent Design Compiler from using the INV_HD high-drive inverter, enter

```
dc_shell-topo> set_dont_use MY_LIB/INV_HD
```

This command affects only the copy of the library that is currently loaded in memory and has no effect on the version that exists on disk. However, if you save the library, the exclusions are saved, and the cells are permanently excluded.

To remove the `dont_use` attribute set by the `set_dont_use` command, use the `remove_attribute` command. For example,

```
dc_shell-topo> remove_attribute MY_LIB/INV_HD dont_use
MY_LIB/INV_HD
```

You can also restrict optimization of a design block by a subset of the target library by using the `set_target_library_subset` command. You can restrict the library cells to be used in a particular block or filter the target library cells on a block-by-block basis.

For more information, see Specifying Target Library Subsets.

Verifying Library Consistency

Consistency between the logic library and the physical library is critical to achieving good results. Before you process your design, make sure that your libraries are consistent by running the `check_library` command.

```
dc_shell-topo> check_library
```

The `check_library` command performs the following checks:

- Integrity of individual logical and physical libraries
- Consistency between logic libraries
- Consistency between logic libraries and physical libraries
- Consistency between physical libraries and technology files

By default, the `check_library` command performs consistency checks between the logic libraries specified in the `link_library` variable and the physical libraries in the current Milkyway design library. You can also explicitly specify logic libraries by using the `-logic_library_name` option or Milkyway reference libraries by using the

-mw_library_name option. If you explicitly specify libraries, these override the default libraries.

You can use the set_check_library_options command to set options for the check_library command to perform various logic library and physical library checks, such as the bus naming style, area of each cell, and so forth.

To see the enabled library consistency checks, run the report_check_library_options -logic_vs_physical command.

Removing Libraries From Memory

To remove libraries from dc_shell memory, use the remove_design command. If you have multiple libraries with the same name loaded into memory, you must specify both the path and the library name. To see the path for each library in memory, use the list_libs command.

Saving Libraries

The write_lib command saves (writes to disk) a compiled library in the Synopsys database or VHDL format. To write out the shell commands to save the current link library settings for design instances, use the write_link_library command.

Specifying Cell Preferences

When Design Compiler maps a design to a logic library, it selects components (library cells) from that library. You can influence the choice of components by excluding cells from the target libraries and by specifying cell preferences.

To specify preferred cells, use the set_prefer command. You can use this command with or without the -min option. If you want Design Compiler to prefer certain cells during the initial mapping of the design, use the command without the -min option.

- Set the preferred attribute on particular cells to override the default cell identified by the library analysis step. This step occurs at the start of compilation to identify the starting cell size for the initial mapping.
- Set the preferred attribute on cells if you know the preferred starting size of the complex cells or the cells with complex timing arcs (such as memories and banked components).

You do not normally need to set the preferred attribute as part of your regular compile methodology because a good starting cell is automatically determined during the library analysis step.

Because nonpreferred gates can be chosen to meet optimization constraints, the effect of preferred attributes might not be noticeable after optimization.

For example, to set a preference for the low-drive inverter INV_LD, enter

```
dc_shell-topo> set_prefer MY_LIB/INV_LD
1
```

To remove cell preferences, use the `remove_attribute` command:

```
dc_shell-topo> remove_attribute MY_LIB/INV_LD preferred
MY_LIB/INV_LD
```

If you want Design Compiler to prefer fewer (but larger-area) buffers or inverters when it fixes hold time violations, use the `-min` option. Normally, Design Compiler gives preference to smaller cell area over the number of cells used in a chain of buffers or inverters. You can change this preference by using the `-min` option, which tells Design Compiler to minimize the number of buffers or inverters by using larger area cells.

For example, to set a `hold_preferred` attribute for the inverter IV, enter

```
dc_shell-topo> set_prefer -min class/IV
1
```

To remove the `hold_preferred` cell attribute, use the `remove_attribute` command.

```
dc_shell-topo> remove_attribute class/IV hold_preferred
class/IV
```

See Also

- Excluding Cells From the Target Libraries

Library-Aware Mapping and Synthesis

You can characterize (or analyze) your target logic library and create a pseudolibrary called ALIB, which has mappings from Boolean functional circuits to actual gates from the target library. Design Compiler reads the ALIB file during compile. The ALIB file provides Design Compiler with greater flexibility and a larger solution space to explore tradeoffs between area and delay during optimization. You must use the `compile_ultra` command to get the benefits from the ALIB library.

Library characterization occurs during the initial stage of compile. Because it can take 5-10 minutes for Design Compiler to characterize each logic library, it is recommended that you generate the ALIB file when you install Design Compiler, and store it in a single repository so that multiple users can share the library.

Generating the ALIB file

To generate the ALIB library corresponding to your target logic library, use the `alib_analyze_libs` command. The tool creates a release-specific subdirectory in the location specified by the `alib_library_analysis_path` variable and stores the generated ALIB files in this directory. For example, the following sequence of commands creates the x.db.alib file for the target library x.db and stores it in /remote/libraries/alib/alib-51.

```
dc_shell-topo> set_app_var target_library "x.db"
dc_shell-topo> set_app_var link_library "x.db"
dc_shell-topo> set_app_var alib_library_analysis_path "/remote/libraries/alib"
dc_shell-topo> alib_analyze_libs
```

Design Compiler creates the alib-51 directory for version control; each release has a different version.

Using the ALIB library

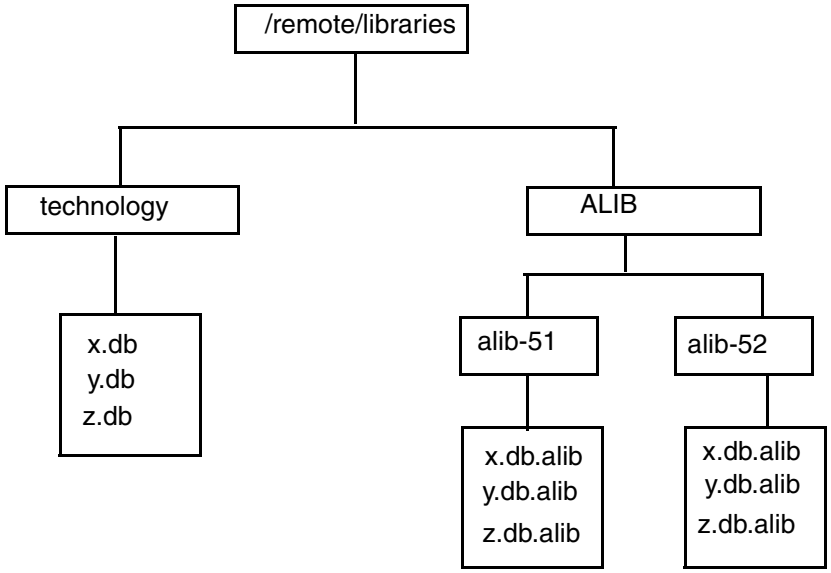
To load the previously generated ALIB library, use the `alib_library_analysis_path` variable to point to the location of the file. For example,

```
dc_shell-topo> set alib_library_analysis_path "remote/libraries/alib"
```

During compile, if no pre-generated ALIB libraries exist, Design Compiler performs library characterization automatically. It generates the ALIB library in the location specified by the `alib_library_analysis_path` variable. If you have not set this variable, the ALIB library is stored in the current working directory. The tool uses this ALIB library for subsequent runs.

It is recommended that you generate the ALIB library for your target logic library when you install Design Compiler. Create a directory structure similar to the one you use for storing technology libraries as shown in Figure 4-1. If you install a new version of Design Compiler, you must regenerate the ALIB library.

Figure 4-1 Directory Structure for ALIB Library



5

Working With Designs in Memory

Design Compiler reads designs into memory from design files. Many designs can be in memory at any time. After a design is read in, you can change it in numerous ways, such as grouping or ungrouping its subdesigns or changing subdesign references.

To learn how to work with designs in memory, see

- Translating Designs From One Technology to Another
- Removing Designs From Memory
- Saving Designs
- Working With Attributes

About Designs

Designs are circuit descriptions that perform logical functions. Designs are described in various design formats, such as VHDL or Verilog HDL. Logic-level designs are represented as sets of Boolean equations. Gate-level designs, such as netlists, are represented as interconnected cells.

Designs can exist and be compiled independently of one another, or they can be used as subdesigns in larger designs. Designs are flat or hierarchical:

- Flat Designs

  Flat designs contain no subdesigns and have only one structural level. They contain only library cells.

- Hierarchical Designs

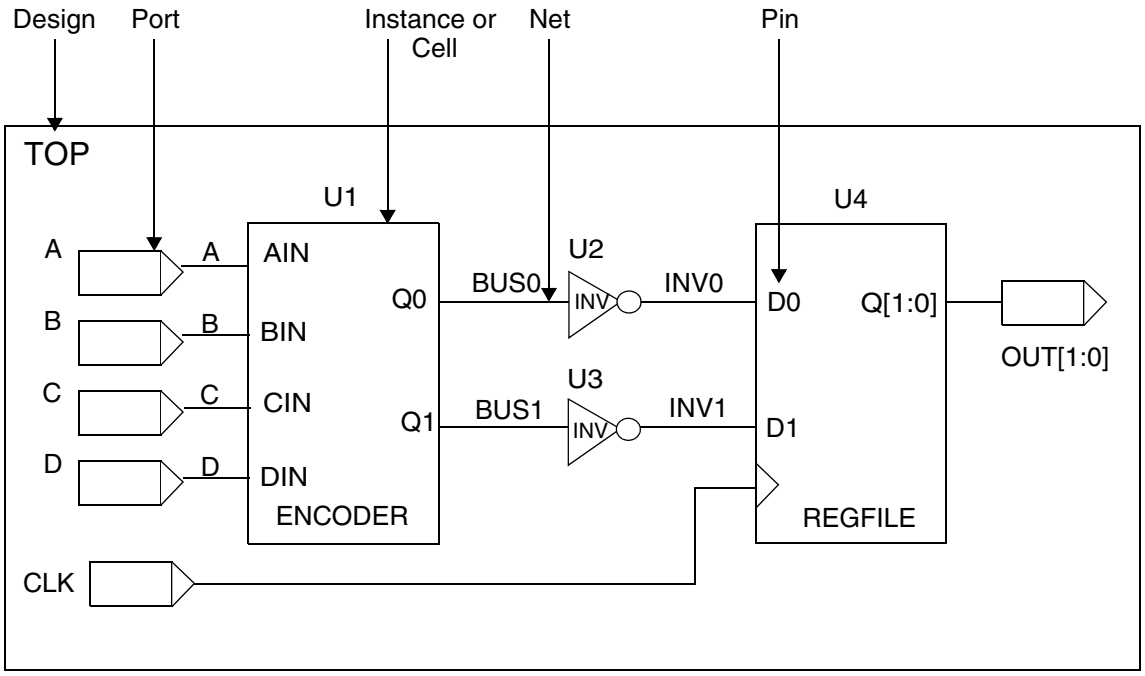
  A hierarchical design contains one or more designs as subdesigns. Each subdesign can further contain subdesigns, creating multiple levels of design hierarchy. Designs that contain subdesigns are called parent designs.

Design Objects

Different companies use different terminology for designs and their components. The following sections describe the terminology used in the Synopsys synthesis tools.

Figure 5-1 shows the design objects in a design called TOP. Synopsys commands, attributes, and constraints are directed toward specific design objects.

Figure 5-1 Design Objects



Design: {TOP, ENCODER, REGFILE}

Reference: {ENCODER, REGFILE, INV}

Instance: {U1, U2, U3, U4}

Design

A design consists of instances, nets, ports, and pins. It can contain subdesigns and library cells. In Figure 5-1, the designs are TOP, ENCODER, and REGFILE. The active design (the design being worked on) is called the current design. Most commands are specific to the current design, that is, they operate within the context of the current design.

Reference

A reference is a library component or design that can be used as an element in building a larger circuit. The structure of the reference can be a simple logic gate or a more complex design (a RAM core or CPU). A design can contain multiple occurrences of a reference; each occurrence is an instance.

References enable you to optimize every cell (such as a NAND gate) in a single design without affecting cells in other designs. The references in one design are independent of the same references in a different design. In Figure 5-1, the references are INV, ENCODER, and REGFILE.

Instance or Cell

An instance is an occurrence in a circuit of a reference (a library component or design) loaded in memory; each instance has a unique name. A design can contain multiple instances; each instance points to the same reference but has a unique name to distinguish it from other instances. An instance is also known as a cell.

A unique instance of a design within another design is called a hierarchical instance. A unique instance of a library cell within a design is called a leaf cell. Some commands work within the context of a hierarchical instance of the current design. The current instance defines the active instance for these instance-specific commands. In Figure 5-1, the instances are U1, U2, U3, and U4.

Ports

Ports are the inputs and outputs of a design. The port direction is designated as input, output, or inout.

Pins

Pins are the input and output of cells (such as gates and flip-flops) within a design. The ports of a subdesign are pins within the parent design.

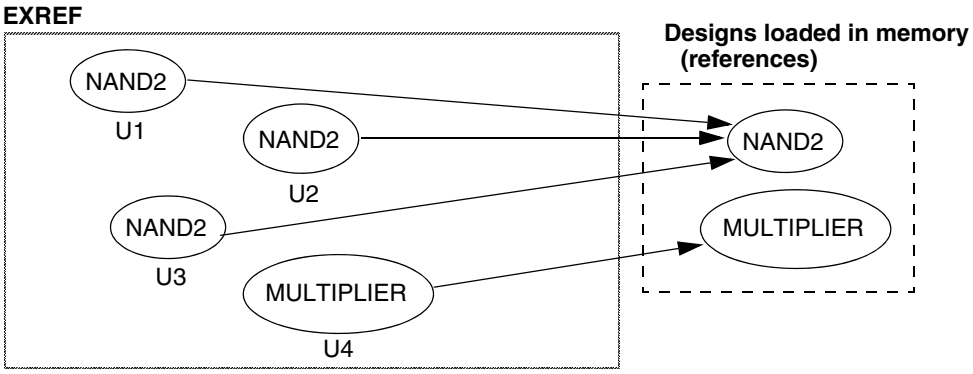
Nets

Nets are the wires that connect ports to pins and pins to each other.

Relationship Between Designs, Instances, and References

Figure 5-2 shows the relationships among designs, instances, and references.

Figure 5-2 Instances and References



The EXREF design contains two references: NAND2 and MULTIPLIER. NAND2 is instantiated three times, and MULTIPLIER is instantiated one time.

The names given to the three instances of NAND2 are U1, U2, and U3. The references of NAND2 and MULTIPLIER in the EXREF design are independent of the same references in different designs.

For information about resolving references, see Linking Designs.

Reporting References

You can use the `report_reference` command to report information about all references in the current instance or current design. Use the `-hierarchy` option to display information across the hierarchy in the current instance or current design.

Using Reference Objects

When you use the `get_references` command, Design Compiler returns a collection of instances that have the specified reference, and you operate on the instances.

For example, the following command returns a collection of instances in the current design that have the reference AN2:

```
dc_shell> get_references AN2
{U2 U3 U4}
```

To see the reference names, use the following command:

```
dc_shell> report_cell [get_references AN*]

Cell                      Reference       Library         Area
Attributes
--------------------------------------------------------------------------
U2                        AN2             lsi_10k         2.000000
U3                        AN2             lsi_10k         2.000000
U4                        AN2             lsi_10k         2.000000
U8                        AN3             lsi_10k         2.000000
```

Reading Designs

Design Compiler can read designs in the formats listed in Table 5-1.

Table 5-1 Supported Input Formats

Format	Description
.ddc	Synopsys internal database format (recommended)
.db	Synopsys internal database format
equation	Synopsys equation format
pla	Berkeley (Espresso) PLA format
st	Synopsys State Table format
Verilog	IEEE standard Verilog (see the HDL Compiler documentation)
VHDL	IEEE standard VHDL (see the HDL Compiler documentation)

Commands for Reading Design Files

Design Compiler provides the following ways to read design files:

- Using the analyze and elaborate Commands
- Using the read_file Command

Using the analyze and elaborate Commands

The `analyze` command does the following:

- Reads an HDL source file
- Checks it for errors (without building generic logic for the design)
- Creates HDL library objects in an HDL-independent intermediate format
- Stores the intermediate files in a location you define

If the `analyze` command reports errors, fix them in the HDL source file and run `analyze` again. After a design is analyzed, you must reanalyze it only when you change it.

Use options to the `analyze` command as follows:

Table 5-2 Using the analyze Command Options

To do this	Use this
Store design elements in a library other than the work library.	`-library` By default, the `analyze` command stores all output in the work library.
Specify the format of the files to be analyzed.	`-format`
Specify a list of files to be analyzed.	*`file_list`*

The `elaborate` command does the following:

- Translates the design into a technology-independent design (GTECH) from the intermediate files produced during analysis
- Allows changing of parameter values defined in the source code
- Allows VHDL architecture selection
- Replaces the HDL arithmetic operators in the code with DesignWare components
- Automatically executes the `link` command, which resolves design references

Use options to the `elaborate` command as follows:

Table 5-3 Using the elaborate Command Options

To do this	Use this
Specify the name of the design to be built (the design can be a Verilog module, a VHDL entity, or a VHDL configuration).	`-design_name`
Find the design in a library other than the work library (the default).	`-library`
Specify the name of the architecture.	`-architecture`
Automatically reanalyze out-of-date intermediate files if the source can be found.	`-update`
Specify a list of design parameters.	`-parameters`

For more information about the `analyze` and `elaborate` commands, see the HDL Compiler documentation.

You can also use the GUI to analyze and elaborate designs. The GUI equivalent command for the `analyze` command is File > Analyze, and the GUI equivalent command for the `elaborate` command is File > Elaborate.

For more information about using the GUI, see the *Design Vision User Guide* and the Design Vision Help.

Using the read_file Command

The `read_file` command does the following:

- Reads several different formats
- Performs the same operations as the `analyze` and `elaborate` commands in a single step
- Creates .mr and .st intermediate files for VHDL
- Does not execute the `link` command automatically

  For more information, see Linking Designs.
- Does not create any intermediate files for Verilog

  However, you can have the `read_file` command create intermediate files by setting the `hdlin_auto_save_templates` variable to true.

For designs in memory, Design Compiler uses the naming convention *path_name* / *design*.ddc. The *path_name* argument is the directory from which the original file was read, and the *design* argument is the name of the design. If you later read in a design that has the same file name, Design Compiler overwrites the original design. To prevent this, use the `-single_file` option with the `read_file` command.

If you do not specify the design format, the `read_file` command will infer the format based on the file extension. If no known extension is used, the tool assumes the .ddc format. Supported extensions for automatic inference are not case-sensitive. The `read_file` command can read compressed files for all formats except the .ddc format, which is compressed internally as it is written. To enable the tool to automatically infer the file format for compressed files, use the following naming structure: *`filename.format`*`.gz`.

The following formats are supported:

- ddc format: Uses the .ddc extension
- db format: Uses the .db, .sldb, .sdb, .db.gz, .sldb.gz, and .sdb.gz extensions
- Verilog format: Uses the .v, .verilog, .v.gz, and .verilog.gz extensions

- SystemVerilog: Uses the .sv, .sverilog, .sv.gz, and .sverilog.gz extensions
- VHDL: Uses the .vhd, .vhdl, vhd.gz, and .vhdl.gz extensions

Use options to the `read_file` command as follows:

Table 5-4 Using the read_file Command Options

To do this	Use this
Specify a list of files to be read.	*`file_list`*
Specify the format in which a design is read. You can specify any input format listed in Table 5-1.	`-format`
Store design elements in a library other than the work library (the default) when reading VHDL design descriptions.	`-library`
Read a Milkyway ILM view.	`-ilm`
Specify that the design being read is a structural or gate-level design when reading Verilog or VHDL designs.	`-netlist -format\ verilog|vhdl`[1]
Specify that the design being read is an RTL design when reading Verilog or VHDL designs.	`-rtl -format\ verilog|vhdl`[2]

1. *The -netlist option is optional when you read a Verilog design.*
2. *The -rtl option is optional when you read a Verilog design.*

You can also use the GUI to read in a design. The GUI equivalent command for the `read_file` command is File > Read. For more information, see the Design Vision Help.

Table 5-5 summarizes the differences between using the `read_file` command and using the `analyze` and `elaborate` commands to read design files.

Table 5-5 read_file Versus analyze and elaborate Commands

Comparison	read_file command	analyze and elaborate commands
Input formats	All formats	VHDL, Verilog.
When to use	Netlists, precompiled designs, and so forth	Synthesize VHDL or Verilog.

Table 5-5 read_file Versus analyze and elaborate Commands (Continued)

Comparison	read_file command	analyze and elaborate commands
Generics	Cannot pass parameters (must use directives in HDL)	Allows you to set parameter values on the `elaborate` command line. Thus for parameterized designs, you can use the `analyze` and `elaborate` commands to build a new design with nondefault values.
Architecture	Cannot specify the architecture to be elaborated	Allows you to specify architecture to be elaborated.
Linking designs	Must use the `link` command to resolve references	The `elaborate` command executes the `link` command automatically to resolve references.

Reading HDL Designs

Use one of the following methods to read HDL design files:

- The `analyze` and `elaborate` commands

  To use this method, analyze the top-level design and all subdesigns in bottom-up order and then elaborate the top-level design and any subdesigns that require parameters to be assigned or overwritten.

  For example, enter

```
dc_shell> analyze -format vhdl -lib -work RISCTYPES.vhd
dc_shell> analyze -format vhdl -lib -work {ALU.vhd STACK_TOP.vhd \
                  STACK_MEM.vhd...}
dc_shell> elaborate RISC_CORE -arch STRUCT -lib WORK -update
```

- The `read_file` command

  For example, enter

```
dc_shell> read_file -format verilog RISC_CORE.v
```

- The `read_verilog` or `read_vhdl` command

  For example, enter

```
dc_shell> read_verilog RISC_CORE.v
```

  You can also use the `read_file -format VHDL` and `read_file -format verilog` commands.

Reading .ddc Files

To read the design data from a .ddc file, use the `read_ddc` command or the `read_file -format ddc` command. For example,

```
dc_shell> read_ddc design_file.ddc
```

Note:

The .ddc format is backward compatible (you can read a .ddc file that was generated with an earlier software version) but not forward compatible (you cannot read a .ddc file that was generated with a later software version).

Reading .db Files

Although you can use the .db format, it is recommended that you use the .ddc format. To read in a .db file, use the `read_db` command or the `read_file -format db` command.

For example,

```
dc_shell> read_db design_file.db
```

The version of a .db file is the version of Design Compiler that created the file. For a .db file to be read into Design Compiler, its file version must be the same as or earlier than the version of Design Compiler you are running. If you attempt to read in a .db file generated by a Design Compiler version that is later than the Design Compiler version you are using, an error message appears. The error message provides details about the version mismatch.

Listing Designs in Memory

To list the names of the designs loaded in memory, use the `list_designs` command.

```
dc_shell> list_designs
A (*)    B    C
1
```

The asterisk (*) next to design A shows that A is the current design.

To list the memory file name corresponding to each design name, use the `-show_file` option.

```
dc_shell> list_designs -show_file

/user1/designs/design_A/A.ddc
A (*)

/home/designer/dc/B.ddc
B       C
```

```
1
```

The asterisk (*) next to design A shows that A is the current design. File B.ddc contains both designs B and C.

Setting the Current Design

You can set the current design (the design you are working on) in the following ways:

- With the `read_file` command

  When the `read_file` command successfully finishes processing, it sets the current design to the design that was read in.

  ```
  dc_shell> read_file -format ddc MY_DESIGN.ddc
  Reading ddc file '/designs/ex/MY_DESIGN.ddc'
  Current design is 'MY_DESIGN'
  ```

- With the `elaborate` command
- With the `current_design` command

  Use this command to set any design in dc_shell memory as the current design.

  ```
  dc_shell> current_design ANY_DESIGN
  Current design is 'ANY_DESIGN'.
  {ANY_DESIGN}
  ```

To display the name of the current design, enter the following command:

```
dc_shell> printvar current_design
current_design = "test"
```

Using the current_design Command

You should avoid writing scripts that use a large number of `current_design` commands, such as in a loop. Using a large number of `current_design` commands can increase runtime. For more information, see *Using Tcl With Synopsys Tools*.

Several commands accept instance objects—that is, cells at a lower level of hierarchy. You can use the following commands on a hierarchical design from any level in the design without using the `current_design` command:

- Netlist editing commands

  For more information, see Editing Designs.

- The `ungroup`, `group`, and `uniquify` commands

  For more information, see Removing Levels of Hierarchy and Uniquify Method.

- The `set_size_only` command

  The command sets a list of attributes on specified leaf cells so sizing optimizations can be performed on these cells during compile.

- The `change_link` command

  For more information, see Changing Design References.

Linking Designs

For a design to be complete, it must connect to all the library components and designs it references. This process is called linking the design or resolving references.

Design Compiler uses the `link` command to resolve references. The `link` command uses the `link_library` and `search_path` system variables and the `local_link_library` attribute to resolve design references.

Design Compiler resolves references by carrying out the following steps:

1. It determines which library components and subdesigns are referenced in the current design and its hierarchy.
2. It searches the link libraries to locate these references.
   a. Design Compiler first searches the libraries and design files defined in the current design's `local_link_library` attribute.
   b. If an asterisk is specified in the value of the `link_library` variable, Design Compiler searches in memory for the reference.
   c. Design Compiler then searches the libraries and design files defined in the `link_library` variable.
3. If it does not find the reference in the link libraries, it searches in the directories specified by the `search_path` variable.

   For more information, see Locating Designs by Using a Search Path.
4. It links (connects) the located references to the design.

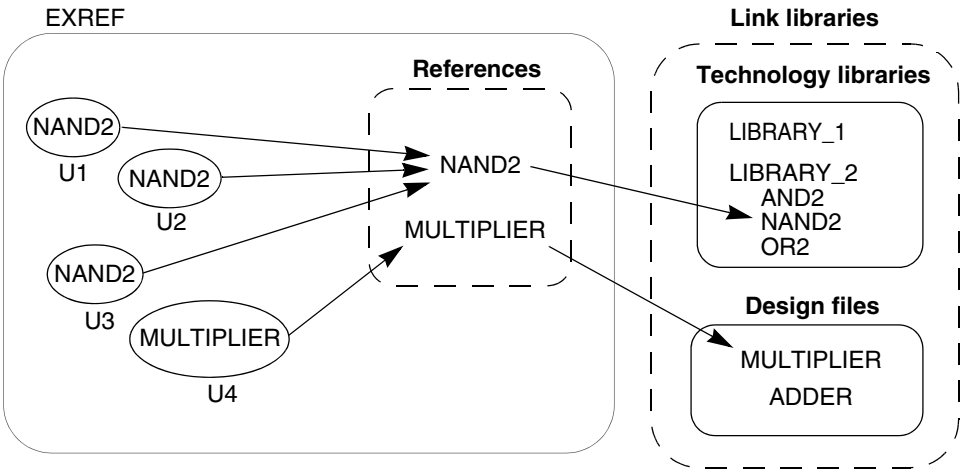
Note:
In a hierarchical design, Design Compiler considers only the top-level design's local link library. It ignores local link libraries associated with the subdesigns.

Design Compiler uses the first reference it locates. If it locates additional references with the same name, it generates a warning message identifying the ignored, duplicate references. If Design Compiler does not find the reference, a warning appears advising that the reference cannot be resolved.

By default, the case sensitivity of the linking process depends on the source of the references. To explicitly define the case sensitivity of the linking process, set the `link_force_case` variable.

The arrows in Figure 5-3 show the connections that the linking process added between the instances, references, and link libraries. In this example, Design Compiler finds library component NAND2 in the LIBRARY_2 logic library; it finds subdesign MULTIPLIER in a design file.

Figure 5-3 Resolving References



Locating Designs by Using a Search Path

You can specify the design file location by using the complete path or only the file name. If you specify only the file name, Design Compiler uses the search path defined in the `search_path` variable. Design Compiler looks for the design files starting with the leftmost directory specified in the `search_path` variable and uses the first design file it finds. By default, the search path includes the current working directory and $SYNOPSYS/libraries/syn, where $SYNOPSYS is the path to the installation directory. To see where Design Compiler finds a file when using the search path, use the `which` command. For example, enter

```
dc_shell> which my_design.ddc
{/usr/designers/example/my_design.ddc}
```

To specify other directories in addition to the default search path, use one following command:

```
dc_shell> lappend search_path project
```

Changing Design References

Use the `change_link` command to change the component or design to which a cell or reference is linked.

- For a cell, the link for that cell is changed.
- For a reference, the link is changed for all cells having that reference.

The link can be changed only to a component or design that has the same number of ports with the same size and direction as the original reference.

When you use `change_link`, all link information is copied from the old design to the new design. If the old design is a synthetic module, all attributes of the old synthetic module are moved to the new link.

After running the `change_link` command, you must run the design with the `link` command.

The `change_link` command accepts instance objects, that is, cells at a lower level in the hierarchy. Additionally, you can use the `-all_instances` option when any cell in the `object_list` is an instance at a lower level in the hierarchy and its parent cell is not unique. All similar cells in the same parent design are automatically linked to the new reference design. You do not have to change the current design to change the link for such cells. If none of the cells in `object_list` is an instance at a lower level of the hierarchy or the parent cell is unique, you do not have to use the `-all_instances` option.

Example 1

The following command shows how cells U1 and U2 are linked from the current design to MY_ADDER:

```
dc_shell> copy_design ADDER MY_ADDER
dc_shell> change_link {U1 U2} MY_ADDER
```

Example 2

The following command changes the link for cell U1, which is at a lower level in the hierarchy:

```
dc_shell> change_link top/sub_inst/U1 lsi_10k/AN3
```

Example 3

This example shows how you can use the `-all_instances` option to change the link for inv1, when its parent design, bot, is instantiated multiple times. The design bot is instantiated twice: mid1/bot1 and mid1/bot2.

```
dc_shell> change_link -all_instances mid1/bot1/inv1 lsi_10k/AN3

Information: Changed link for all instances of cell 'inv1'
in subdesign 'bot'. (UID-193)

dc_shell> get_cells -hierarchical -filter "ref_name == AN3"

{mid1/bot1/inv1 mid1/bot2/inv1}
1
```

Listing Design Objects

Design Compiler provides commands for accessing various design objects. These commands refer to design objects located in the current design. Each command in Table 5-6 performs one of the following actions:

- List

  Provides a listing with minimal information.

- Display

  Provides a report that includes characteristics of the design object.

- Return

  Returns a collection that can be used as input to another dc_shell command.

Table 5-6 lists the commands and the actions they perform.

Table 5-6 Commands to Access Design Objects

Object	Command	Action
Instance	`list_instances`	Lists instances and their references.
	`report_cell`	Displays information about instances.
Reference	`report_reference`	Displays information about references.
Port	`report_port`	Displays information about ports.
	`report_bus`	Displays information about bused ports.
	`all_inputs`	Returns all input ports.
	`all_outputs`	Returns all output ports.
Net	`report_net`	Displays information about nets.
	`report_bus`	Displays information about bused nets.
Clock	`report_clock`	Displays information about clocks.
	`all_clocks`	Returns all clocks.
Register	`all_registers`	Returns all registers.
Collections	`get_*`	Returns a collection of cells, designs, libraries and library cell pins, nets, pins, and ports.

You can also use the GUI to list design objects and attribute values, provide reports, and return collections. To list and display information about designs, cells, and objects, choose List in the menu bar. To generate design object reports, use the Design menu commands.

For more information about using the GUI, see the *Design Vision User Guide* and the Design Vision Help.

Specifying Design Objects

You can specify design objects by using either a relative path or an absolute path:

- Using a Relative Path
- Using an Absolute Path

Using a Relative Path

If you use a relative path to specify a design object, the object must be in the current design. Specify the path relative to the current instance. The current instance is the frame of reference within the current design. By default, the current instance is the top level of the current design. Use the `current_instance` command to change the current instance.

For example, to place a `dont_touch` attribute on hierarchical cell U1/U15 in the Count_16 design, you can enter either

```
dc_shell> current_design Count_16
Current design is 'Count_16'.
{Count_16}
dc_shell> set_dont_touch U1/U15
```

or

```
dc_shell> current_design Count_16
Current design is 'Count_16'.
{Count_16}
dc_shell> current_instance U1
Current instance is '/Count_16/U1'.
/Count_16/U1
dc_shell> set_dont_touch U15
1
```

In the first command sequence, the frame of reference remains at the top level of design Count_16. In the second command sequence, the frame of reference changes to instance U1. Design Compiler interprets all future object specifications relative to instance U1.

To reset the current instance to the top level of the current design, enter the `current_instance` command without an argument.

```
dc_shell> current_instance
Current instance is the top-level of the design 'Count_16'
```

The `current_instance` variable points to the current instance. To display the current instance, enter the following command:

```
dc_shell> printvar current_instance
current_instance = "Count_16/U1"
```

Using an Absolute Path

When you use an absolute path to specify a design object, the object can be in any design in dc_shell memory. Use the following syntax to specify an object by using an absolute path:

[*file*:]*design/object*

file

The path name of a memory file followed by a colon (:). Use the file argument when multiple designs in memory have the same name.

design

The name of a design in dc_shell memory.

object

The name of the design object, including its hierarchical path. If several objects of different types have the same name and you do not specify the object type, Design Compiler looks for the object by using the types allowed by the command.

To specify an object type, use the `get_*` command. For more information about these commands, see *Using Tcl With Synopsys Tools*.

For example, to place a `dont_touch` attribute on hierarchical cell U1/U15 in the Count_16 design, enter

```
dc_shell> set_dont_touch /usr/designs/Count_16.ddc:Count_16/U1/U5
1
```

Creating Designs

The `create_design` command creates a new design. The memory file name is *my_design*.db, and the path is the current working directory.

```
dc_shell> create_design my_design
1
dc_shell> list_designs -show_file

/work_dir/mapped/test.ddc
test (*) test_DW01_inc_16_0 test_DW02_mult_16_16_1

/work_dir/my_design.db
my_design
1
```

Designs created with `create_design` contain no design objects. Use the appropriate create commands (such as `create_clock`, `create_cell`, or `create_port`) to add design objects to the new design. For information about these commands, see Editing Designs.

Copying Designs

The `copy_design` command copies a design in memory and renames the copy. The new design has the same path and memory file as the original design.

```
dc_shell> copy_design test test_new
Information: Copying design /designs/test.ddc:to designs/
test.ddc:test_new
1
dc_shell> list_designs -show_file

/designs/test.ddc
test (*) test_new
```

You can use the `copy_design` command with the `change_link` command to manually create unique instances. For example, assume that a design has two identical cells, U1 and U2, both linked to COMP.

Enter the following commands to create unique instances:

```
dc_shell> copy_design COMP COMP1
Information: Copying design /designs/COMP.ddc:COMP to
     designs/COMP.ddc:COMP1
1

dc_shell> change_link U1 COMP1
Performing change_link on cell 'U1'.
1

dc_shell> copy_design COMP COMP2
Information: Copying design /designs/COMP.ddc:COMP to
                designs/COMP.ddc:COMP2
1

dc_shell> change_link U2 COMP2
Performing change_link on cell 'U2'.
1
```

Renaming Designs

Use the `rename_design` command to rename a design in memory. You can assign a new name to a design or move a list of designs to a file. To save a renamed file, use the `write_file` command. Use options to the `rename_design` command as follows:

Table 5-7 Using the rename_design Command Options

To do this	Use this
Specify a list of designs to be renamed.	*-design_list*
Specify the new name of the design.	*-target_name*
Specify a string to be prefixed to the design name.	-prefix
Specify a string to be appended to the design name.	-postfix
Relink cells to the renamed reference design.	-update_links

In the following example, the `list_designs` command is used to show the design before and after you use the `rename_design` command:

```
dc_shell> list_designs -show_file

/designs/test.ddc
test(*) test_new
1

dc_shell> rename_design test_new test_new_1
Information: Renaming design /designs/test.ddc:test_new to
     /designs/test.ddc:test_new_1
1

dc_shell> list_designs -show_file

/designs/test.ddc
test (*) test_new test_new_1
1
```

You can use the `-prefix`, `-postfix,` and `-update_links` options to rename designs and update cell links for the entire design hierarchy. For example, the following script prefixes the string NEW_ to the name of the design D and updates links for its instance cells:

```
dc_shell> get_cells -hierarchical -filter "ref_name == D"

{b_in_a/c_in_b/d1_in_c b_in_a/c_in_b/d2_in_c}

dc_shell> rename_design D -prefix NEW_ -update_links
```

```
Information: Renaming design /test_dir/D.ddc:D to
                            /test_dir/D.ddc:NEW_D. (UIMG-45)

dc_shell> get_cells -hierarchical -filter "ref_name == D"
# no  such cells!

dc_shell> get_cells -hierarchical -filter "ref_name == NEW_D"
{b_in_a/c_in_b/d1_in_c b_in_a/c_in_b/d2_in_c}
```

Cells `b_in_a/c_in_b/d1_in_c` and `b_in_a/c_in_b/d2_in_c` instantiate design D. After you have run the `rename_design D -prefix NEW -update_links` command, the instances are relinked to the renamed reference design NEW_D.

Changing the Design Hierarchy

When possible, reflect the design partitioning in your HDL description. If your HDL code is already developed, Design Compiler enables you to change the hierarchy without modifying the HDL description.

The `report_hierarchy` command displays the design hierarchy. Use this command to understand the current hierarchy before making changes and to verify the hierarchy changes.

Design Compiler provides the following hierarchy manipulation capabilities:

- Adding Levels of Hierarchy
- Removing Levels of Hierarchy
- Merging Cells From Different Subdesigns

Adding Levels of Hierarchy

Adding a level of hierarchy is called grouping. You can create a level of hierarchy by grouping cells or related components into subdesigns.

Grouping Cells Into Subdesigns

You use the `group` command to group cells (instances) in the design into a new subdesign, creating a new level of hierarchy. The grouped cells are replaced by a new instance (cell) that references the new subdesign.

The ports of the new subdesign are named after the nets to which they are connected in the design. The direction of each port of the new subdesign is determined from the pins of the corresponding net.

To create a new subdesign by using the `group` command, use its arguments and options as follows:

Table 5-8 Using the group Command

To do this	Use this
Specify a list of cells to be grouped into the new subdesign. When the parent design is unique, the list can include cells from a lower level in the hierarchy; however, these cells should be at the same level of hierarchy in relation to one another. To exclude cells from the specified list use the `-except` option.	Provide a list of cells as an argument to the `group` command.
Specify the name of the new subdesign.	`-design_name`
Specify the new instance name (optional). If you do not specify an instance name, Design Compiler creates one for you. The created instance name has the format U*n,* where *n* is an unused cell number (for example, U107).	`-cell_name`

Note:

Grouping cells might not preserve all the attributes and constraints of the original cells.

The following examples illustrate how to use the `group` command.

Example 1

To group two cells into a new design named SAMPLE with an instance name U, enter

```
dc_shell> group {u1 u2} -design_name SAMPLE -cell_name U
```

Example 2

To group all cells that begin with alu into a new design uP with cell name UCELL, enter

```
dc_shell> group "alu*" -design_name uP -cell_name UCELL
```

Example 3

In the following example, three cells—bot1, cell1, and j—are grouped into a new subdesign named SAMPLE, with an instance name U1. The cells are at a lower level in the hierarchy and at the same hierarchical level; the parent design is unique.

```
dc_shell> group {mid1/bot1 mid1/cell1 mid1/j} \
           -cell_name U1 -design_name SAMPLE
```

The preceding command is equivalent to issuing the following two commands:

```
dc_shell> current_design mid
dc_shell> group {bot1 cell1 j} -cell_name U1 -design_name SAMPLE
```

Grouping Related Components Into Subdesigns

You also use the `group` command (but with different options) to group related components into subdesigns. To group related components, use options to the `group` command as follows:

Table 5-9 Using the group Command Options

To do this	Use this
Specify one of the following component types:	
Bused gates	`-hdl_bussed`
Combinational logic	`-logic`
Finite state machines	`-fsm`
HDL blocks	`-hdl_all_blocks` `-hdl_block` *block_name*
PLA specifications	`-pla`
Specify the name of the new subdesign.	`-design_name`
Optionally, specify the new instance name. If you do not specify an instance name, Design Compiler creates one for you. The created instance name has the format U*n,* where *n* is an unused cell number (for example, U107).	`-cell_name`

Note:

You cannot use the `-design_name` and `-cell_name` options with the `hdl_all_blocks` or `hdl_bussed` option.

Example 1

To group all cells in the HDL function bar in the process ftj into design new_block, enter

```
dc_shell> group -hdl_block ftj/bar -design_name new_block
```

Example 2

To group all bused gates beneath process ftj into separate levels of hierarchy, enter

```
dc_shell> group -hdl_block ftj -hdl_bussed
```

Removing Levels of Hierarchy

Removing a level of hierarchy is called ungrouping. Ungrouping merges subdesigns of a given level of the hierarchy into the parent cell or design. Ungrouping can be done before optimization or during optimization (either explicitly or automatically).

The `compile_ultra` command automatically ungroups logical hierarchies. Ungrouping merges subdesigns of a given level of the hierarchy into the parent cell or design. It removes hierarchical boundaries and allows Design Compiler to improve timing by reducing the levels of logic and to improve area by sharing logic. For more information, see the *Design Compiler Optimization Reference Manual*.

You can also manually ungroup hierarchies by using the `ungroup` command or the `set_ungroup` command followed by `compile_ultra`.

Note:

Designs, subdesigns, and cells that have the `dont_touch` attribute cannot be ungrouped (including automatic ungrouping) before or during optimization.

Ungrouping Hierarchies Before Optimization

To ungroup one or more designs before optimization, use the `ungroup` command with the options and arguments in Table 5-10.

Table 5-10 Using the ungroup Command Options

To do this	Use this
Specify a list of cells to be ungrouped. When the parent design is unique, the list can include cells from a lower level in the hierarchy (that is, the `ungroup` command can accept instance objects).	Provide a list of cells as an argument to the `ungroup` command.
Ungroup all cells in the current design or current instance.	`-all`
Ungroup each cell recursively until all levels of hierarchy within the current design (instance) are removed.	`-flatten`
Ungroup cells recursively starting at any hierarchical level. You must specify a number for this option: 1, 2, 3, and so on. A value of 1 indicates that cells from the current design are to be ungrouped. The cells that are at the level specified by the `-start_level` option are included in the ungrouping. Additionally, when you use this option, the current instance cannot be set.	`-start_level` *number*
Specify the prefix to use in naming ungrouped cells. If you do not specify a prefix, Design Compiler uses the prefix *cell_to_be_ungrouped/ old_cell_name {number}*.	`-prefix` *prefix_name*
Ungroup subdesigns with fewer leaf cells than a specified number.	`-small` *number*

Note:

If you ungroup cells and then use the `change_names` command to modify the hierarchy separator (/), you might lose attribute and constraint information.

The following examples illustrate how to use the `ungroup` command.

Example 1

To ungroup a list of cells, enter

```
dc_shell> ungroup {high_decoder_cell low_decoder_cell}
```

Example 2

To ungroup the cell U1 and specify the prefix to use when creating new cells, enter

```
dc_shell> ungroup U1 -prefix "U1:”
```

Example 3

To completely collapse the hierarchy of the current design, enter

```
dc_shell> ungroup -all -flatten
```

Example 4

To recursively ungroup cells belonging to CELL_X, which is three hierarchical levels below the current design, enter

```
dc_shell> ungroup -start_level 3 CELL_X
```

Example 5

To recursively ungroup cells that are three hierarchical levels below the current design and belong to cells U1 and U2 (U1 and U2 are child cells of the current design), enter

```
dc_shell> ungroup -start_level 2 {U1 U2}
```

Example 6

To recursively ungroup all cells that are three hierarchical levels below the current design, enter

```
dc_shell> ungroup -start_level 3 -all
```

Example 7

This example illustrates how the `ungroup` command can accept instance objects (cells at a lower level of hierarchy) when the parent design is unique. In the example, MID1/BOT1 is a unique instantiation of design BOT. The command ungroups the cells MID1/BOT1/CELL1 and MID1/BOT1/CELL2.

```
dc_shell> ungroup {MID1/BOT1/CELL1 MID1/BOT1/CELL2}
```

The preceding command is equivalent to issuing the following two commands:

```
dc_shell> current_instance MID1/BOT1
dc_shell> ungroup {CELL1 CELL2}
```

Ungrouping Hierarchies During Optimization

You can ungroup designs during optimization either explicitly or automatically.

Ungrouping Hierarchies Explicitly During Optimization

You can control which designs are ungrouped during optimization by using the `set_ungroup` command followed by the `compile` command or the `-ungroup_all` compile option.

- Use the `set_ungroup` command when you want to specify the cells or designs to be ungrouped. This command assigns the `ungroup` attribute to the specified cells or referenced designs. If you set the attribute on a design, all cells that reference the design are ungrouped.

  For example, to ungroup cell U1 during optimization, enter the following commands:

  ```
  dc_shell> set_ungroup U1
  dc_shell> compile
  ```

  To see whether an object has the `ungroup` attribute set, use the `get_attribute` command.

  ```
  dc_shell> get_attribute object ungroup
  ```

  To remove an `ungroup` attribute, use the `remove_attribute` command or set the `ungroup` attribute to false.

  ```
  dc_shell> set_ungroup object false
  ```

- Use the `-ungroup_all` compile option to remove all lower levels of the current design hierarchy (including DesignWare parts). For example, enter

  ```
  dc_shell> compile -ungroup_all
  ```

Ungrouping Hierarchies Automatically During Optimization

During optimization, Design Compiler performs the following types of automatic grouping:

- Area-based automatic ungrouping

  Before initial mapping, the `compile_ultra` command performs area-based automatic ungrouping. The tool estimates the area for unmapped hierarchies and removes small subdesigns; the goal is to improve area and timing quality of results. Because the tool performs automatic ungrouping at an early stage, it has a better optimization context. Additionally, datapath extraction is enabled across ungrouped hierarchies. These factors improve the timing and area quality of results.

- Delay-based automatic ungrouping

  During delay optimization, the `compile_ultra` command performs delay-based automatic ungrouping. It ungroups hierarchies along the critical path and is used essentially for timing optimization.

- When you use the `-spg` option with the `compile_ultra` command, Design Compiler Graphical ungroups additional hierarchies to improve QoR.

For more information about automatic ungrouping, see the *Design Compiler Optimization Reference Manual.*

Preserving Hierarchical Pin Timing Constraints During Ungrouping

Hierarchical pins are removed when a cell is ungrouped. Depending on whether you are ungrouping a hierarchy before optimization or after optimization, Design Compiler handles timing constraints placed on hierarchical pins in different ways.

When preserving timing constraints, Design Compiler reassigns the timing constraints to appropriate adjacent, persistent pins (that is, pins on the same net that remain after ungrouping). The constraints are moved forward or backward to other pins on the same net. Note that the constraints can be moved backward only if the pin driving the given hierarchical pin drives no other pin. Otherwise the constraints must be moved forward.

If the constraints are moved to a leaf cell, that cell is assigned a `size_only` attribute to preserve the constraints during a compile. Thus, the number of `size_only` cells can increase, which might limit the scope of the optimization process. To counter this effect, when both the forward and backward directions are possible, Design Compiler chooses the direction that helps limit the number of newly assigned `size_only` attributes to leaf cells.

You can disable this behavior by setting the `auto_ungroup_preserve_constraints` variable to `false`. The default is `true`.

When you apply ungrouping to an unmapped design, the constraints on a hierarchical pin are moved to a leaf cell and the `size_only` attribute is assigned. However, the constraints are preserved through the compile process only if there is a one-to-one match between the unmapped cell and a cell from the target library.

Only the timing constraints set with the following commands are preserved:

- `set_false_path`
- `set_multicycle_path`
- `set_min_delay`
- `set_max_delay`
- `set_input_delay`
- `set_output_delay`
- `set_disable_timing`
- `set_case_analysis`

- `create_clock`
- `create_generated_clock`
- `set_propagated_clock`
- `set_clock_latency`

Note:

The `set_rtl_load` constraint is not preserved. Also, only the timing constraints of the current design are preserved. Timing constraints in other designs might be lost as a result of ungrouping hierarchy in the current design.

Merging Cells From Different Subdesigns

To merge cells from different subdesigns into a new subdesign,

1. Group the cells into a new design.
2. Ungroup the new design.

For example, the following command sequence creates a new alu design that contains the cells that initially were in subdesigns u_add and u_mult.

```
dc_shell> group {u_add u_mult} -design alu
dc_shell> current_design alu
dc_shell> ungroup -all
dc_shell> current_design top_design
```

Editing Designs

Design Compiler provides commands for incrementally editing a design that is in memory. These commands allow you to change the netlist or edit designs by using dc_shell commands instead of an external format.

Table 5-11 Design Editing Tasks and Commands

Object	Task	Command
Cells	Create a cell	`create_cell`
	Delete a cell	`remove_cell`
Nets	Create a net	`create_net`
	Connect a net	`connect_net`
	Disconnect a net	`disconnect_net`
	Delete a net	`remove_net`

Table 5-11 Design Editing Tasks and Commands (Continued)

Object	Task	Command
Ports	Create a port	`create_port`
	Delete a port	`remove_port`
		`remove_unconnected_ports`
Pins	Connect pins	`connect_pin`
Buses	Create a bus	`create_bus`
	Delete a bus	`remove_bus`

For unique designs, these netlist editing commands accept instance objects—that is, cells at a lower level of hierarchy. You can operate on hierarchical designs from any level in the design without using the `current_design` command. For example, you can enter the following command to create a cell called cell1 in the design mid1:

```
dc_shell> create_cell mid1/cell1 my_lib/AND2
```

When connecting or disconnecting nets, use the `all_connected` command to see the objects that are connected to a net, port, or pin. For example, this sequence of commands replaces the reference for cell U8 with a high-power inverter.

```
dc_shell> get_pins U8/*
{"U8/A", "U8/Z"}
dc_shell> all_connected U8/A
{"n66"}
dc_shell> all_connected U8/Z
{"OUTBUS[10]"}
dc_shell> remove_cell U8
Removing cell 'U8' in design 'top'.
1
dc_shell> create_cell U8 IVP
Creating cell 'U8' in design 'top'.
1
dc_shell> connect_net n66 [get_pins U8/A]
Connecting net 'n66' to pin 'U8/A'.
1
dc_shell> connect_net OUTBUS[10] [get_pins U8/Z]
Connecting net 'OUTBUS[10]' to pin 'U8/Z'.
1
```

Note:

You can achieve the same result by using the `change_link` command instead of the series of commands listed previously. For example, the following command replaces the reference for cell U8 with a high-power inverter:

```
dc_shell> change_link U8 IVP
```

The following netlist editing commands are similar to IC Compiler commands:

- Resizing a cell

  To return a collection of equivalent library cells for a specific cell or library cell, use the `get_alternative_lib_cells` command. You can then use the collection to replace or resize the cell. The `size_cell` command allows you to change the drive strength of a leaf cell by linking it to a new library cell that has the required properties.

- Inserting buffers or inverter pairs

  To add a buffer at pins or ports, use the `insert_buffer` command. The `-inverter_pair` option allows you specify that a pair of inverting library cells is to be inserted instead of a single non-inverting library cell. To retrieve a collection of all buffers and inverters from the library, you can use the `get_buffers` command.

- Inserting repeaters

  To select a driver of a two-pin net and insert a chain of single-fanout buffers in the net driven by this driver, use the `insert_buffer` command with the `-no_of_cells` option.

- Removing buffers

  To remove buffers, use the `remove_buffer` command.

Translating Designs From One Technology to Another

To translate a design from one technology to another, use the `translate` command. If you are using Design Compiler in topographical mode, use the `compile_ultra -incremental` command. Designs are translated cell by cell from the original logic library to a new logic library, preserving the gate structure of the original design. The translator uses the functional description of each existing cell (component) to determine the matching component in the new logic library (target library). If no exact replacement exists for a component, it is remapped with components from the target library.

You can influence the replacement-cell selection by preferring or disabling specific library cells (using the `set_prefer` and `set_dont_use` commands) and by specifying the types of registers (using the `set_register_type` command). The target libraries are specified in the `target_library` variable. The `local_link_library` attribute of the top-level design is set to the `target_library` value after the design is linked.

The `translate` command does not operate on cells or designs having the `dont_touch` attribute. After the translation process, Design Compiler reports cells that are not successfully translated.

Translating Designs in Design Compiler

The following procedure works for most designs, but manual intervention might be necessary for some complex designs.

To translate a design,

1. Read in your mapped design.

   ```
   dc_shell> read_file design.ddc
   ```

2. Set the target library to the new logic library.

   ```
   dc_shell> set target_library target_lib.db
   ```

3. Invoke the `translate` command.

   ```
   dc_shell> translate
   ```

After a design is translated, you can compile it to improve the implementation in the new logic library.

Translating Designs in Design Compiler Topographical Mode

To map a design to a new technology in Design Compiler topographical mode,

1. Add the original logic library to the link library:

   ```
   dc_shell-topo> lappend link_library tech_orig.db
   ```

2. Set up your Design Compiler environment for the new target technology.

   a. Specify the logic libraries.

   b. Specify the physical libraries.

3. Read in your mapped design:

   ```
   dc_shell-topo> read_verilog design.v
   ```

4. Run the `compile_ultra -incremental` command to translate the design to the new technology.

Restrictions on Translating Between Technologies

Keep the following restrictions in mind when you translate a design from one technology to another:

- The `translate` command translates functionality logically but does not preserve drive strength during translation. It always uses the lowest drive strength version of a cell, which might produce a netlist with violations.
- Buses driven by CMOS three-state components must be fully decoded (Design Compiler can assume that only one bus driver is ever active). If this is the case, bus drivers are translated into control logic. To enable this feature, set the `compile_assume_fully_decoded_three_state_buses` variable to `true` before translating.
- If a three-state bus within a design is connected to one or more output ports, translating the bus to a multiplexed signal changes the port functionality. Because `translate` does not change port functionality, this case is reported as a translation error.

Removing Designs From Memory

The `remove_design` command removes designs from dc_shell memory. For example, after completing a compilation session and saving the optimized design, you can use `remove_design` to delete the design from memory before reading in another design.

By default, the `remove_design` command removes only the specified design. To remove its subdesigns, specify the `-hierarchy` option. To remove all designs (and libraries) from memory, specify the `-all` option.

If you defined variables that reference design objects, Design Compiler removes these references when you remove the design from memory. This prevents future commands from attempting to operate on nonexistent design objects. For example,

```
dc_shell> set PORTS [all_inputs]
{"A0", "A1", "A2", "A3"}
dc_shell> query_objects $PORTS
PORTS = {"A0", "A1", "A2", "A3"}
dc_shell> remove_design
Removing design 'top'
1
dc_shell> query_objects $PORTS
Error: No such collection '_sel2' (SEL-001)
```

Saving Designs

You can save (write to disk) the designs and subdesigns of the design hierarchy at any time, using different names or formats. After a design is modified, you should manually save it. Design Compiler does not automatically save designs before it exits.

To write a design netlist or schematic to a file that you specify, use the `write_file` command. Design Compiler supports the design file formats listed in Table 5-12.

Table 5-12 Supported Output Formats

Format	Description
.ddc	Synopsys internal database format
Milkyway	Format for writing a Milkyway database within Design Compiler
Verilog	IEEE Standard Verilog (see the HDL Compiler documentation)
svsim	SystemVerilog netlist wrapper. Note: The `write_file -format svsim` command writes out only the netlist wrapper, not the gate-level DUT itself. To write out the gate-level DUT, you must use the existing `write_file -format verilog` command. For details, see the *HDL Compiler for SystemVerilog User Guide*.
VHDL	IEEE Standard VHDL (see the HDL Compiler documentation)

Saving Designs in .ddc Format

To save the design data in a .ddc file, use the `write_file -format ddc` command.

By default, the `write_file` command saves just the top-level design. To save the entire design, specify the `-hierarchy` option. If you do not use the `-output` option to specify the output file name, the `write_file -format ddc` command creates a file called *top_design*.ddc, where *top_design* is the name of the current design.

Example 1

The following command writes out all designs in the hierarchy of the specified design:

```
dc_shell> write_file -hierarchy -format ddc top
Writing ddc file 'top.ddc'
Writing ddc file 'A.ddc'
Writing ddc file 'B.ddc'
```

Example 2

The following command writes out multiple designs to a single file:

```
dc_shell> write_file -format ddc -output test.ddc {ADDER MULT16}

Writing ddc file 'test.ddc'
```

Writing a Milkyway Database

To write to a Milkyway database, use the `write_milkyway` command within dc_shell. The `write_milkyway` command creates a design file based on the netlist in memory and saves the design data for the current design in that file.

For more information, see Using a Milkyway Database.

Saving Designs Using GUI Commands

You can also use the GUI to save the current design and each of its subdesigns.

To save the current design and each of its subdesigns in separate .ddc format files named *design_name*.ddc, where *design_name* is the name of the design, choose File > Save.

To save the current design and all of its subdesigns in a single file with a different file name or file format, choose File > Save As.

You can also save window images in the GUI by using the `gui_write_window_image` command. The command saves an image of a view window or a top-level window in the specified image file. You can specify the format of the image as `bmp`, `jpg`, `xpm`, or `png`. The default format is `png`. You can also save window images in batch mode from a script.

For more information about using the GUI, see the *Design Vision User Guide* and the Design Vision Help.

Ensuring Name Consistency Between the Design Database and the Netlist

Before writing a netlist from within dc_shell, make sure that all net and port names conform to the naming conventions for your layout tool. Also ensure that you are using a consistent bus naming style.

Some ASIC and EDA vendors have a program that creates a .synopsys_dc.setup file that includes the appropriate commands to convert names to their conventions. If you need to change any net or port names, use the `define_name_rules` and `change_names` commands.

Naming Rules Section of the .synopsys_dc.setup File

Example 5-1 shows sample naming rules as created by a specific layout tool vendor. These naming rules do the following:

- Limit object names to alphanumeric characters
- Change DesignWare cell names to valid names (changes “*cell*” to “U” and “*-return” to “RET”)

Your vendor might use different naming conventions. Check with your vendor to determine the naming conventions you need to follow.

Example 5-1 Naming Rules Section of .synopsys_dc.setup File

```
define_name_rules simple_names -allowed "A-Za-z0-9_" \
-last_restricted "_" \
-first_restricted "_" \
-map { {{"\*cell\*","U"}, {"*-return","RET"}} }
```

Specifying the Name Mapping and Replacement Rules

Example 5-2 shows how to use the `-map` option with the `define_name_rules` command to define the name mapping and replacement rules to avoid an error in the format of the string. If you do not follow this convention, an error appears.

Example 5-2 Using define_name_rules -map

```
define_name_rules naming_convention
-map { {{string1, string2}} } -type cell
```

For example, to remove trailing underscores from cell names, enter

```
dc_shell> define_name_rules naming_convention \
                        -map {{{_$, ""}} } -type cell
```

Resolving Naming Problems in the Flow

You might encounter conflicts in naming conventions in design objects, input and output files, and tool sets. In the design database file, you can have many design objects (such as ports, nets, cells, logic modules, and logic module pins), all with their own naming conventions. Furthermore, you might be using several input and output file formats in your flow. Each file format is different and has its own syntax definitions. Using tool sets from several vendors can introduce additional naming problems.

To resolve naming issues, use the `change_names` command to ensure that all the file names match. Correct naming eliminates name escaping or mismatch errors in your design.

Methodology for Resolving Naming Issues

To resolve naming issues, make the name changes in the design database file before you write any files:

1. Read in your design RTL and apply constraints.

   No changes to your method need to be made here.

2. Compile the design to produce a gate-level description.

   Compile or reoptimize your design as you normally would, using your standard set of scripts.

3. Apply name changes and resolve naming issues. Use the `change_names` command and its Verilog or VHDL option before you write the design.

   Important:
   Always use the `change_names -rules [verilog|vhdl] -hierarchy` command whenever you want to write out a Verilog or VHDL design because naming in the design database file is not Verilog or VHDL compliant. For example, enter

   ```
   dc_shell> change_names -rules verilog -hierarchy
   ```

4. Write the files to disk. Use the `write -format verilog` command.

   Look for reported name changes, which indicate you need to repeat step 3 and refine your name rules.

5. If all the appropriate name changes have been made, your output file should match the design database file. Enter the following commands and compare the output.

   ```
   dc_shell> write -format verilog -hierarchy -output "consistent.v"
   dc_shell> write -format ddc -hierarchy -output "consistent.ddc"
   ```

6. Write the files for third-party tools.

If you need more specific naming control, use the `define_name_rules` command. See Specifying the Name Mapping and Replacement Rules.

Summary of Commands for Changing Names

Table 5-13 summarizes commands for changing names.

Table 5-13 Summary of Commands for Changing Names

To do this	Use this
Change the names of ports, cells, and nets in a design to be Verilog or VHDL compliant.	`change_names`
Show effects of `change_names` without making the changes.	`report_names`
Define a set of rules for naming design objects. Name rules are used by `change_names` and `report_names`.	`define_name_rules`
List available name rules.	`report_name_rules`

Working With Attributes

Attributes describe logical, electrical, physical, and other properties of objects in the design database. An attribute is attached to a design object and is saved with the design database.

Design Compiler uses attributes on the following types of objects:

- Entire designs
- Design objects, such as clocks, nets, pins, and ports
- Design references and cell instances within a design
- Technology libraries, library cells, and cell pins

An attribute has a name, a type, and a value. Attributes can have the following types: string, numeric, or logical (Boolean).

Some attributes are predefined and are recognized by Design Compiler; other attributes are user-defined. Appendix C lists the predefined attributes.

Some attributes are read-only. Design Compiler sets these attribute values and you cannot change them. Other attributes are read/write. You can change these attribute values at any time.

Most attributes apply to one object type; for example, the `rise_drive` attribute applies only to input and inout ports. Some attributes apply to several object types; for example, the `dont_touch` attribute can apply to a net, cell, port, reference, or design. You can get detailed

information about the predefined attributes that apply to each object type by using the commands listed in Table 5-14.

Table 5-14 Commands to Get Attribute Descriptions

Object type	Command
All	`man attributes`
Designs	`man design_attributes`
Cells	`man cell_attributes`
Clocks	`man clock_attributes`
Nets	`man net_attributes`
Pins	`man pin_attributes`
Ports	`man port_attributes`
Libraries	`man library_attributes`
Library cells	`man library_cell_attributes`
References	`man reference_attributes`

Setting Attribute Values

To set the value of an attribute, use one of the following:

- An attribute-specific command
- The `set_attribute` command

Using an Attribute-Specific Command

Use an attribute-specific command to set the value of the command's associated attribute.

For example,

```
dc_shell> set_dont_touch U1
```

Using the set_attribute Command

Use this command to set the value of any attribute or to define a new attribute and set its value.

For example, to set the dont_touch attribute on the lsi_10k/FJK3 library cell, enter

```
dc_shell> set_attribute lsi_10K/FJK3 dont_touch true
```

The set_attribute command enforces the predefined attribute type and generates an error if you try to set an attribute with a value of an incorrect type.

To determine the predefined type for an attribute, use the list_attributes -application command. This command generates a list of all application attributes and their types. To generate a smaller report, you can use the -class attribute to limit the list to attributes that apply to one of the following classes: design, port, cell, clock, pin, net, lib, or reference.

For example, the max_fanout attribute has a predefined type of float. Suppose you enter the following command, Design Compiler displays an error message:

```
set_attribute lib/lcell/lpin max_fanout 1 -type integer
```

If an attribute applies to more than one object type, Design Compiler searches the database for the named object. For information about the search order, see The Object Search Order.

When you set an attribute on a reference (subdesign or library cell), the attribute applies to all cells in the design with that reference. When you set an attribute on an instance (cell, net, or pin), the attribute overrides any attribute inherited from the instance's reference.

Viewing Attribute Values

To see all attributes on an object, use the report_attribute command.

```
dc_shell> report_attribute object_list
```

To see the value of a specific attribute on an object, use the get_attribute command.

For example, to get the value of the maximum fanout on port OUT7, enter

```
dc_shell> get_attribute OUT7 max_fanout
Performing get_attribute on port 'OUT7'.
{3.000000}
```

If an attribute applies to more than one object type, Design Compiler searches the database for the named object. For information about the search order, see The Object Search Order.

You can use the Properties dialog box in the GUI to view attributes and other object properties for selected designs, design objects, or timing paths. You can also set, change, or remove the attribute values for certain properties.

For more information about using the GUI, see the *Design Vision User Guide* and the Design Vision Help.

Saving Attribute Values

Design Compiler does not automatically save attribute values when you exit dc_shell. Use the `write_script` command to generate a dc_shell script that re-creates the attribute values.

By default, the `write_script` command writes the dc_shell commands to standard output. Use the redirection operator (>) to redirect the output to a file.

```
dc_shell> write_script > attr.scr
```

The `write_script` command does not support user-defined attributes.

Defining Attributes

The `set_attribute` command enables you to create new attributes. Use the `set_attribute` command described in Using the set_attribute Command.

If you want to change the value of an attribute, remove the attribute and then re-create it to store the desired type.

Removing Attributes

To remove a specific attribute from an object, use the `remove_attribute` command.

You cannot use the `remove_attribute` command to remove inherited attributes. For example, if a `dont_touch` attribute is assigned to a reference, remove the attribute from the reference, not from the cells that inherited the attribute.

For example, to remove the `max_fanout` attribute from port OUT7, enter

```
dc_shell> remove_attribute OUT7 max_fanout
```

You can remove selected attributes by using the `remove_*` commands. Note that some attributes still require the `set_*` command with a `-default` option specified to remove the attribute previously set by the command. See the man page for a specific command to determine whether it has the `-default` option or uses a corresponding `remove` command.

To remove all attributes from the current design, use the `reset_design` command.

```
dc_shell> reset_design
Resetting current design 'EXAMPLE'.
1
```

The `reset_design` command removes all design information, including clocks, input and output delays, path groups, operating conditions, timing ranges, and wire load models. The

result of using `reset_design` is often equivalent to starting the design process from the beginning.

The Object Search Order

When Design Compiler searches for an object, the search order is command dependent. Objects include designs, cells, nets, references, and library cells.

If you do not use a `get` command, Design Compiler uses an implicit find to locate the object. Commands that can set an attribute on more than one type of object use this search order to determine the object to which the attribute applies.

For example, the `set_dont_touch` command operates on cells, nets, references, and library cells. If you define an object, *X,* with the `set_dont_touch` command and two objects (such as the design and a cell) are named *X,* Design Compiler applies the attribute to the first object type found. In this case, the attribute is set on the design, not on the cell.

Design Compiler searches until it finds a matching object, or it displays an error message if it does not find a matching object.

You can override the default search order by using the `get_*` command to specify the object.

For example, assume that the current design contains both a cell and a net named critical. The following command sets the `dont_touch` attribute on the cell because of the default search order:

```
dc_shell> set_dont_touch critical
1
```

To place the `dont_touch` attribute on the net instead, use the following command:

```
dc_shell> set_dont_touch [get_nets critical]
1
```

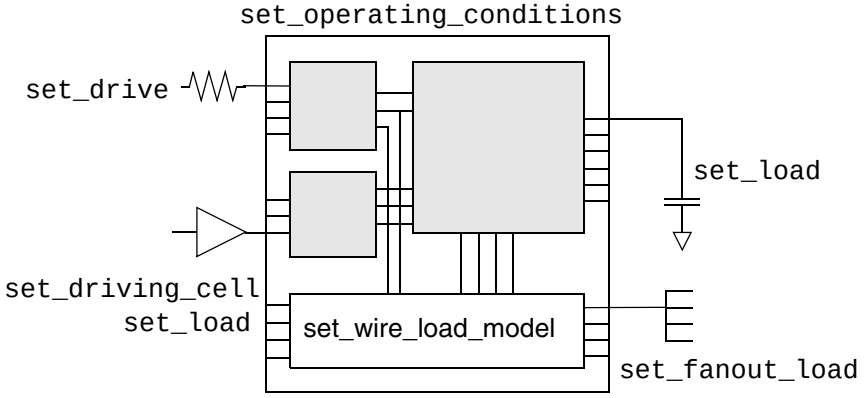
6

Defining the Design Environment

Before a design can be optimized, you must define the environment in which the design is expected to operate. You define the environment by specifying operating conditions, wire load models (used only when Design Compiler is not operating in topographical mode), and system interface characteristics. Operating conditions include temperature, voltage, and process variations. Wire load models estimate the effect of wire length on design performance. System interface characteristics include input drives, input and output loads, and fanout loads. The environment model directly affects design synthesis results.

In Design Compiler, the model is defined by a set of attributes and constraints that you assign to the design, using specific dc_shell commands. Figure 6-1 illustrates the commands used to define the design environment.

Figure 6-1 Commands Used to Define the Design Environment



To learn how to define the design environment, see

- Defining the Operating Conditions
- Defining Wire Load Models
- Modeling the System Interface
- Setting Logic Constraints on Ports
- Specifying Low Power Intent
- Support for Multicorner-Multimode Designs

Defining the Operating Conditions

In most technologies, variations in operating temperature, supply voltage, and manufacturing process can strongly affect circuit performance (speed). These factors, called operating conditions, have the following general characteristics:

- Operating temperature variation

  Temperature variation is unavoidable in the everyday operation of a design. Effects on performance caused by temperature fluctuations are most often handled as linear scaling effects, but some submicron silicon processes require nonlinear calculations.

- Supply voltage variation

  The design's supply voltage can vary from the established ideal value during day-to-day operation. Often a complex calculation (using a shift in threshold voltages) is employed, but a simple linear scaling factor is also used for logic-level performance calculations.

- Process variation

  This variation accounts for deviations in the semiconductor fabrication process. Usually process variation is treated as a percentage variation in the performance calculation.

When performing timing analysis, Design Compiler must consider the worst-case and best-case scenarios for the expected variations in the process, temperature, and voltage factors.

Determining Available Operating Condition Options

Most technology libraries have predefined sets of operating conditions. Use the `report_lib` command to list the operating conditions defined in a logic library. The library must be loaded in memory before you can run the `report_lib` command. To see the list of libraries loaded in memory, use the `list_libraries` or the `list_libs` command.

For example, to generate a report for the library my_lib, which is stored in my_lib.db, enter the following commands:

```
dc_shell> read_file my_lib.db
dc_shell> report_lib my_lib
```

Example 6-1 shows the resulting operating conditions report.

Example 6-1 Operating Conditions Report

```
****************************************
Report : library
Library: my_lib
...
****************************************
...
```

```
Operating Conditions:

    Name       Library       Process     Temp    Volt    Interconnect Model
    -------------------------------------------------------------------------
    WCCOM      my_lib            1.50    70.00    4.75    worst_case_tree
    WCIND      my_lib            1.50    85.00    4.75    worst_case_tree
    WCMIL      my_lib            1.50   125.00    4.50    worst_case_tree
...
```

Specifying Operating Conditions

If the logic library contains operating condition specifications, you can let Design Compiler use them as default conditions. Alternatively, you can use the `set_operating_conditions` command to specify explicit operating conditions, which supersede the default library conditions.

For example, to set the operating conditions for the current design to worst-case commercial, enter

```
dc_shell> set_operating_conditions WCCOM -library my_lib
```

Use the `report_design` command to see the operating conditions defined for the current design.

Defining Wire Load Models

Wire load models are used only when Design Compiler is not operating in topographical mode. For topographical mode details, see Using Design Compiler Topographical Technology. Wire load modeling allows you to estimate the effect of wire length and fanout on the resistance, capacitance, and area of nets. Design Compiler uses these physical values to calculate wire delays and circuit speeds.

Semiconductor vendors develop wire load models, based on statistical information specific to the vendors' process. The models include coefficients for area, capacitance, and resistance per unit length, and a fanout-to-length table for estimating net lengths (the number of fanouts determines a nominal length).

Note:

You can also develop custom wire load models. For more information about developing wire load models, see the Library Compiler documentation.

In the absence of back-annotated wire delays, Design Compiler uses the wire load models to estimate net wire lengths and delays. Design Compiler determines which wire load model to use for a design, based on the following factors, listed in order of precedence:

1. Explicit user specification
2. Automatic selection based on design area
3. Default specification in the logic library

If none of this information exists, Design Compiler does not use a wire load model. Without a wire load model, Design Compiler does not have complete information about the behavior of your target technology and cannot compute loading or propagation times for your nets; therefore, your timing information will be optimistic.

In hierarchical designs, Design Compiler must also determine which wire load model to use for nets that cross hierarchical boundaries. The tool determines the wire load model for cross-hierarchy nets based on one of the following factors, listed in order of precedence:

1. Explicit user specification
2. Default specification in the logic library
3. Default mode in Design Compiler

The following sections discuss the selection of wire load models for nets and designs.

Hierarchical Wire Load Models

Design Compiler supports three modes for determining which wire load model to use for nets that cross hierarchical boundaries:

- Top

  Design Compiler models nets as if the design has no hierarchy and uses the wire load model specified for the top level of the design hierarchy for all nets in a design and its subdesigns. The tool ignores any wire load models set on subdesigns with the `set_wire_load_model` command.

  Use top mode if you plan to flatten the design at a higher level of hierarchy before layout.

- Enclosed

  Design Compiler uses the wire load model of the smallest design that fully encloses the net. If the design enclosing the net has no wire load model, the tool traverses the design hierarchy upward until it finds a wire load model. Enclosed mode is more accurate than top mode when cells in the same design are placed in a contiguous region during layout.

  Use enclosed mode if the design has similar logical and physical hierarchies.

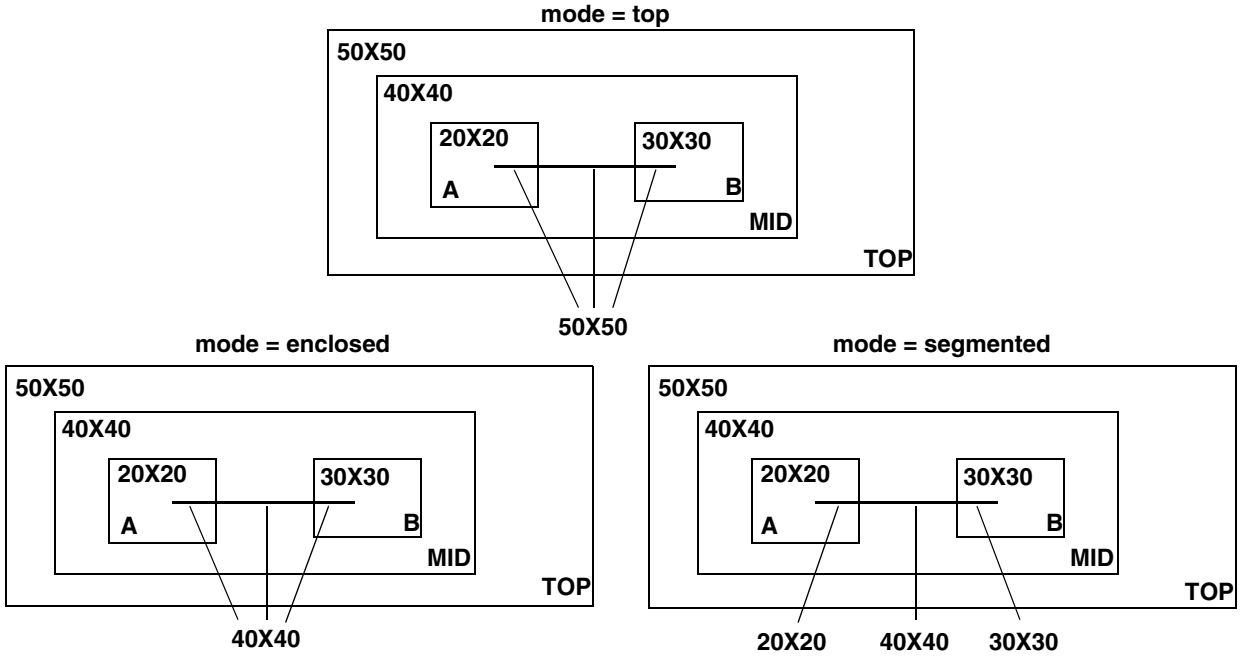
- Segmented

  Design Compiler determines the wire load model of each segment of a net by the design encompassing the segment. Nets crossing hierarchical boundaries are divided into segments. For each net segment, Design Compiler uses the wire load model of the design containing the segment. If the design contains a segment that has no wire load model, the tool traverses the design hierarchy upward until it finds a wire load model.

  Use segmented mode if the wire load models in your technology have been characterized with net segments.

Figure 6-2 shows a sample design with a cross-hierarchy net, cross_net. The top level of the hierarchy (design TOP) has a wire load model of 50x50. The next level of hierarchy (design MID) has a wire load model of 40x40. The leaf-level designs, A and B, have wire load models of 20x20 and 30x30, respectively.

Figure 6-2 Comparison of Wire Load Mode



In top mode, Design Compiler estimates the wire length of net cross_net, using the 50x50 wire load model. Design Compiler ignores the wire load models on designs MID, A, and B.

In enclosed mode, Design Compiler estimates the wire length of net cross_net, using the 40x40 wire load model (the net cross_net is completely enclosed by design MID).

In segmented mode, Design Compiler uses the 20x20 wire load model for the net segment enclosed in design A, the 30x30 wire load model for the net segment enclosed in design B, and the 40x40 wire load model for the segment enclosed in design MID.

Determining Available Wire Load Models

Most technology libraries have predefined wire load models. Use the `report_lib` command to list the wire load models defined in a logic library. The library must be loaded in memory before you run the `report_lib` command. To see a list of libraries loaded in memory, use the `list_libs` command.

The wire load report contains the following sections:

- Wire Loading Model section

  This section lists the available wire load models.

- Wire Loading Model Mode section

  This section identifies the default wire load mode. If a library default does not exist, Design Compiler uses top mode.

- Wire Loading Model Selection Group section

  The presence of this section indicates that the library supports automatic area-based wire load model selection.

To generate a wire load report for the my_lib library, enter

```
dc_shell> read_file my_lib.db
dc_shell> report_lib my_lib
```

Example 6-2 shows the resulting wire load models report. The library my_lib contains three wire load models: 05x05, 10x10, and 20x20. The library does not specify a default wire load mode (so Design Compiler uses top as the default wire load mode), and it supports automatic area-based wire load model selection.

Example 6-2 Wire Load Models Report

```
****************************************
Report : library
Library: my_lib
...
****************************************
...
Wire Loading Model:

Name           :   05x05
Location       :   my_lib
Resistance     :   0
Capacitance    :   1
Area           :   0
Slope          :   0.186
Fanout   Length   Points Average Cap Std Deviation
------------------------------------------------------------------------
     1     0.39
```

```
Name           :   10x10
Location       :   my_lib
Resistance     :   0
Capacitance    :   1
Area           :   0
Slope          :   0.311
Fanout   Length   Points Average Cap Std Deviation
------------------------------------------------------------------------
     1     0.53

Name           :   20x20
Location       :   my_lib
Resistance     :   0
Capacitance    :   1
Area           :   0
Slope          :   0.547
Fanout   Length   Points Average Cap Std Deviation
------------------------------------------------------------------------
     1     0.86
Wire Loading Model Selection Group:

   Name           : my_lib

        Selection             Wire load name
    min area   max area
   ----------------------------------------
        0.00    1000.00          05x05
     1000.00    2000.00          10x10
     2000.00    3000.00          20x20
...
```

Specifying Wire Load Models and Modes

The logic library can define a default wire load model that is used for all designs implemented in that technology. The `default_wire_load` library attribute identifies the default wire load model for a logic library.

Some libraries support automatic area-based wire load selection. Design Compiler uses the library function `wire_load_selection` to choose a wire load model based on the total cell area. The wire load model selected the first time you compile is used in subsequent compiles.

For large designs with many levels of hierarchy, automatic wire load selection can increase runtime. To manage runtime, set the wire load manually.

You can turn off automatic selection of the wire load model by setting the `auto_wire_load_selection` variable to `false`. For example, enter the following command:

```
dc_shell> set auto_wire_load_selection false
```

The logic library can also define a default wire load mode. The `default_wire_load_mode` library attribute identifies the default mode. If the current library does not define a default

mode, Design Compiler looks for the attribute in the libraries specified in the `link_library` variable. (To see the link library, use the `list` command.) In the absence of a library default (and an explicit specification), Design Compiler uses that top mode.

To change the wire load model or mode specified in a logic library, use the `set_wire_load_model` and `set_wire_load_mode` commands. The wire load model and mode you define override all defaults. Explicitly selecting a wire load model also disables area-based wire load model selection for that design.

For example, to select the 10x10 wire load model, enter

```
dc_shell> set_wire_load_model "10x10"
```

To select the 10x10 wire load model and specify enclosed mode, enter

```
dc_shell> set_wire_load_mode enclosed
```

The wire load model you choose for a design depends on how that design is implemented in the chip. Consult your semiconductor vendor to determine the best wire load model for your design.

Use the `report_design` command or the `report_timing` command to see the wire load model and mode defined for the current design.

To remove the wire load model, use the `remove_wire_load_model` command with no model name.

Modeling the System Interface

Design Compiler supports the following ways to model the design's interaction with the external system:

- Defining Drive Characteristics for Input Ports
- Defining Loads on Input and Output Ports
- Defining Fanout Loads on Output Ports

Defining Drive Characteristics for Input Ports

Design Compiler uses drive strength information to buffer nets appropriately in the case of a weak driver.

Note:

Drive strength is the reciprocal of the output driver resistance, and the transition time delay at an input port is the product of the drive resistance and the capacitance load of the input port.

By default, Design Compiler assumes zero drive resistance on input ports, meaning infinite drive strength. There are three commands for overriding this unrealistic assumption:

- `set_driving_cell`
- `set_drive`
- `set_input_transition`

Both the `set_driving_cell` and `set_input_transition` commands affect the port transition delay, but they do not place design rule requirements, such as `max_fanout` and `max_transition`, on input ports. However, the `set_driving_cell` command does place design rules on input ports if the driving cell has design rule constraints.

Note:

For heavily loaded driving ports, such as clock lines, keep the drive strength setting at 0 so that Design Compiler does not buffer the net. Each semiconductor vendor has a different way of distributing these signals within the silicon.

Both the `set_drive` and the `set_driving_cell` commands affect the port transition delay. The `set_driving_cell` command can place design rule requirements, such as the `max_fanout` or `max_transition` attributes, on input ports if the specified cell has input ports.

The most recently used command takes precedence. For example, setting a drive resistance on a port with the `set_drive` command overrides previously run `set_driving_cell` commands.

Specifying Drive Characteristics on Ports Driven by Logic Library Cells

Use the `set_driving_cell` command to specify drive characteristics on ports that are driven by cells in the logic library. This command is compatible with all the delay models, including the nonlinear delay model and piecewise linear delay model. The `set_driving_cell` command associates a library pin with an input port so that delay calculators can accurately model the drive capability of an external driver.

Use the `remove_driving_cell` command or `reset_design` command to remove driving cell attributes on ports.

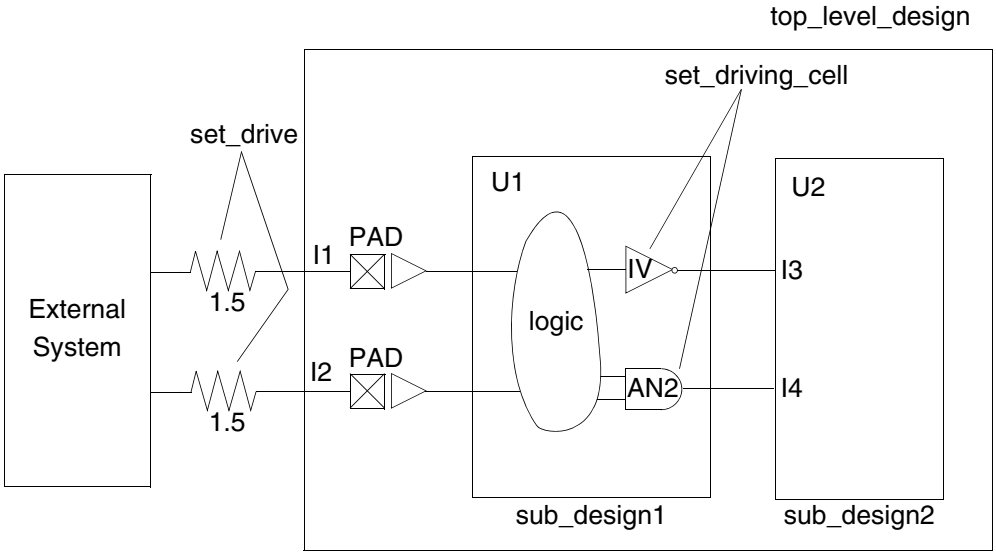
Setting the Drive Resistance on Top-Level Ports

Use the `set_drive` or `set_input_transition` command to set the drive resistance on the top-level ports of the design when the input port drive capability cannot be characterized with a cell in the logic library.

You can use `set_drive` and the `drive_of` commands together to represent the drive resistance of a cell. However, these commands are not as accurate for nonlinear delay models as the `set_driving_cell` command is.

Figure 6-3 shows a hierarchical design. The top-level design has two subdesigns, U1 and U2. Ports I1 and I2 of the top-level design are driven by the external system and have a drive resistance of 1.5.

Figure 6-3 Drive Characteristics



To set the drive characteristics for this example, follow these steps:

1. Because ports I1 and I2 are not driven by library cells, use the `set_drive` command to define the drive resistance. Enter

```
dc_shell> current_design top_level_design
dc_shell> set_drive 1.5 {I1 I2}
```

2. To describe the drive capability for the ports on design sub_design2, change the current design to sub_design2. Enter

```
dc_shell> current_design sub_design2
```

3. An IV cell drives port I3. Use the `set_driving_cell` command to define the drive resistance. Because IV has only one output and one input, define the drive capability as follows. Enter

   ```
   dc_shell> set_driving_cell -lib_cell IV {I3}
   ```

4. An AN2 cell drives port I4. Because the different arcs of this cell have different transition times, select the worst-case arc to define the drive. For checking setup violations, the worst-case arc is the slowest arc. For checking hold violations, the worst-case arc is the fastest arc.

   For this example, assume that you want to check for setup violations. The slowest arc on the AN2 cell is the B-to-Z arc, so define the drive as follows. Enter

   ```
   dc_shell> set_driving_cell -lib_cell AN2 -pin Z -from_pin B {I4}
   ```

Defining Loads on Input and Output Ports

By default, Design Compiler assumes zero capacitive load on input and output ports. Use the `set_load` command to set a capacitive load value on input and output ports of the design. This information helps Design Compiler select the appropriate cell drive strength of an output pad and helps model the transition delay on input pads.

For example, to set a load of 30 on output pin out1, enter

```
dc_shell> set_load 30 {out1}
```

Make the units for the load value consistent with the target logic library. For example, if the library represents the load value in picofarads, the value you set with the `set_load` command must be in picofarads. Use the `report_lib` command to list the library units.

Example 6-3 shows the library units for the library my_lib.

Example 6-3 Library Units Report

```
****************************************
Report : library
Library: my_lib
...
****************************************
Library Type            : Technology
Tool Created            : 1999.05
Date Created            : February 7, 1992
Library Version         : 1.800000
Time Unit               : 1ns
Capacitive Load Unit    : 0.100000ff
Pulling Resistance Unit : 1kilo-ohm
Voltage Unit            : 1V
Current Unit            : 1uA
...
```

Defining Fanout Loads on Output Ports

You can model the external fanout effects by specifying the expected fanout load values on output ports with the `set_fanout_load` command.

For example, enter

```
dc_shell> set_fanout_load 4 {out1}
```

Design Compiler tries to ensure that the sum of the fanout load on the output port plus the fanout load of cells connected to the output port driver is less than the maximum fanout limit of the library, library cell, and design. For more information about maximum fanout limits, see Design Rule Constraints.

Fanout load is not the same as load. Fanout load is a unitless value that represents a numerical contribution to the total fanout. Load is a capacitance value. Design Compiler uses fanout load primarily to measure the fanout presented by each input pin. An input pin normally has a fanout load of 1, but it can have a higher value.

Setting Logic Constraints on Ports

Design Compiler provides commands for setting ports to improve optimization quality. Table 6-1 lists the commands that eliminate redundant ports or inverters.

Table 6-1 Commands Eliminating Redundant Ports or Inverters

Command	Description
`set_equal`	Defines ports as logically equivalent
`set_opposite`	Defines ports as logically opposite
`set_logic_dc`	Specifies one or more ports driven by don't care
`set_logic_one`	Specifies one or more ports tied to logic 1
`set_logic_zero`	Specifies one or more ports tied to logic 0
`set_unconnected`	Lists output ports to be unconnected

To learn about eliminating redundant ports or inverters, see

- Defining Ports as Logically Equivalent
- Defining Logically Opposite Input Ports

Defining Ports as Logically Equivalent

Some input ports are driven by logically related signals. For example, the signals driving a pair of input ports might always be the same (logically equal) or might always be different (logically opposite).

The `set_equal` command specifies that two input ports are logically equivalent.

To specify that ports IN_X and IN_Y are equal, enter

```
dc_shell> set_equal IN_X IN_Y
```

To remove this attribute, use the `reset_design` command. The `reset_design` command removes all user-specified objects and attributes from the current design, except those defined with the `set_attribute` command.

Defining Logically Opposite Input Ports

The `set_opposite` command specifies that two input ports in the current designs are logically opposite. Use the `set_opposite` command to eliminate redundant inverters and to improve the quality of optimization.

To remove this attribute, use the `reset_design` command, which removes all user-specified objects and attributes from the current design, except those defined with the `set_attribute` command. To remove those defined with `set_attribute`, use the `remove_attribute` command.

Allowing Assignment of Any Signal to an Input

The `set_logic_dc` command specifies an input port driven by a dont_care. This information is used to create smaller designs during compile. After optimization, a port connected to a dont_care usually does not drive anything inside the optimized design.

Use `set_logic_dc` to allow assignment of any signal to that input, including but not limited to 0 and 1 during compilation. The outputs of the design are significant only when the inputs that are not dont_care completely determine all the outputs, independent of the dont_care inputs.

Use the set_logic_dc command on input ports. To specify output ports as unused, use the set_unconnected command.

Example

```
dc_shell> set_logic_dc { A B }
```

For a 2:1 multiplexer design with inputs S, A, B, and output Z, the function is computed as

```
Z = S*A + S'*B
```

The command set_logic_dc B implies that the value of Z is significant when S = 1 and is a don't care when S = 0. The resulting simplification, done during compilation, gives the reduced logic as a wire and the function is

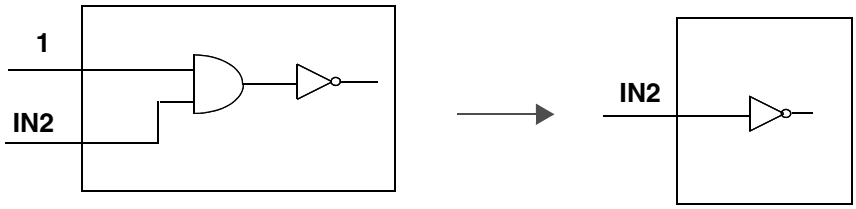
```
Z = A
```

To undo this command, use the remove_attribute command.

Specifying Input Ports Always One or Zero

If an input port is always logic-high or -low, Design Compiler might be able to simplify the surrounding logic function during optimization and create a smaller design. Figure 6-4 shows simplified input port logic.

Figure 6-4 Simplified Input Port Logic



You can specify that input ports are connected to logic 1 or logic 0.

Tying Input Ports to Logic 1

The set_logic_one command lists the input ports tied to logic 1. After optimization, a port connected to logic 1 usually does not drive anything inside the optimized design.

Use the set_logic_one command on input ports. To specify output ports as unused, use the set_unconnected command.

To undo this command, use the remove_attribute command.

Tying Input Ports to Logic 0

The `set_logic_zero` command lists input ports tied to logic 0. After optimization, a port connected to logic 0 usually does not drive anything inside the optimized design.

Use the `set_logic_zero` command on input ports. To specify output ports as unused, use the `set_unconnected` command.

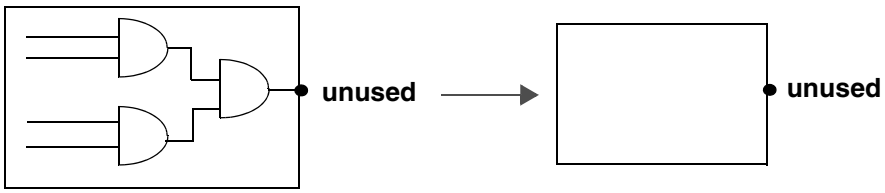
Example

```
dc_shell> set_logic_one IN
```

Specifying Unconnected Output Ports

If an output port is not used, that is, if it is unconnected, the logic driving the port can be minimized or eliminated during optimization. Figure 6-5 shows an example.

Figure 6-5 Minimizing Logic Driving an Unconnected Output Port



The `set_unconnected` command specifies output ports to be unconnected to outside logic.

Example

```
dc_shell> set_unconnected OUT
```

To undo this command, use the `remove_attribute` command.

Specifying Low Power Intent

The IEEE 1801, also known as Unified Power Format (UPF), standard establishes a set of commands used to specify the low-power design intent for electronic systems. Using UPF commands, you can specify the power domains of the design and the desired multivoltage implementation and verification strategies for each domain, including the domain's isolation cells, retention cells, and level shifters. During a compile operation, Design Compiler automatically inserts these power management cells according to the specified strategies. The same set of UPF commands can be used throughout the design, analysis, verification,

and implementation flow. A Power Compiler license is required for using UPF commands in Design Compiler.

Power optimization for multicorner-multimode designs is supported. However, it is only supported in Design Compiler Graphical. Use the `set_leakage_optimization` and `set_dynamic_optimization` commands to set the leakage and dynamic power constraints on specific scenarios of a multicorner-multimode design.

For more information about using UPF commands, and for details about multicorner-multimode design optimization and the UPF flow for multicorner-multimode design optimization, see the *Power Compiler User Guide*.

Support for Multicorner-Multimode Designs

Designs are often required to operate under multiple modes, such as test or standby mode, and multiple operating conditions, sometimes referred to as corners. Such designs are referred to as multicorner-multimode designs. Design Compiler Graphical can analyze and optimize across multiple modes and corners concurrently. The multicorner-multimode feature in Design Compiler Graphical provides compatibility between flows in Design Compiler and IC Compiler.

To define your modes and corners, you use the `create_scenario` command. A scenario definition usually includes commands that specify the TLUPlus libraries, operating conditions, and constraints.

For details about setting up multicorner-multimode analysis, see Optimizing Multicorner-Multimode Designs in Design Compiler Graphical.

7

Defining Design Constraints

Constraints are declarations that define your design's goals in measurable circuit characteristics such as timing, area, and capacitance. Without constraints, the Design Compiler tool cannot effectively optimize your design.

To learn about design constraints and how to define them, see

- Design Compiler Constraint Types
- Design Rule Constraints
- Optimization Constraints
- Managing Constraint Priorities
- Reporting Constraints
- Propagating Constraints in Hierarchical Designs

Design Compiler Constraint Types

When Design Compiler optimizes your design, it uses two types of constraints:

- Design rule constraints

  These are implicit constraints; the logic library defines them. These constraints are requirements for a design to function correctly, and they apply to any design using the library. You can make these constraints more restrictive than optimization constraints.

- Optimization constraints

  These are explicit constraints; you define them. Optimization constraints apply to the design on which you are working for the duration of the dc_shell session and represent the design's goals. They must be realistic.

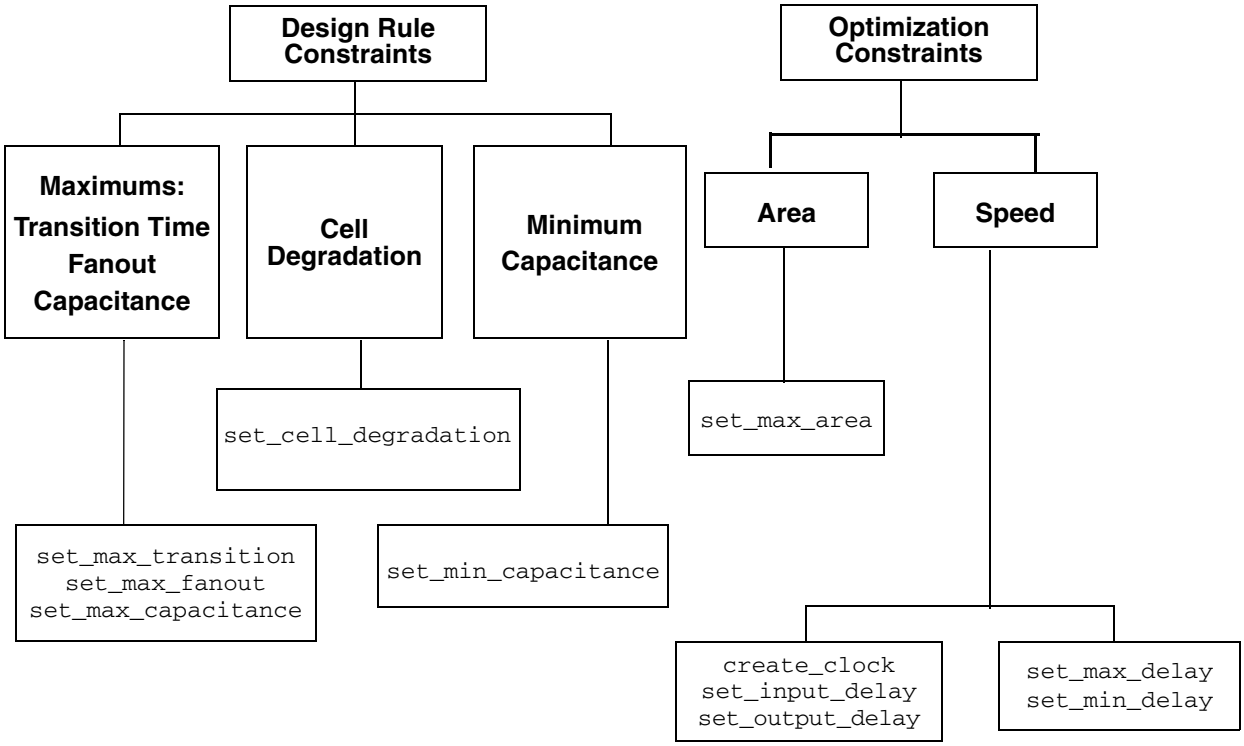
Design Compiler tries to meet both design rule constraints and optimization constraints, but design rule constraints take precedence.

Design Compiler calculates two cost functions: one for design rule constraints and one for optimization constraints. During gate-level optimization, Design Compiler reports the value of each cost function whenever a change is made to the design.

You specify constraints interactively on the command line or in a constraints file.

Figure 7-1 shows the major Design Compiler design rule constraints and optimization constraints and the dc_shell interface commands to set the constraints.

Figure 7-1 Major Design Compiler Constraints



Note:

If you use the Power Compiler tool from Synopsys, `max_dynamic_power` and `max_leakage_power` are optimization constraints. To use these constraints, you need a Power Optimization license and supporting libraries characterized for power.

Design Rule Constraints

Design rule constraints reflect technology-specific restrictions that your design must meet to function as intended. Design rules constrain the nets of a design but are associated with the pins of cells from a logic library. Most technology libraries specify default design rules. Design Compiler cannot violate design rule constraints, even if it means violating optimization constraints (delay and area goals). You can apply more restrictive design rules, but you cannot apply less restrictive ones.

The design rule constraints comprise

- Maximum transition time
- Maximum fanout

- Minimum and maximum capacitance
- Cell degradation

You cannot remove `max_transition`, `max_fanout`, `max_capacitance`, and `min_capacitance` attributes set in a logic library, because they are requirements for the technology, but you can set values that are more restrictive. If both implicit and explicit values are set on a design or a port, the more restrictive value applies. You can remove values you have set.

Design Rule Cost Function

Design Compiler helps you fix design rule violations. If there are multiple violations, Design Compiler tries to fix the violation with the highest priority first. When possible, it also evaluates and selects alternatives that reduce violations of other design rules. Design Compiler uses the same approach with delay violations.

Figure 7-2 shows the design rule cost function equation.

Figure 7-2 Design Rule Cost Equation

$$\sum_{i=1}^{m} max(d_i,\ 0) \times w_i$$

i = Index
d = Delta Constraint
m = Total Number of Constraints
w = Constraint Weight

In a compilation report, the DESIGN RULE COST field reports the cost function for the design rule constraints on the design.

Maximum Transition Time

Maximum transition time is a design rule constraint. The maximum transition time for a net is the longest time required for its driving pin to change logic values. Many technology libraries contain restrictions on the maximum transition time for a pin, creating an implicit transition time limit for designs using that library.

Design Compiler attempts to make the transition time of each net less than the `max_transition` value, for example, by buffering the output of the driving gate. The `max_transition` value can vary with the operating frequency of a cell.

Transition times on nets are computed by use of timing data from the logic library.

To change or add to the implicit transition time values from a logic library, use the `set_max_transition` command.

If both a library max_transition and a design max_transition attribute are defined, Design Compiler tries to meet the smaller (more restrictive) value.

If your design uses multiple technology libraries and each has a different default max_transition value, Design Compiler uses the smallest max_transition value globally across the design.

Defining Maximum Transition Time

The set_max_transition command sets the max_transition attribute to the specified value on clock groups, ports, or designs. During compile, Design Compiler attempts to ensure that the transition time for a net is less than the specified value.

Example

To set a maximum transition time of 3.2 for the design adder, enter the following command:

```
dc_shell> set_max_transition 3.2 [get_designs adder]
```

To undo a set_max_transition command, use remove_attribute. Enter the following command:

```
dc_shell> remove_attribute [get_designs adder] max_transition
```

Specifying Clock-Based Maximum Transition

The max_transition value can vary with the operating frequency of a cell. The operating frequency of a cell is defined as the highest clock frequency on the registers driving a cone of logic (clock frequencies are defined with the create_clock command).

For designs with multiple clock domains, you can use the set_max_transition command to set the max_transition attribute on pins in a specific clock group.

Design Compiler follows these rules in determining the max_transition value:

- When the max_transition attribute is set on a design or port and a clock group, the most restrictive constraint is used.
- If multiple clocks launch the same paths, the most restrictive constraint is used.
- If max_transition attributes are already specified in a logic library, the tool automatically attempts to meet these constraints during compile.

For example the following command sets a max_transition value of 5 on all pins belonging to the Clk clock group:

```
dc_shell> set_max_transition 5 [get_clocks Clk]
```

Maximum Fanout

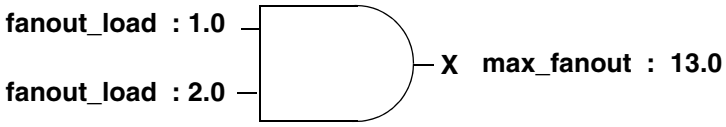
Maximum fanout is a design rule constraint. Most technology libraries place fanout restrictions on driving pins, creating an implicit fanout constraint for every driving pin in designs using that library.

You can set a more conservative fanout constraint on an entire library or define fanout constraints for specific pins in the library description of an individual cell.

If a library fanout constraint exists and you specify a `max_fanout` attribute, Design Compiler tries to meet the smaller (more restrictive) value.

Design Compiler models fanout restrictions by associating a `fanout_load` attribute with each input pin and a `max_fanout` attribute with each output (driving) pin on a cell. Figure 7-3 shows these fanout attributes.

Figure 7-3 fanout_load and max_fanout Attributes



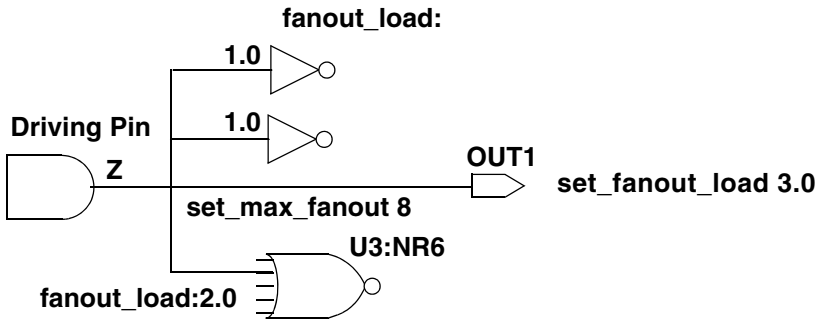
To evaluate the fanout for a driving pin (such as X in Figure 7-3), Design Compiler calculates the sum of all the `fanout_load` attributes for inputs driven by pin X and compares that number with the number of `max_fanout` attributes stored at the driving pin X.

- If the sum of the fanout loads is not more than the `max_fanout` value, the net driven by X is valid.
- If the net driven by X is not valid, Design Compiler tries to make that net valid, perhaps by choosing a higher-drive component.

Maximum Fanout Calculation Example

Figure 7-4 shows how maximum fanout is calculated.

Figure 7-4 Calculation of Maximum Fanout



You can set a maximum fanout constraint on every driving pin and input port as follows:

```
dc_shell> set_max_fanout 8 [get_designs ADDER]
```

To check whether the maximum fanout constraint is met for driving pin Z, Design Compiler compares the specified `max_fanout` attribute against the fanout load.

In this case, the design meets the constraints.

$$\underbrace{8 \geq 1.0 + 1.0 + 3.0 + 2.0}_{\text{Total Fanout Load}}$$
$$7$$

The fanout load imposed by a driven cell (U3) is not necessarily 1.0. Library developers can assign higher fanout loads (for example, 2.0) to model internal cell fanout effects.

You can also set a fanout load on an output port (OUT1) to model external fanout effects.

Fanout load is a dimensionless number, not a capacitance. It represents a numerical contribution to the total effective fanout.

Defining Maximum Fanout

The `set_max_fanout` command sets the maximum allowable fanout load for the listed input ports. The `set_max_fanout` command sets a `max_fanout` attribute on the listed objects.

To undo a maximum fanout value set on an input port or design, use the `remove_attribute` command. For example,

```
dc_shell> remove_attribute [get_ports port_name] max_fanout
dc_shell> remove_attribute [get_designs design_name] max_fanout
```

Defining Expected Fanout for Output Ports

The set_fanout_load command sets the expected fanout load value for listed output ports.

Design Compiler adds the fanout value to all other loads on the pin driving each port in *port_list* and tries to make the total load less than the maximum fanout load of the pin.

To undo a set_fanout_load command set on an output port, use the remove_attribute command. For example,

```
dc_shell> remove_attribute port_name fanout_load
```

To determine the fanout load, use the get_attribute command.

Examples

To find the fanout load on the input pin of library cell AND2 in library libA, enter

```
dc_shell> get_attribute "libA/AND2/i" fanout_load
```

To find the default fanout load set on logic library libA, enter

```
dc_shell> get_attribute libA default_fanout_load
```

Maximum Capacitance

Maximum capacitance is a design rule constraint. It is set as a pin-level attribute that defines the maximum total capacitive load that an output pin can drive. That is, the pin cannot connect to a net that has a total capacitance (load pin capacitance and interconnect capacitance) greater than or equal to the maximum capacitance defined at the pin.

The maximum capacitance design rule constraint allows you to control the capacitance of nets directly. (The design rule constraints max_fanout and max_transition limit the actual capacitance of nets indirectly.) The max_capacitance constraint operates similarly to max_transition, but the cost is based on the total capacitance of the net, rather than the transition time. The max_capacitance attribute functions independently, so you can use it with max_fanout and max_transition.

The max_capacitance value can vary with the operating frequency of a cell. You can have Design Compiler annotate each driver pin with a frequency-based max_capacitance value by setting the compile_enable_dyn_max_cap variable to true. For more information, see Specifying Frequency-Based Maximum Capacitance.

The max_capacitance constraint has priority over the cell degradation constraint.

If both a library max_capacitance attribute and a design max_capacitance attribute exist, Design Compiler tries to meet the smaller (more restrictive) value.

Defining Maximum Capacitance

The set_max_capacitance command sets a maximum capacitance for the nets attached to named ports or to all the nets in a design. This command allows you to control capacitance directly and places a max_capacitance attribute on the listed objects.

Design Compiler calculates the capacitance on a net by adding the wire capacitance of the net to the capacitance of the pins attached to the net. To determine whether a net meets the capacitance constraint, Design Compiler compares the calculated capacitance value with the max_capacitance value of the pin driving the net.

Use the set_max_capacitance command to specify a capacitance value on input ports or designs. This value should be less than or equal to the max_capacitance value of the pin driving the net.

Examples

To set a maximum capacitance of 3 for the design adder, enter one the following command:

```
dc_shell> set_max_capacitance 3 [get_designs adder]
```

To undo a set_max_capacitance command, use remove_attribute. For example, enter the following command:

```
dc_shell> remove_attribute [get_designs adder] max_capacitance
```

Specifying Frequency-Based Maximum Capacitance

The max_capacitance value can vary with the operating frequency of a cell. You can have Design Compiler annotate each driver pin with a frequency-based max_capacitance value by setting the compile_enable_dyn_max_cap variable to true.

Before you use this feature, your logic library should be characterized for multiple frequencies. This characterization consists of associating a max_capacitance value with each driver pin for each frequency and capturing this information in a one-dimensional lookup table. For information about creating the max_capacitance lookup table, see the Library Compiler documentation.

Design Compiler follows these steps in determining the max_capacitance value:

1. It identifies the operating frequency of each cell.

   The operating frequency of a cell is defined as the highest clock frequency on the registers driving a cone of logic (clock frequencies are defined with the create_clock command).

2. It looks up the corresponding max_capacitance value for the identified frequency from the lookup table in the logic library.

3. It annotates each driver pin with the appropriate max_capacitance value and uses these values in synthesis.

Minimum Capacitance

Minimum capacitance is a design rule constraint. Some technology libraries specify minimum capacitance. The min_capacitance design rule specifies the minimum load a cell can drive. It specifies the lower bound of the range of loads with which a cell has been characterized to operate. During optimization, Design Compiler ensures that the load driven by a cell meets the minimum capacitance requirement for that cell. When a violation occurs, Design Compiler fixes the violation by sizing the driver.

You can use the min_capacitance design rule with the existing design rules. The min_capacitance design rule has higher priority than the maximum transition time, maximum fanout, and maximum capacitance constraints.

You can set a minimum capacitance for nets attached to input ports to determine whether violations occur for driving cells at the input boundary. If violations are reported after compilation, you can fix the problem by recompiling the module driving the ports. Use the set_min_capacitance command to set minimum capacitance on input or inout ports; you cannot set minimum capacitance on a design.

If library min_capacitance and design min_capacitance attributes both exist, Design Compiler tries to meet the larger (more restrictive) value.

Defining Minimum Capacitance

The set_min_capacitance command sets a defined minimum capacitance value on listed input or bidirectional ports. To set a minimum capacitance for nets attached to input or bidirectional ports, use the set_min_capacitance command.

Example

To set a minimum capacitance value of 12.0 units on the port named high_drive, enter

```
dc_shell> set_min_capacitance 12.0 high_drive
```

To report only minimum capacitance constraint information, use the `-min_capacitance` option of the `report_constraint` command.

To get information about the current port settings, use the `report_port` command.

To undo a `set_min_capacitance` command, use the `remove_attribute` command.

Cell Degradation

Cell degradation is a design rule constraint. Some technology libraries contain cell degradation tables. The tables list the maximum capacitance that can be driven by a cell as a function of the transition times at the inputs of the cell.

The `cell_degradation` design rule specifies that the capacitance value for a net is less than the cell degradation value. The `cell_degradation` design rule can be used with other design rules, but the `max_capacitance` design rule has a higher priority than the `cell_degradation` design rule. If the `max_capacitance` rule is not violated, applying the `cell_degradation` design rule does not cause it to be violated.

The `set_cell_degradation` command sets the `cell_degradation` attribute to a specified value on specified input ports.

During compilation, if `cell_degradation` tables are specified in a logic library, Design Compiler tries to ensure that the capacitance value for a net is less than the specified value. The `cell_degradation` tables give the maximum capacitance that a cell can drive, as a function of the transition times at the inputs of the cell.

If cell_degradation tables are not specified in a logic library, you can set `cell_degradation` explicitly on the input ports.

By default, a port has no `cell_degradation` constraint.

Note:
Use of the `set_cell_degradation` command requires a DC Ultra license.

Example

This command sets a maximum capacitance value of 2.0 units on the port named late_riser:

```
dc_shell> set_cell_degradation 2.0 late_riser
```

To get information about optimization and design rule constraints, use the `report_constraint` command.

To remove the `cell_degradation` attribute, use the `remove_attribute` command.

Connection Class

The connection class constraint on a port describes the connection requirements for a given technology. Only loads and drivers with the same connection class label can be legally connected. This constraint can be specified in the library or by the user. To set a connection class constraint on a port, use the `set_connection_class` command.

Managing Design Rule Constraint Priorities

By default, design rule constraints have a higher optimization priority than optimization constraints. You can change that priority, however; see Managing Constraint Priorities.

Precedence of Design Rule Constraints

Design Compiler follows this descending order of precedence when it tries to resolve conflicts among design rule constraints (see Table 7-2):

1. Minimum capacitance
2. Maximum transition
3. Maximum fanout
4. Maximum capacitance
5. Cell degradation

The following details apply to the precedence of design rule constraints:

- Maximum transition has precedence over maximum fanout. If a maximum fanout constraint is not met, investigate the possibility of a conflicting maximum transition constraint. Design Compiler does not make a transition time worse to fix a maximum fanout violation.
- Maximum fanout has precedence over maximum capacitance.

- Design Compiler calculates transition time for a net in two ways, depending on the library.
  - For libraries using the CMOS delay model, Design Compiler calculates the transition time by using the drive resistance of the driving cell and the capacitive load on the net.
  - For libraries using a nonlinear delay model, Design Compiler calculates the transition time by using table lookup and interpolation. This is a function of capacitance at the output pin and of the input transition time.
- The `set_driving_cell` and `set_drive` commands behave differently, depending on your logic library.
  - For libraries using the CMOS delay model, drive resistance is a constant. In this case, the `set_drive` command and the `set_driving_cell` command give the same result.
  - For libraries using a nonlinear delay model, the `set_driving_cell` command calculates the transition time dynamically, based on the load from the tables. The `set_drive` command returns one value when the command is issued and uses a value from the middle of the range in the tables for load.

Both the `set_driving_cell` and the `set_drive` commands affect the port transition delay. The `set_driving_cell` command places the design rule constraints, annotated with the driving cell, on the affected port.

The `set_load` command places a load on a port or a net. The units of this load must be consistent with your logic library. This value is used for timing optimizations, not for maximum fanout optimizations.

Design Rule Scenarios

Typical design rule scenarios are

- `set_max_fanout` and `set_max_transition` commands
- `set_max_fanout` and `set_max_capacitance` commands

Typically, a logic library specifies a default `max_transition` or `max_capacitance`, but not both. To achieve the best result, do not mix `max_transition` and `max_capacitance`.

Disabling DRC Violation Fixing on Special Nets

To enable or disable DRC violation fixing on clock, constant, or scan nets, use the `set_auto_disable_drc_nets` command. The command acts on all the nets of a given type in the current design; that is, depending on the options you specify, it acts on all the clock nets, all the constant nets, all the scan nets, or on any combination of these three net types. Nets that have DRC violation fixing disabled are marked with the `auto_disable_drc_nets` attribute.

By default, the clock and constant nets of a design have DRC violation fixing disabled. This is the same as using the `-default` option of the command; the scan nets do not. You can use the `-all` option to disable DRC violation fixing on all three net types.

Alternatively, you can independently disable or enable DRC violation fixing for the clock nets, constant nets, or scan nets by assigning a `true` or `false` value to the `-on_clock_network`, `-constant`, or `-scan` options, respectively. You can use the `-none` option to enable DRC violation fixing on all three types of nets.

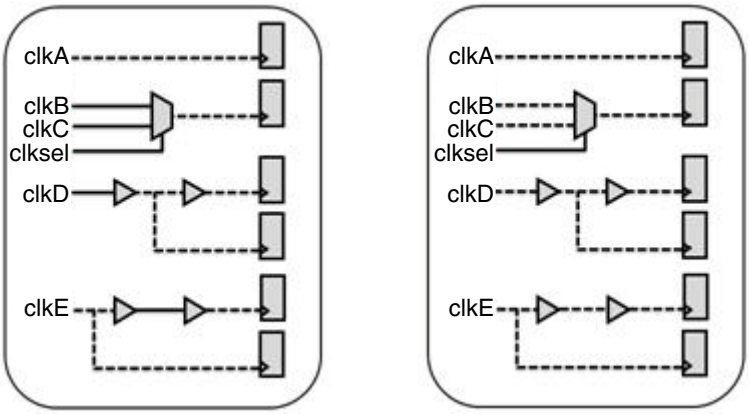
It is recommended that you use the `-on_clock_network` option instead of the `-clock` option to enable or disable DRC violation fixing for the clock network. The `-clock` option is supported for backward compatibility. However, it is set to `true` by default, meaning that it automatically disables DRC violation fixing on clock networks. As shown in Figure 7-5, the `-clock` option starts at register clock pins and propagates back, but not through, the logic on the clock tree. Optimization might buffer non-DRC disabled clock nets upstream of existing clock logic.

To correctly disable DRC violation fixing on the entire clock tree, use the `-on_clock_network` option set to `true`, also shown in Figure 7-5. The option is set to `false` by default. The default setting has not been changed to `true` to avoid affecting existing flows.

If both the `-on_clock_network` option and the `-clock` option are set to `true`, the effect is the same as using the `-on_clock_network` option alone; the `-on_clock_network` option overrides the `-clock` option in this case.

Figure 7-5 shows DRC violation fixing disabled for clock nets by the `-clock` option (on the left) and by the `-on_clock_network` option (on the right). The DRC-disabled clock nets are represented by dashed lines.

Figure 7-5 DRC Violation Fixing Disabled For Clock Nets



DRC violation fixing disabled by `-clock`

DRC violation fixing disabled by `-on_clock_network`

Clock nets are ideal nets by default. Using the `set_auto_disable_drc_nets` command to enable design rule fixing does not affect the ideal timing properties of clock nets. You must use the `set_propagated_clock` command to affect the ideal timing of clock nets.

You cannot use the `set_auto_disable_drc_nets` command to override disabled design rule fixing on ideal networks marked with ideal network attributes. This command never overrides the settings specified by the `set_ideal_net` or `set_ideal_network` command.

Summary of Design Rule Commands and Objects

Table 7-1 summarizes the design rule commands and the objects on which to set them.

Table 7-1 Design Rule Command and Object Summary

Command	Object
`set_max_fanout`	Input ports or designs
`set_fanout_load`	Output ports
`set_load`	Ports or nets
`set_max_transition`	Ports or designs
`set_cell_degradation`	Input ports
`set_min_capacitance`	Input ports

Optimization Constraints

Optimization constraints represent speed and area design goals and restrictions that you want but that might not be crucial to the operation of a design. Speed (timing) constraints have higher priority than area.

By default, optimization constraints are secondary to design rule constraints. However, that priority can be changed; see Managing Constraint Priorities.

The optimization constraints comprise

- Timing constraints (performance and speed)
  - Input and output delays (synchronous paths)
  - Minimum and maximum delay (asynchronous paths)
- Maximum area (number of gates)

Optimization Cost Function

During the first phase of mapping, Design Compiler works to reduce the optimization cost function. Design Compiler evaluates this function to determine whether a change to the design improves the cost.

The full optimization cost function takes into account the following components, listed in order of importance. Not all components are active on all designs.

1. Maximum delay cost
2. Minimum delay cost
3. Maximum area cost

Design Compiler evaluates cost function components independently in order of importance and accepts an optimization move if it decreases the cost of one component without increasing more-important costs. For example, an optimization move that improves maximum delay cost is always accepted.

Optimization stops when all costs are zero or no further improvements can be made to the cost function.

Timing Constraints

When defining timing constraints you should consider that your design has synchronous paths and asynchronous paths. Synchronous paths are constrained by specifying clocks in the design. Use the `create_clock` command to specify a clock. After specifying the clocks, it is recommended you also specify the input and output port timing specifications. Use the `set_input_delay` and `set_output_delay` commands.

Asynchronous paths are constrained by specifying minimum and maximum delay values. Use the `set_max_delay` and `set_min_delay` commands to specify these point-to-point delays. The calculation of minimum and maximum delays are described in the following sections.

For additional information, see the *Synopsys Timing Constraints and Optimization User Guide*.

Maximum Delay

Maximum delay is an optimization constraint. Design Compiler contains a built-in static timing analyzer for evaluating timing constraints. A static timing analyzer calculates path delays from local gate and interconnect delays but does not simulate the design. The Design Compiler timing analyzer performs critical path tracing to check minimum and maximum delays for every timing path in the design. The most critical path is not necessarily the longest combinational path in a sequential design, because paths can be relative to different clocks at path startpoints and endpoints.

The timing analyzer calculates minimum and maximum signal rise and fall path values based on the timing values and environmental information in the logic library.

Cost Calculation

Maximum delay is usually the most important portion of the optimization cost function. The maximum delay optimization cost guides Design Compiler to produce a design that functions at the speed you want.

The maximum delay cost includes multiple parts. Figure 7-6 shows the maximum delay cost equation.

Figure 7-6 Cost Calculation for Maximum Delay

$$\sum_{i=1}^{m} v_i \times w_i$$

i = Index
v = worst violation
m = number of path groups
w = weight

Maximum delay target values for each timing path in the design are automatically determined after considering clock waveforms and skew, library setup times, external delays, multicycle or false path specifications, and `set_max_delay` commands. Load, drive, operating conditions, wire load model, and other factors are also taken into account.

The maximum delay cost is affected by how paths are grouped by the `group_path` and `create_clock` commands.

- If only one path group exists, the maximum delay cost is the cost for that group: the amount of the worst violation multiplied by the group weight.
- If multiple path groups exist, the costs for all the groups are added together to determine the maximum delay cost of the design. The group cost is always zero or greater.

```
Delta = max(delta(pin1), delta(pin2), ... delta(pinN))
```

Minimum Delay

Minimum delay is an optimization constraint, but Design Compiler fixes the minimum delay constraint when it fixes design rule violations. Minimum delay constraints are set explicitly with the `set_min_delay` command or set implicitly due to hold time requirements.

The minimum delay to a pin or port must be greater than the target delay.

Design Compiler considers the minimum delay cost only if the `set_fix_hold` command is used.

If `fix_hold` is not specified on any clocks, the minimum delay cost is not considered during compilation. If `fix_hold` or `min_delay` is specified, the minimum delay cost is a secondary optimization cost.

`set_min_delay`

Defines a minimum delay for timing paths in the design.

`set_fix_hold`

Directs Design Compiler to fix hold violations at registers during compilation. You can override the default path delay for paths affected by `set_fix_hold`, by using the `set_false_path` or `set_multicycle_path` commands.

Hold time violations are fixed only if the `set_fix_hold` command is applied to related clocks.

Cost Calculation

The minimum delay cost for a design is different from the maximum delay cost. The minimum delay cost is not affected by path groups, and all violations contribute to the cost. Figure 7-7 shows the minimum delay cost equation.

Figure 7-7 Cost Calculation for Minimum Delay

$$\sum_{i=1}^{m} v_i$$

i = index
m = number of paths affected by set_min_delay or set_fix_hold
v = minimum delay violation max(0,required_path_delay - actual_path_delay)

The minimum delay cost function has the same delta for single pins or ports and multiple pins or ports.

```
Delta = min_delay - minimum delay(pin or port)
```

Maximum Area

Maximum area is an optimization constraint. Maximum area represents the number of gates in the design, not the physical area the design occupies. Usually the area requirements for the design are stated as the smallest design that meets the performance goal. Defining a maximum area directs Design Compiler to optimize the design for area after timing optimization is complete.

The `set_max_area` command specifies the maximum allowable area for the current design. Design Compiler computes the area of a design by adding the areas of each component on the lowest level of the design hierarchy (and the area of the nets).

Design Compiler ignores the following components when it calculates circuit area:

- Unknown components
- Components with unknown areas
- Technology-independent generic cells

The area of a cell (component) is technology-dependent and obtained from the logic library.

Cost Calculation

The maximum-area cost equation is

```
cost = max (0, current area - max_area)
```

Defining Maximum Area

The `set_max_area` command specifies an area target and places a `max_area` attribute on the current design.

Examples

```
dc_shell> set_max_area 0.0
dc_shell> set_max_area 14.0
```

Determining the smallest design can be helpful. The following script guides Design Compiler to optimize for area only. It constrains a design only for minimum area, when you do not care about timing. For the timing to make sense, you must apply clocking and input and output delay.

```
# Example script for smallest design
remove_constraint -all
remove_clock -all
set_max_area 0
```

Managing Constraint Priorities

During optimization, Design Compiler uses a cost vector to resolve any conflicts among competing constraint priorities. Table 7-2 shows the default order of priorities. The table shows that, by default, design rule constraints have priority over optimization constraints. However, you can reorder the priorities of the constraints listed in **bold** type by using the `set_cost_priority` command.

Table 7-2 Constraints Default Cost Vector

Priority (descending order)	Notes
connection classes	
multiple_port_net_cost	
min_capacitance	Design rule constraint
max_transition	Design rule constraint
max_fanout	Design rule constraint
max_capacitance	Design rule constraint
cell_degradation	Design rule constraint
max_delay	Optimization constraint
min_delay	Optimization constraint
power	Optimization constraint

Table 7-2 Constraints Default Cost Vector (Continued)

Priority (descending order)	Notes
area	Optimization constraint
cell count	

The following are circumstances under which you might want to move the optimization constraint `max_delay` ahead of the maximum design rule constraints.

- In many technology libraries, the only significant design rule violations that cannot be fixed without hurting delay are overconstrained nets, such as input ports with large external loads or around logic marked `dont_touch`. Placing `max_delay` ahead of the design rule constraints in priority allows these design rule constraint violations to be fixed in a way that does not hurt delay. Design Compiler might, for example, resize the drivers in another module.
- In compilation of a small block of logic, such as an extracted critical region of a larger design, the possibility of overconstraints at the block boundaries is high. In this case, design rule fixing might better be postponed until the small block has been regrouped into the larger design.

The syntax is

`set_cost_priority [-default] [-delay]` *`cost_list`*

`-default`

Directs Design Compiler to use its default priority, as shown in Constraints Default Cost Vector.

`-delay`

Specifies that `max_delay` has higher priority than the maximum design rule constraints.

`cost_list`

Specifies the order of priority (listing the highest first) of the following costs: `max_delay`, `min_delay`, `max_transition`, `max_fanout`, `max_capacitance`, `cell_degradation`, and `max_design_rules`.

Note:

Use of the `cost_list` option requires a DC Ultra license.

Examples

To prioritize max_delay ahead of the maximum design rule constraints, enter

```
dc_shell> set_cost_priority -delay
```

To assign top priority to max_capacitance, max_delay, and max_fanout—in that order—enter

```
dc_shell> set_cost_priority {max_capacitance max_delay max_fanout}
```

Design Compiler assigns any costs you do not list to a lower priority than the costs you do list. This example does not list max_transition, cell_degradation, or min_delay, so Design Compiler assigns them priority following max_fanout.

If you specify set_cost_priority more than one time on a design, Design Compiler uses the most recent setting.

Reporting Constraints

The report_constraint command reports the constraint values in the current design, enabling you to check design rules and optimization goals.

The constraint report lists the following for each constraint in the current design:

- Whether the constraint was met or violated
- By how much the constraint value was met or violated
- The design object that is the worst violator
- The maximum delay information, showing cost by path group. This includes violations of setup time on registers or ports with output delay as well as violations of the set_max_delay command.
- The minimum delay cost, which includes violations of hold time on registers or ports with output delay as well as violations of the set_min_delay command.

The constraint report summarizes the constraints in the order in which you set the priorities. Constraints not present in your design are not included in the report. To list more information in a constraint report, use the -verbose option of the report_constraint command. To list all constraint violators, use the report_constraint command with the -all_violators option.

Propagating Constraints in Hierarchical Designs

Hierarchical designs are composed of subdesigns. You can propagate constraints up or down the hierarchy in the following ways:

- Characterizing

  Captures information about the environment of specific cell instances and assigns the information as attributes on the design to which the cells are linked.

- Modeling

  Creates a characterized design as a library cell.

- Propagating constraints up the hierarchy

  Propagates clocks, timing exceptions, and disabled timing arcs from lower level subdesigns to the current design.

Characterizing Subdesigns

When you compile subdesigns separately, boundary conditions such as the input drive strengths, input signal delays (arrival times), and output loads can be derived from the parent design and set on each subdesign. You can do this in the following ways:

- Manually

  Use the `set_drive`, `set_driving_cell`, `set_input_delay`, `set_output_delay`, and `set_load` commands.

- Automatically

  Use the `characterize` command or the design budgeting tool.

Using the characterize Command

The `characterize` command places on a design the information and attributes that characterize its environment in the context of a specified instantiation in the top-level design.

The primary purpose of `characterize` is to capture the timing environment of the subdesign. This occurs when you use `characterize` with no arguments or when you use its `-constraints`, `-connections`, or `-power` options.

The `characterize` command derives and asserts the following information and attributes on the design to which the instance is linked:

- Unless the `-no_timing` option is specified, the `characterize` command places on the subdesigns any timing characteristics previously set by the following commands:

`create_clock`	`set_load`
`group_path`	`set_max_delay`
`read_timing`	`set_max_time_borrow`
`set_annotated_check`	`set_min_delay`
`set_annotated_delay`	`set_multicycle_path`
`set_auto_disable_drc_nets`	`set_operating_conditions`
`set_drive`	`set_output_delay`
`set_driving_cell`	`set_resistance`
`set_false_path`	`set_timing_ranges`
`set_ideal_net`	`set_wire_load_model`
`set_ideal_network`	`set_wire_load_mode`
`set_input_delay`	`set_wire_load_selection_group`
	`set_wire_load_min_block_size`

- If you specify the `-constraint` option, the `characterize` command places on the subdesigns any area, power, connection class, and design rule constraints previously set by the following commands:

`set_cell_degradation`	`set_max_fanout`
`set_connection_class`	`set_max_power`
`set_dont_touch_network`	`set_max_transition`
`set_fanout_load`	`set_min_capacitance`
`set_max_area`	`set_max_capacitance`

- If you specify the `-connection` option, the `characterize` command places on the subdesigns the connection attributes set by the following commands. Connection class information is applied only when you use the `-constraint` option.

`set_equal`	`set_logic_zero`
`set_logic_dc`	`set_opposite`
`set_logic_one`	`set_unconnected`

- If you specify the `-power` option, the `characterize` command places on the subdesigns the switching activity information, toggle rates, and static probability previously set, calculated, or saved by the following commands:

`report_power`	`set_switching_activity`

Removing Previous Annotations

In most cases, characterizing a design removes the effects of a previous characterization and replaces the relevant information. However, in the case of back-annotation (`set_load`, `set_resistance`, `read_timing`, `set_annotated_delay`, `set_annotated_check`), the characterize step removes the annotations and cannot overwrite existing annotations made on the subdesign. In this case, you must explicitly remove annotations from the subdesign (using `reset_design`) before you run the `characterize` command again.

Optimizing Bottom Up Versus Optimizing Top Down

During optimization, you can use `characterize` with `set_dont_touch` to maintain hierarchy. This is known as bottom-up optimization, which you can apply by using a golden instance or a uniquify approach (either manually with the `uniquify` command or automatically as part of the `compile` command). An alternative to bottom-up optimization is top-down optimization, also called hierarchical compile. During top-down optimization, the tool automatically performs characterization and optimization for subdesigns.

Deriving the Boundary Conditions

The `characterize` command automatically derives the boundary conditions of a subdesign based on its context in a parent design. It examines an instance's surroundings to obtain actual drive, load, and timing parameters and computes the following types of boundary conditions:

- Timing conditions

  Expected signal delays at input ports.

- Constraints

  Inherited requirements from the parent design, such as maximum delay.

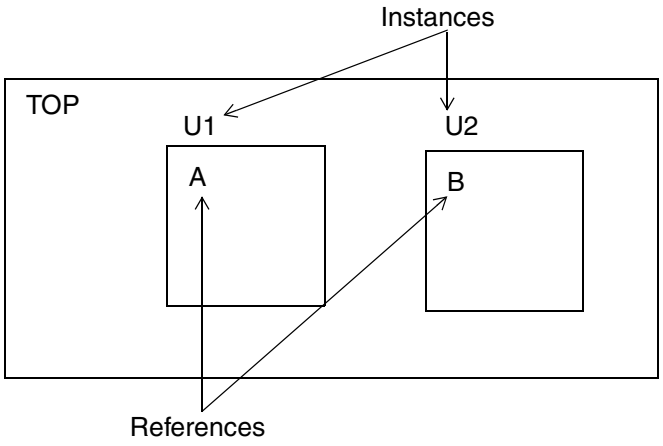
- Connection relations

  Logical relationships between ports or between ports and power or ground, such as always logic 0, logical opposite of another port, or unconnected.

The `characterize` command summarizes the boundary conditions for one instance of a subdesign in one invocation. The result is applied to the reference.

Figure 7-8 shows instances and references.

Figure 7-8 Instances and References



If a subdesign is used in more than one place, you must either characterize it manually or create a copy of the design for each instantiation and characterize each.

For more information, see Characterizing Multiple Instances.

Saving Attributes and Constraints

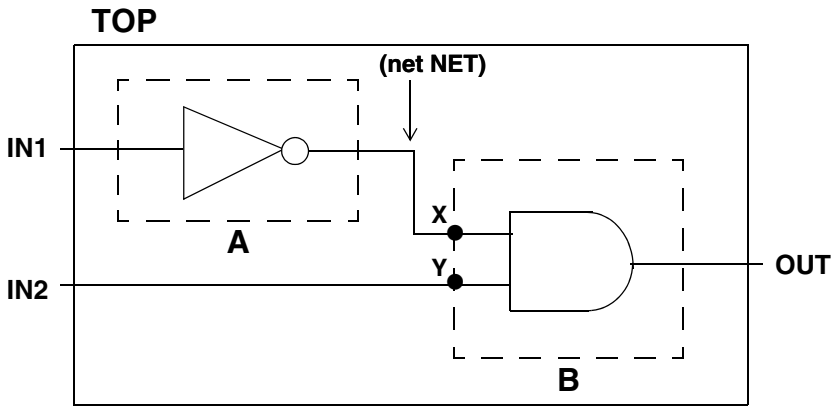
The `characterize` command captures the timing environment of the subdesign. Use the `write_script` command with `characterize` to save the attributes and constraints for the current design. For more information about saving and re-creating attribute values, see Saving Attribute Values.

The characterize Command Calculations

The `characterize` command derives specific load and timing values for ports. The command uses values that allow the timing numbers on the output ports of a characterized design to be the same as if the design were flattened and then timed.

The Load Calculations and Input Delay Calculations sections use the hierarchical design example in Figure 7-9 to describe the load and input delay calculations used by the `characterize` command.

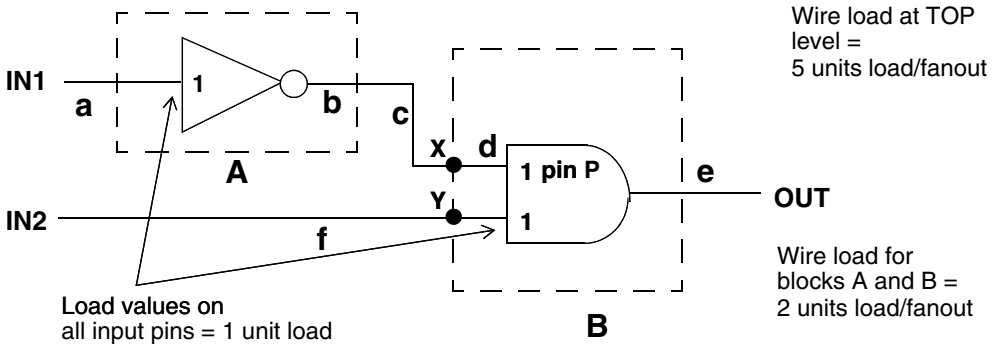
Figure 7-9 Hierarchical Design Example



Load Calculations

Figure 7-10 provides values for wire loads and input pin capacitances for the example shown in Figure 7-9.

Figure 7-10 Hierarchical Design With Annotated Loads



In Figure 7-10, a through f are wire segments used in the calculations. This example assumes a segmented wire loading model, which takes the interconnection net loads on the blocks into account and uses a linear function for the wire loads.

- The calculation for the outside load of pin P of hierarchical block B is

```
outside load = sum of the loads of all pins on the net
     loading P that are not in B
     plus the sum of the loads of all segments of net
     driving or loading P that are not in B
```

- The calculation for each segment's load is

```
segment load = number of fanouts * wire load
```

  - The calculation for the outside load on input IN1 to block A uses 0 driving pins, a fanout count of 1 for segment a, and the TOP wire load of 5 loads per fanout.

    The calculation is

```
load pins on driving net + load of segment a
     = 0 + (1 * 5)
     = 5
```

- The calculation for the outside load on the output of block A is

```
load pins on net
           + load of segment c
           + load of segment d
      = 1 (for load pin P)
      + (1 * 5)
      + (1 * 2)
      = 8
```

For each segment in the calculation, the local wire load model is used to calculate the load. That is, the calculation for block A's output pin uses TOP's wire load of 5 loads per fanout for segment c and block B's wire load of 2 loads per fanout for segment d.

- The calculation for the outside load on the output of block B is

```
load pins on net + load of segment e
     = 0 + (1 * 5)
     = 5
```

- The calculation for the outside load on the input pin X of block B is

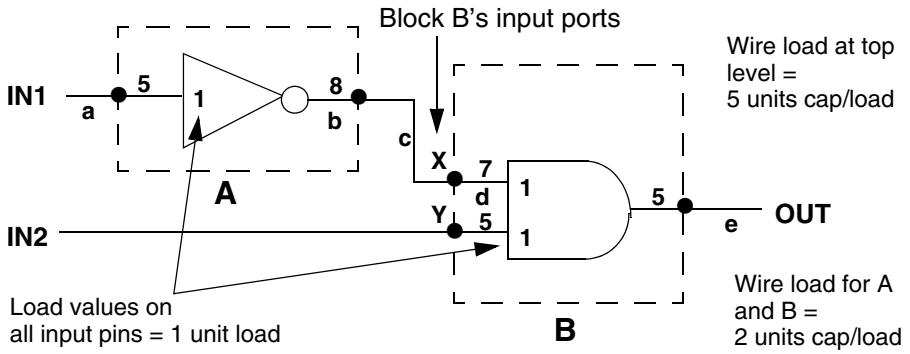
```
load pins on net
          + load of segment b
          + load of segment c
          = 0 + (1 * 2) + (1 * 5)
          = 7
```

- The calculation for the outside load on the input pin Y is

```
pin loads on driving net + load of segment f
     = 0 + 1 * 5
     = 5
```

Figure 7-11 shows the modified version of Hierarchical Design Example with the outside loads annotated.

Figure 7-11 Loads After Characterization



Input Delay Calculations

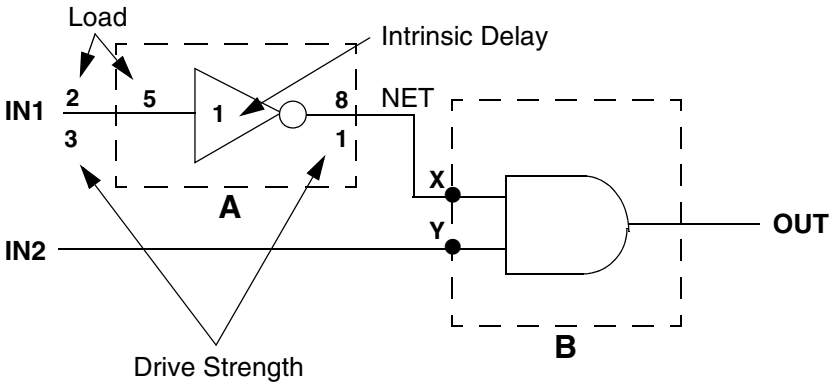
Because characterizing provides accurate details of outside loads, the path delays of input signals reflect only the delay through the intrinsic delay of the last gate driving the port. The path delays of input signals do not include the gate's load delay or the connect delay on the net.

For example, the characterized input delay on the input pins of block B is calculated from the delay to the pin that drives the port being characterized, without the gate's load delay or the connect delay on the net.

The timing calculations for characterizing block B follow Figure 7-12.

Figure 7-12 shows the default drive strengths and intrinsic delays of block A and signal IN1.

Figure 7-12 Design With Annotations for Timing Calculations



The delay calculation for input pin X is

```
drive strength at IN1 * (wire load + pin load)
          + intrinsic delay of A's
cell
          = 3 * (5 + 2 + 1)
          + 1
          = 25
```

Characterizing Subdesign Port Signal Interfaces

The `characterize` command with no options replaces a subdesign's port signal information (clocks, port drive, input and output delays, maximum and minimum path delays, and port load) with information derived from the parent design. The subdesign also inherits operating conditions and timing ranges from the top-level design.

You can manually set port signal information by using the following commands:

```
create_clock
set_clock_latency
```

```
set_clock_uncertainty
set_drive
set_driving_cell
set_input_delay
set_load
set_max_delay
set_min_delay
set_output_delay
set_propagated_clock
```

The `characterize` command sets the wire loading model selection group and model for subdesigns. The subdesigns inherit the top-level wire loading mode. The wire loading model for subdesigns is determined as follows:

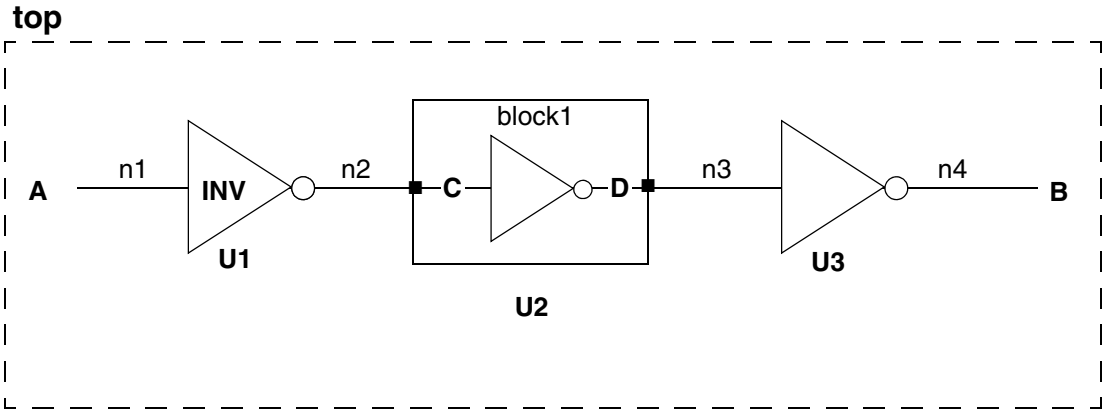
- If the top-level mode is top, the subdesigns inherit the top-level wire loading model. Unless an instance specific model or selection group has be specified. Instance specific model and selection group settings take precedence over design level settings.
- If the top-level mode is enclosed or segmented, the wire loading model is based on the following:
  - If no wire loading model is defined for the lower block and the wire load cannot be determine by the area, the wire loading model of the top-level design is used.
  - If no wire loading model is defined for the lower block but the wire load can be determine by the area, the wire loading model is reselected at compile time, based on the cell area.

Combinational Design Example

Figure 7-13 shows subdesign block in the combinational design top. Set the port interface attributes either manually or automatically.

- Script 1 uses characterize to set the attributes automatically.
- Script 2 sets the attributes manually.

Figure 7-13 Characterizing Drive, Timing, and Load Values—Combinational Design




```
current_design top
set_input_delay 0 A
set_max_delay 10 -to B
```

Script 1

```
current_design top
characterize U2
```

Script 2

```
current_design block1
set_driving_cell -lib_cell INV {C}
set_input_delay 3.3 C
set_load    1.3 D
set_max_delay 9.2 -to D
current_design top
```

In Script 2,

Line 2 captures driving cell information.

Line 3 sets the arrival time of net n2 as 3.3.

Line 4 sets the load of U3 as 1.3.

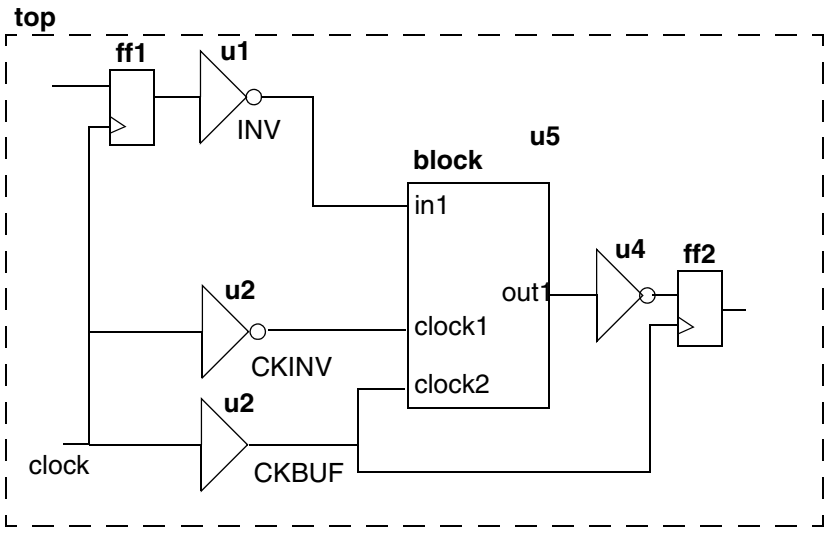
Line 5 sets the inherited `set_max_delay`, which is 10 – .8 (.4 for each inverter).

Sequential Design Example

Figure 7-14 shows subdesign block in the sequential design top. Set the port interface information either manually or automatically.

- Script 1 sets the information manually.
  - The input delay (from ff1/CP to u5/in1) is 1.8
  - The output delay (through u4 plus the setup time of ff2) is 1.2
  - The outside load on the out1 net is 0.85
- Script 2 uses the `characterize` command to set the information automatically.

Figure 7-14 Characterizing Sequential Design Drive, Timing, and Load Values



Script 1

```
current_design top
create_clock -period 10 -waveform {0 5} clock
current_design block
create_clock -name clock -period 10 -waveform {0 5} clock1
create_clock -name clock_bar -period 10 \
   -waveform {5 10} clock2
set_input_delay -clock clock 1.8 in1
set_output_delay -clock clock 1.2 out1
set_driving_cell -lib_cell INV -input_transition_rise 1 in1
set_driving_cell -lib_cell CKINV clock1
set_driving_cell -lib_cell CKBUF clock2
set_load 0.85 out1
current_design top
```

Script 2

```
current_design top
create_clock -period 10 -waveform {0 5} clock
characterize u5
```

Characterizing Subdesign Constraints

The `characterize -constraints` command uses values derived from the parent design to replace the `max_area`, `max_fanout`, `fanout_load`, `max_capacitance`, and `max_transition` attributes of a subdesign.

Characterizing Subdesign Logical Port Connections

The `characterize -connections` command uses connection attributes derived from the parent design to replace the port connection attributes of a subdesign.

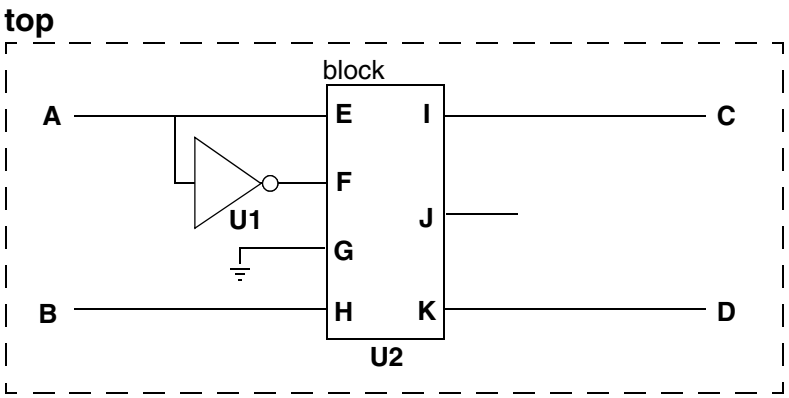
The connection attributes are those set by the commands `set_equal`, `set_opposite`, `set_logic_one`, `set_logic_zero`, and `set_unconnected`.

Example

Figure 7-15 shows subdesign block in design top. Set logical port connections either manually or automatically.

- Script 1 sets the attributes manually.
- Script 2 uses `characterize -no_timing -connections` to set the attributes automatically. The `-no_timing` option inhibits computation of port timing information.

Figure 7-15 Characterizing Port Connection Attributes



Script 1

```
current_design block
set_opposite    { E F }
set_logic_zero  { G }
set_unconnected { J }
```

Script 2

```
current_design top
characterize U2 -no_timing -connections
```

Characterizing Multiple Instances

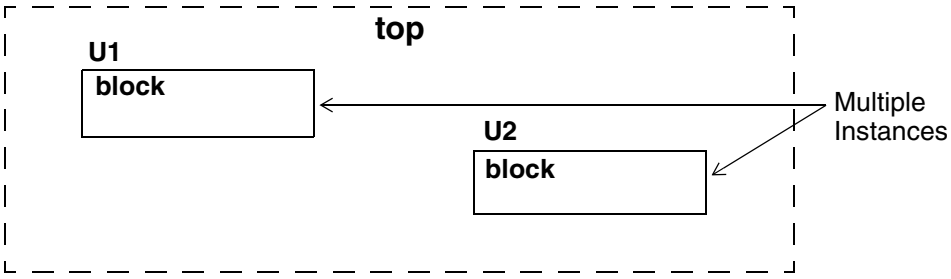
The `characterize` command summarizes the boundary conditions for one instance of a subdesign in each invocation. If a subdesign is used more than one time, use the `uniquify` command to make each instance distinctive before using `characterize` for each instance.

The `uniquify` command creates copies of subdesigns that are referenced more than one time. It then renames the copies and updates the corresponding cell references.

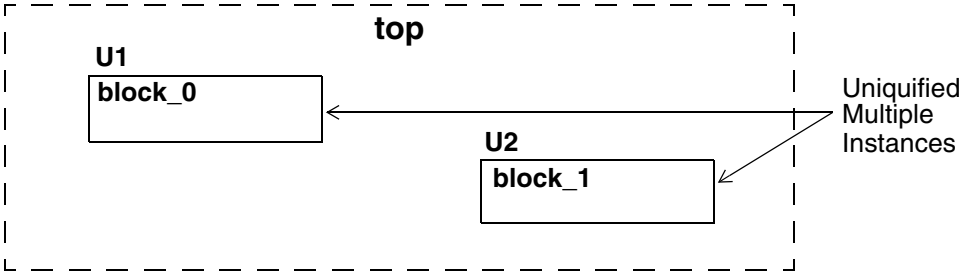
Example

Figure 7-16 shows how to use `uniquify` and `characterize` for the subdesign block, which is referenced in cells U1 and U2.

Figure 7-16 Characterizing a Subdesign Referenced Multiple Times




```
dc_shell> current_design top
dc_shell> uniquify -reference block
```




```
dc_shell> characterize U1
dc_shell> characterize U2
```

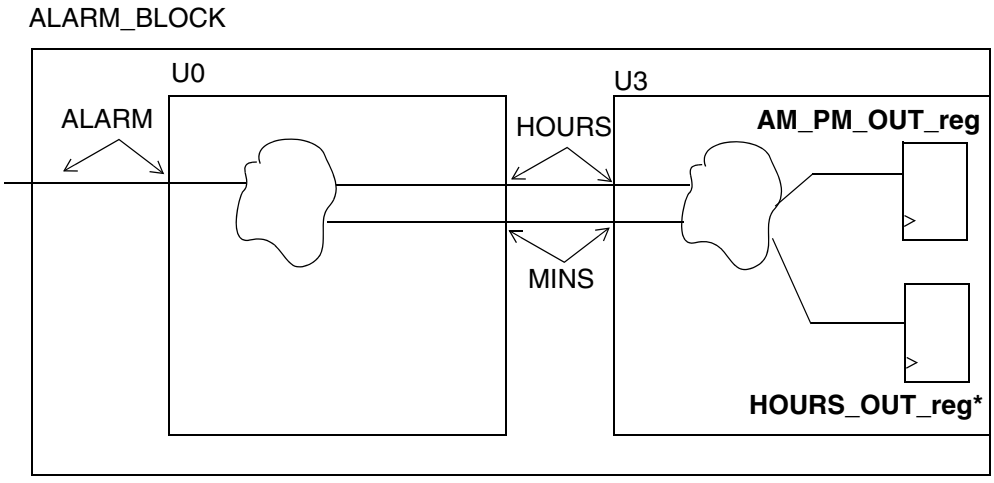
Characterizing Designs With Timing Exceptions

When paths crossing design hierarchies contain different timing exceptions, `characterize` creates timing constraints with virtual clocks to capture this information.

Possible timing exceptions include `set_multicycle_path`, `set_false_path`, `set_max_delay`, and `set_min_delay`. The virtual clock scheme can also handle multiple clocks.

Figure 7-17 shows a sample design in which the path from the ALARM input port to HOURS_OUT_reg* in U3 is constrained as a two-cycle path. Example 7-1 shows the a portion of the uncompressed `characterize -verbose` result of U0 block.

Figure 7-17 characterize -verbose Result of U0 Block



Example 7-1 characterize -verbose Output

```
create_clock -period 10 -waveform {0 5} [get_ports {CLK}]
create_clock -name "CLK_virtual1" -period 10 -waveform {0 5}
create_clock -name "CLK_virtual2" -period 10 -waveform {0 5}
create_clock -name "CLK_virtual3" -period 10 -waveform {0 5}
create_clock -name "CLK_virtual4" -period 10 -waveform {0 5}
create_clock -name "CLK_virtual5" -period 10 -waveform {0 5}
set_input_delay 2 -clock "CLK" [get_ports {MINUTES_BUTTON}]
set_input_delay 2 -clock "CLK" [get_ports {HOURS_BUTTON}]
set_input_delay 2 -clock "CLK_virtual1" [get_ports {ALARM_BUTTON}]
set_output_delay 7.62796 -max -rise -clock "CLK" [get_ports {MINS}]
set_output_delay 6.93955 -max -fall -clock "CLK" [get_ports {MINS}]
set_output_delay 2.20324 -min -rise -clock "CLK" [get_ports {MINS}]
set_output_delay 2.40013 -min -fall -clock "CLK" [get_ports {MINS}]
set_output_delay 8.05575 -add_delay -max -rise -clock "CLK_virtual2" \
   [get_ports {MINS}]
set_output_delay 6.83933 -add_delay -max -fall -clock "CLK_virtual2" \
   [get_ports {MINS}]
set_output_delay 2.89679 -add_delay -min -rise -clock "CLK_virtual2" \
   [get_ports {MINS}]
set_output_delay 2.80452 -add_delay -min -fall -clock "CLK_virtual2" \
   [get_ports {MINS}]
...
set_output_delay 10.232 -add_delay -max -rise -clock "CLK_virtual5" \
   [get_ports {MINS}]
set_output_delay 9.4791 -add_delay -max -fall -clock "CLK_virtual5" \
   [get_ports {MINS}]
set_output_delay 3.90222 -add_delay -min -rise -clock "CLK_virtual5" \
   [get_ports {MINS}]
set_output_delay 3.57818 -add_delay -min -fall -clock "CLK_virtual5" \
   [get_ports {MINS}]
...
set_multicycle_path 2 -from [get_clocks {CLK_virtual1}] -to \
   [get_clocks {CLK_virtual2}]
set_multicycle_path 2 -from [get_clocks {CLK_virtual1}] -to \
   [get_clocks {CLK_virtual3}]
set_multicycle_path 2 -from [get_clocks {CLK_virtual1}] -to \
   [get_clocks {CLK_virtual4}]
set_multicycle_path 2 -from [get_clocks {CLK_virtual1}] -to \
   [get_clocks {CLK_virtual5}]
```

Limitations of the characterize Command

The `characterize` command provides many useful features, but do not always rely on this command to derive constraints for the subdesigns in a design hierarchy. Before you characterize a design, keep in mind the following limitations:

The `characterize` command

- Does not derive timing budgets. (It reflects the current state of the design.)
- Ignores `clock_skew` and `max_time_borrow` attributes placed on a hierarchical boundary (generally not an issue, because these attributes are usually placed on clocks and cells).

With no options, the `characterize` command replaces a subdesign's port signal information (clocks, port drive, input and output delays, maximum and minimum path delays, and port load) with information derived from the parent design.

The `characterize` command recognizes when the top-level design has back-annotated information (load, resistance, or delay) and to move this data down to the subdesign in preparation for subsequent optimization.

Propagating Constraints up the Hierarchy

If you have hierarchical designs and compile the subdesigns, then move up to the higher-level blocks (bottom-up compilation), you can propagate clocks, timing exceptions, and disabled timing arcs from lower-level .ddc files to the current design, using the `propagate_constraints` command.

Methodology for Propagating Constraints Upward

Use the `propagate_constraints` command to propagate constraints from lower levels of hierarchy to the current design. If you do not use the command, you can propagate constraints from a higher-level design to the `current_instance`, but you cannot propagate constraints set on a lower-level block to the higher-level blocks in which it is instantiated.

Note:
Using the `propagate_constraints` command might cause memory usage to increase.

Example 7-2 shows a methodology in which constraints are propagated upward. Assume that A is the top-level and B is the lower-level design.

Example 7-2 Propagating Constraints Upward

```
current_design B
source constraints.tcl
compile
current_design A
```

```
propagate_constraints -design B
report_timing_requirements
compile
report_timing
```

To generate a report of all the constraints that were propagated up, use the `-verbose` and `-dont_apply` options and redirect the output to a file:

```
dc_shell> propagate_constraints -design name \
           -verbose -dont_apply -output report.cons
```

Use the `write_file -format ddc` command to save the .ddc file with the propagated constraints so there is no need to go through the propagation again when restarting a new dc_shell session.

Handling of Conflicts Between Designs

Conflicts between the lower-level and top-level designs causes the following:

- Clock name conflict

  The lower-level clock has the same name as the clock of the current design (or another block).

  The clock is not propagated. A warning is issued.

- Clock source conflict

  A clock source of a lower-level block is already defined as a clock source of a higher-level block.

  The lower-level clock is not propagated. A warning is issued.

- Exceptions from or to an unpropagated clock

  This can be either a virtual clock, or a clock that was not propagated from that block due to a conflict.

- Exceptions

  A lower-level exception overrides a higher-level exception that is defined on the same path.

8

Optimizing the Design

Optimization is the Design Compiler synthesis step that maps the design to an optimal combination of specific target library cells, based on the design's functional, speed, and area requirements. You use the `compile_ultra` command or the `compile` command to compile a design. Design Compiler provides options that enable you to customize and control optimization. Several of the many factors affecting the optimization outcome are discussed in this chapter. For detailed information, see the *Design Compiler Optimization Reference Manual*.

To learn about Design Compiler optimization processes and techniques, see

- The Optimization Process
- Selecting and Using a Compile Strategy
- Resolving Multiple Instances of a Design Reference
- Preserving Subdesigns
- Understanding the Compile Cost Function
- Performing Design Exploration
- Performing Design Implementation
- Specifying Target Library Subsets
- Specifying Library Subsets for Sequential Cells
- Specifying Link Library Subsets

The Optimization Process

Design Compiler performs the following levels of optimization:

- Architectural Optimization
- Logic-Level Optimization
- Gate-Level Optimization

Architectural Optimization

Architectural optimization works on the HDL description. It includes such high-level synthesis tasks as

- Sharing common subexpressions
- Sharing resources
- Selecting DesignWare implementations
- Reordering operators
- Identifying arithmetic expressions for data-path synthesis (DC Ultra only).

Except for DesignWare implementations, these high-level synthesis tasks occur only during the optimization of an unmapped design. DesignWare selection can recur after gate-level mapping.

High-level synthesis tasks are based on your constraints and your HDL coding style. After high-level optimization, circuit function is represented by GTECH library parts, that is, by a generic, technology-independent netlist.

For more information about how your coding style affects architectural optimization, see Preparing Design Files for Synthesis.

Logic-Level Optimization

Logic-level optimization works on the GTECH netlist. It consists of the following two processes:

- Structuring

  This process adds intermediate variables and logic structure to a design, which can result in reduced design area. Structuring is constraint based. It is best applied to noncritical timing paths.

  During structuring, Design Compiler searches for subfunctions that can be factored out and evaluates these factors, based on the size of the factor and the number of times the factor appears in the design. Design Compiler turns the subfunctions that most reduce the logic into intermediate variables and factors them out of the design equations.

- Flattening

  The goal of this process is to convert combinational logic paths of the design to a two-level, sum-of-products representation. Flattening is carried out independently of constraints. It is useful for speed optimization because it leads to just two levels of combinational logic.

  During flattening, Design Compiler removes all intermediate variables, and therefore all its associated logic structure, from a design.

Gate-Level Optimization

Gate-level optimization works on the generic netlist created by logic synthesis to produce a technology-specific netlist. It includes the following processes:

- Mapping

  This process uses gates (combinational and sequential) from the target technology libraries to generate a gate-level implementation of the design whose goal is to meet timing and area goals. You can use the various options of the `compile_ultra` command or the `compile` command to control the mapping algorithms used by Design Compiler.

- Delay optimization

  The process goal is to fix delay violations introduced in the mapping phase. Delay optimization does not fix design rule violations or meet area constraints.

- Design rule fixing

  The process goal is to correct design rule violations by inserting buffers or resizing existing cells. Design Compiler tries to fix these violations without affecting timing and area results, but if necessary, it does violate the optimization constraints.

- Area optimization

  The process goal is to meet area constraints after the mapping, delay optimization, and design rule fixing phases are completed. However, Design Compiler does not allow area recovery to introduce design rule or delay constraint violations as a means of meeting the area constraints.

You can change the priority of the constraints by using the `set_cost_priority` command. Also, you can disable design rule fixing by specifying the `-no_design_rule` option when you run the `compile_ultra` command or `compile` command. However, if you use this option, your synthesized design might violate design rules.

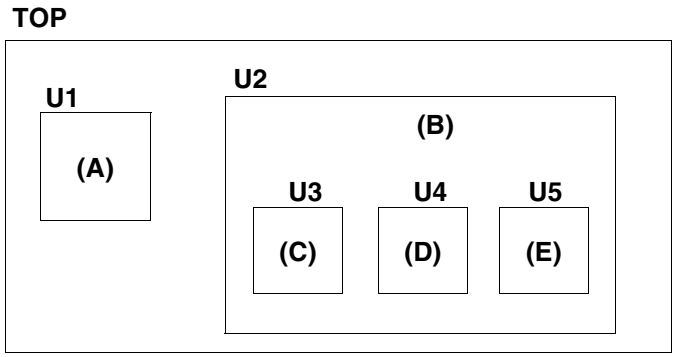
Selecting and Using a Compile Strategy

You can use various strategies to compile your hierarchical design. The basic strategies are

- Top-down compile, in which the top-level design and all its subdesigns are compiled together
- Bottom-up compile, in which the individual subdesigns are compiled separately, starting from the bottom of the hierarchy and proceeding up through the levels of the hierarchy until the top-level design is compiled
- Mixed compile, in which the top-down or bottom-up strategy, whichever is most appropriate, is applied to the individual subdesigns

In the following sections, the top-down and bottom-up compile strategies are demonstrated, using the simple design shown in Figure 8-1.

Figure 8-1 Design to Illustrate Compile Strategies



The top-level, or global, specifications for this design, given in Table 8-1, are defined by the script of Example 8-1. These specifications apply to TOP and all its subdesigns.

Table 8-1 Design Specifications for Design TOP

Specification type	Value
Operating condition	WCCOM
Wire load model	"20x20"
Clock frequency	40 MHz
Input delay time	3 ns
Output delay time	2 ns
Input drive strength	drive_of (IV)
Output load	1.5 pF

Example 8-1 Constraints File for Design TOP (defaults.con)

```
set_operating_conditions WCCOM
set_wire_load_model "20x20"
create_clock -period 25 clk
set_input_delay 3 -clock clk \
   [remove_from_collection [all_inputs] [get_ports clk]]
set_output_delay 2 -clock clk [all_outputs]
set_load 1.5 [all_outputs]
set_driving_cell -lib_cell IV [all_inputs]
set_drive 0 clk
```

Note:

To prevent buffering of the clock network, the script sets the input drive resistance of the clock port (clk) to 0 (infinite drive strength).

Top-Down Compile

You can use the top-down compile strategy for designs that are not memory or CPU limited. Furthermore, top-level designs that are memory limited can often be compiled using the top-down strategy if you first replace some of the subdesigns with interface logic model representations. Replacing a subdesign with an interface logic model can greatly reduce the memory requirements for the subdesign instantiation in the top-level design. For information about how to generate and use interface logic models, see Using Hierarchical Models.

The top-down compile strategy has these advantages:

- Provides a push-button approach
- Takes care of interblock dependencies automatically

However, the top-down compile strategy requires more memory and might result in longer runtimes for designs with over 100K gates.

To implement a top-down compile, carry out the following steps:

Note:
If your top-level design contains one or more interface logic models, use the compile flow described in Using Hierarchical Models.

1. Read in the entire design.
2. Apply attributes and constraints to the top level.

   Attributes and constraints implement the design specification. For information about attributes, see Working With Attributes. For information about constraints, see Defining the Design Environment and Defining Design Constraints.

   Note:
   You can assign local attributes and constraints to subdesigns, provided that those attributes and constraints are defined with respect to the top-level design.
3. Compile the design.

A top-down compile script for the TOP design is shown in Example 8-2.The script contains comments that identify each of the steps. The constraints are applied by including the constraint file (defaults.con) shown in Example 8-1.

Example 8-2 Top-Down Compile Script

```
# read in the entire design
read_verilog E.v
read_verilog D.v
read_verilog C.v
read_verilog B.v
read_verilog A.v
read_verilog TOP.v
current_design TOP
link

# apply constraints and attributes
source defaults.con

# compile the design
compile
```

Bottom-Up Compile

Use the bottom-up compile strategy for medium-size and large designs.

Note:
The bottom-up compile strategy is also known as the compile-characterize-write_script-recompile method.

The bottom-up compile strategy provides these advantages:

- Compiles large designs by using the divide-and-conquer approach
- Requires less memory than top-down compile
- Allows time budgeting

The bottom-up compile strategy requires

- Iterating until the interfaces are stable
- Manual revision control

The bottom-up compile strategy compiles the subdesigns separately and then incorporates them in the top-level design. The top-level constraints are applied, and the design is checked for violations. Although it is possible that no violations are present, this outcome is unlikely because the interface settings between subdesigns usually are not sufficiently accurate at the start.

To improve the accuracy of the interblock constraints, you read in the top-level design and all compiled subdesigns and apply the `characterize` command to the individual cell instances of the subdesigns. Based on the more realistic environment provided by the compiled subdesigns, the `characterize` command captures environment and timing information for each cell instance and then replaces the existing attributes and constraints of each cell's referenced subdesign with the new values.

Using the improved interblock constraint, you recompile the characterized subdesigns and again check the top-level design for constraint violations. You should see improved results, but you might need to iterate the entire process several times to remove all significant violations.

The bottom-up compile strategy requires these steps:

1. Develop both a default constraint file and subdesign-specific constraint files.

   The default constraint file includes global constraints, such as the clock information and the drive and load estimates. The subdesign-specific constraint files reflect the time budget allocated to the subblocks.

2. Compile the subdesigns independently.

3. Read in the top-level design and any compiled subdesigns not already in memory.
4. Set the current design to the top-level design, link the design, and apply the top-level constraints.

   If the design meets its constraints, you are finished. Otherwise, continue with the following steps.
5. Apply the `characterize` command to the cell instance with the worst violations.
6. Use `write_script` to save the characterized information for the cell.

   You use this script to re-create the new attribute values when you are recompiling the cell's referenced subdesign.
7. Use `remove_design -all` to remove all designs from memory.
8. Read in the RTL design of the previously characterized cell.

   Recompiling the RTL design instead of the cell's mapped design usually leads to better optimization.
9. Set `current_design` to the characterized cell's subdesign and recompile, using the saved script of characterization data.
10. Read in all other compiled subdesigns.
11. Link the current subdesign.
12. Choose another subdesign, and repeat steps 3 through 9 until you have recompiled all subdesigns, using their actual environments.

When applying the bottom-up compile strategy, consider the following:

- The `read_file` command runs most quickly with the .ddc format. If you will not be modifying your RTL code after the first time you read (or elaborate) it, save the unmapped design to a .ddc file. This will save time when you reread the design.
- The `compile` command affects all subdesigns of the current design. If you want to optimize only the current design, you can remove or not include its subdesigns in your database, or you can place the `dont_touch` attribute on the subdesigns (by using the `set_dont_touch` command).
- The subdesign constraints are not preserved after you perform a top-level compile. To ensure that you are using the correct constraints, always reapply the subdesign constraints before compiling or analyzing a subdesign.
- By default, compile modifies the original copy in the current design. This could be a problem when the same design is referenced from multiple modules, which are compiled separately in sequence. For example, the first compile could change the interface or the functionality of the design by boundary optimization. When this design is referenced from another module in the subsequent compile, the modified design is uniquified and used.

You can set the `compile_keep_original_for_external_references` variable to true, which enables compile to keep the original design when there is an external reference to the design. When the variable is set to `true`, the original design and its subdesigns are copied and preserved (before doing any modifications during compile) if there is an external reference to this design.

Typically, you require this variable only when you are doing a bottom-up compile without setting a `dont_touch` attribute on all the subdesigns, especially those with boundary optimizations turned on. If there is a `dont_touch` attribute on any of the instances of the design or in the design, this variable has no effect.

A bottom-up compile script for the TOP design is shown in Example 8-3. The script contains comments that identify each of the steps in the bottom-up compile strategy. In the script it is assumed that block constraint files exist for each of the subblocks (subdesigns) in design TOP. The compile script also uses the default constraint file (defaults.con) shown in Example 8-1.

Note:

This script shows only one pass through the bottom-up compile procedure. If the design requires further compilations, you repeat the procedure from the point where the top-level design, TOP.v, is read in.

Example 8-3 Bottom-Up Compile Script

```
set all_blocks {E D C B A}

# compile each subblock independently
foreach block $all_blocks {
   # read in block
   set block_source "$block.v"
   read_file -format verilog $block_source
   current_design $block
   link
   # apply global attributes and constraints
   source defaults.con
   # apply block attributes and constraints
   set block_script "$block.con"
   source $block_script
   # compile the block
   compile
}

# read in entire compiled design
read_file -format verilog TOP.v
current_design TOP
link
write -hierarchy -format ddc -output first_pass.ddc

# apply top-level constraints
source defaults.con
source top_level.con
```

```
# check for violations
report_constraint

# characterize all instances in the design
set all_instances {U1 U2 U2/U3 U2/U4 U2/U5}
characterize -constraint $all_instances

# save characterize information
foreach block $all_blocks {
   current_design $block
   set char_block_script "$block.wscr"
   write_script > $char_block_script
}

# recompile each block
foreach block $all_blocks {

   # clear memory
   remove_design -all

   # read in previously characterized subblock
   set block_source "$block.v"
   read_file -format verilog $block_source

   # recompile subblock
   current_design $block
   link
   # apply global attributes and constraints
   source defaults.con
   # apply characterization constraints
   set char_block_script "$block.wscr"
   source $char_block_script
   # apply block attributes and constraints
   set block_script "$block.con"
   source $block_script
   # recompile the block
   compile
}
```

Note:

When performing a bottom-up compile, if the top-level design contains glue logic as well as the subblocks (subdesigns), you must also compile the top-level design. In this case, to prevent Design Compiler from recompiling the subblocks, you first apply the `set_dont_touch` command to each subdesign.

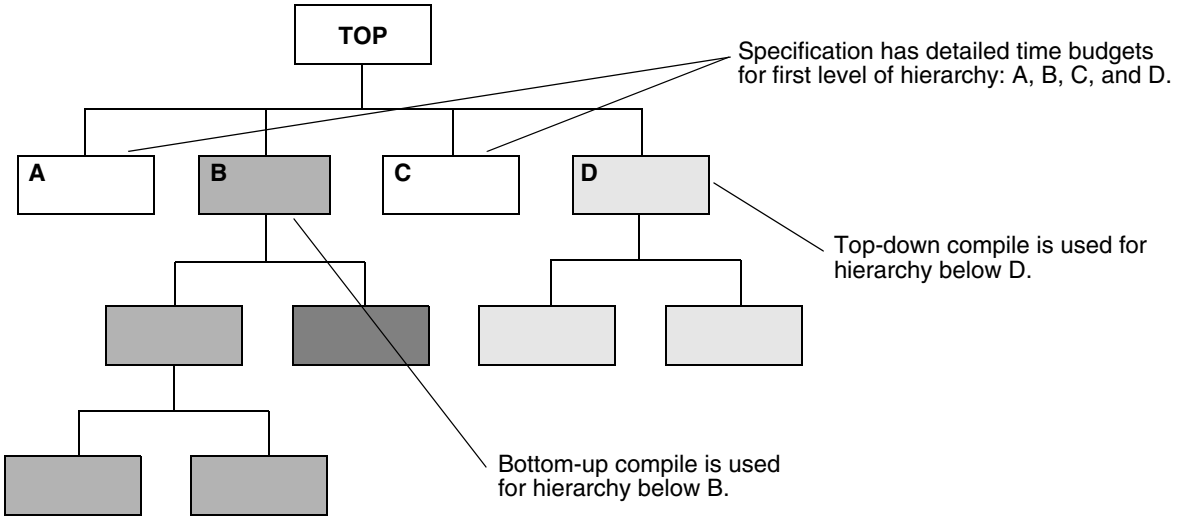
Mixed Compile Strategy

You can take advantage of the benefits of both the top-down and the bottom-up compile strategies by using both.

- Use the top-down compile strategy for small hierarchies of blocks.
- Use the bottom-up compile strategy to tie small hierarchies together into larger blocks.

Figure 8-2 shows an example of the mixed compilation strategy.

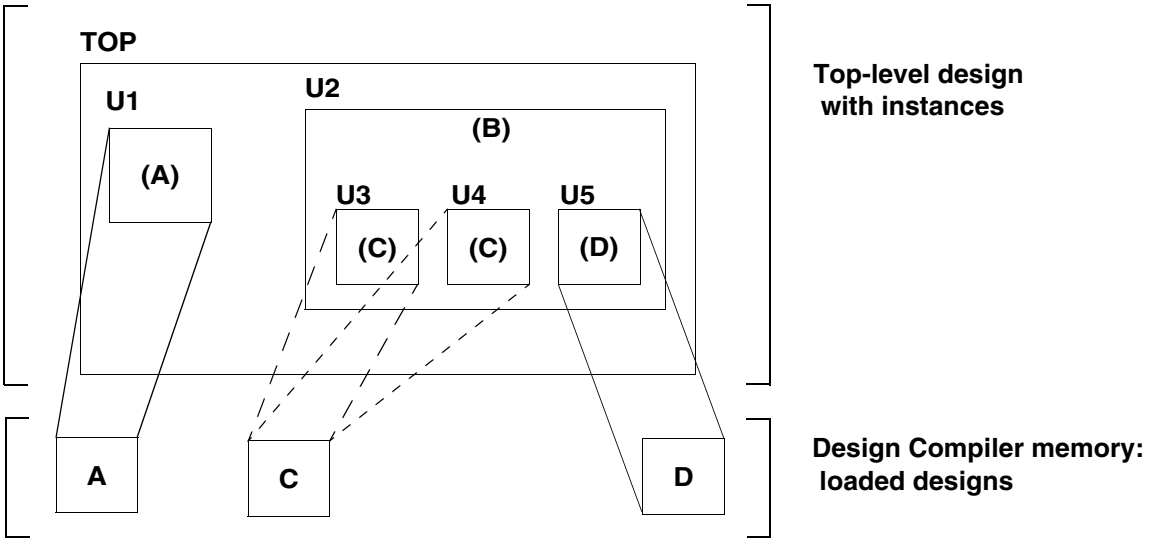
Figure 8-2 Mixing Compilation Strategies



Resolving Multiple Instances of a Design Reference

In a hierarchical design, subdesigns are often referenced by more than one cell instance, that is, multiple references of the design can occur. For example, Figure 8-3 shows the design TOP, in which design C is referenced twice (U2/U3 and U2/U4).

Figure 8-3 Multiple Instances of a Design Reference



To report information messages related to multiply-instantiated designs, use the `check_design -multiple_designs` command. The command lists all multiply instantiated designs along with instance names and associated attributes (`dont_touch`, `black_box`, and `ungroup`). The following methods are available for handling designs with multiple instances:

- The uniquify method

  In earlier releases, you had manually to run the `uniquify` command to create a uniquely named copy of the design for each instance. However, beginning with version V-2004.06, the tool automatically uniquifies designs as part of the compile process.

  Note that you can still manually force the tool to uniquify designs before compile by running the `uniquify` command, but this step contributes to longer runtimes because the tool automatically “re-uniquifies” the designs when compile the design. You cannot turn off the uniquify process.

- The compile-once-don’t-touch method

  This method uses the `set_dont_touch` command to preserve the compiled subdesign while the remaining designs are compiled.

- The ungroup method

  This method uses the `ungroup` command to remove the hierarchy.

Uniquify Method

The uniquify process copies and renames any multiply referenced design so that each instance references a unique design. The process removes the original design from memory after it creates the new, unique designs. The original design and any collections that contain it or its objects are no longer accessible.

In earlier releases, you had manually to run the `uniquify` command to create a uniquely named copy of the design for each instance. However, beginning with version V-2004.06, the tool automatically uniquifies designs as part of the compile process. The uniquification process can resolve multiple references throughout the hierarchy the current design (except those having a `dont_touch` attribute). After this process finishes, the tool can optimize each design copy based on the unique environment of its cell instance.

You can also create unique copies for specific references by using the `-reference` option of the `uniquify` command, or you can specify specific cells by using the `-cell` option. Design Compiler makes unique copies for cells specified with the `-reference` or the `-cells` option, even if they have a `dont_touch` attribute.

The `uniquify` command accepts instance objects—that is, cells at a lower level of hierarchy. When you use the `-cell` option with an instance object, the complete path to the instance is uniquified. For example, the following command uniquifies both instances mid1 and mid1/bot1, assuming that mid1 is not unique:

```
dc_shell> uniquify -cell mid1/bot1
```

Design Compiler uses the naming convention specified in the `uniquify_naming_style` variable to generate the name for each copy of the subdesign. The default naming convention is `%s_%d`, which is defined as follows:

`%s`

The original name of the subdesign, or the name specified in the `-base_name` option.

`%d`

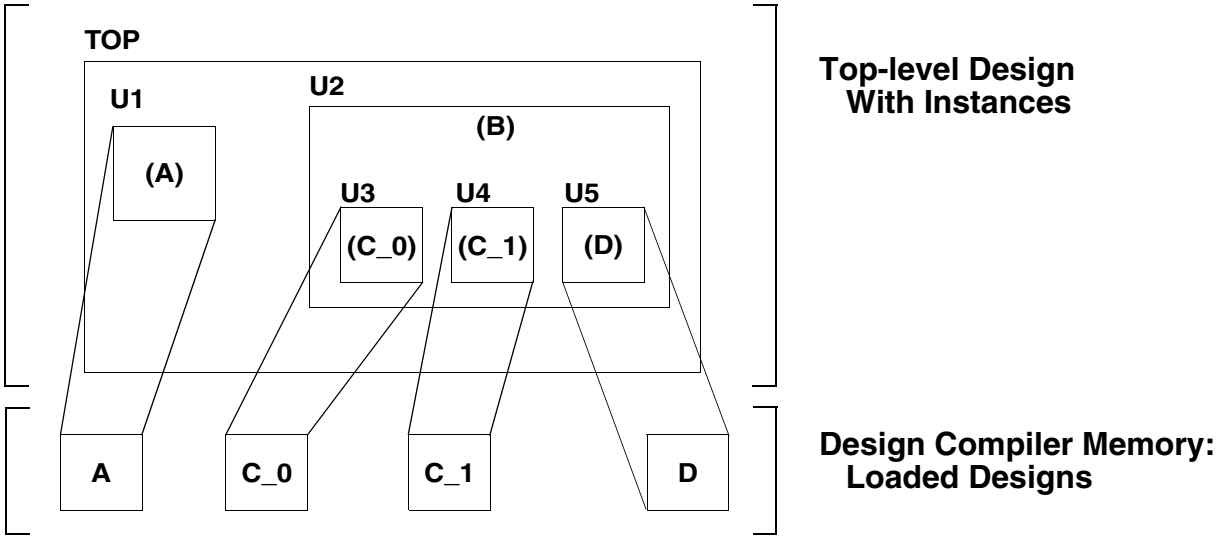
The smallest integer value that forms a unique subdesign name.

The following command sequence resolves the multiple instances of design C in design TOP shown in Figure 8-3; it uses the automatic uniquify method to create new designs C_0 and C_1 by copying design C and then replaces design C with the two copies in memory.

```
dc_shell> current_design top
dc_shell> compile
```

Figure 8-4 shows the result of running this command sequence.

Figure 8-4 Uniquify Results



Compared with the compile-once-don't-touch method, the uniquify method has the following characteristics:

- Requires more memory
- Takes longer to compile

Compile-Once-Don't-Touch Method

If the environments around the instances of a multiply referenced design are sufficiently similar, use the compile-once-don't-touch method. In this method, you compile the design, using the environment of one of its instances, and then you use the `set_dont_touch` command to preserve the subdesign during the remaining optimization. For details about the `set_dont_touch` command, see Preserving Subdesigns.

To use the compile-once-don't-touch method to resolve multiple instances, follow these steps:

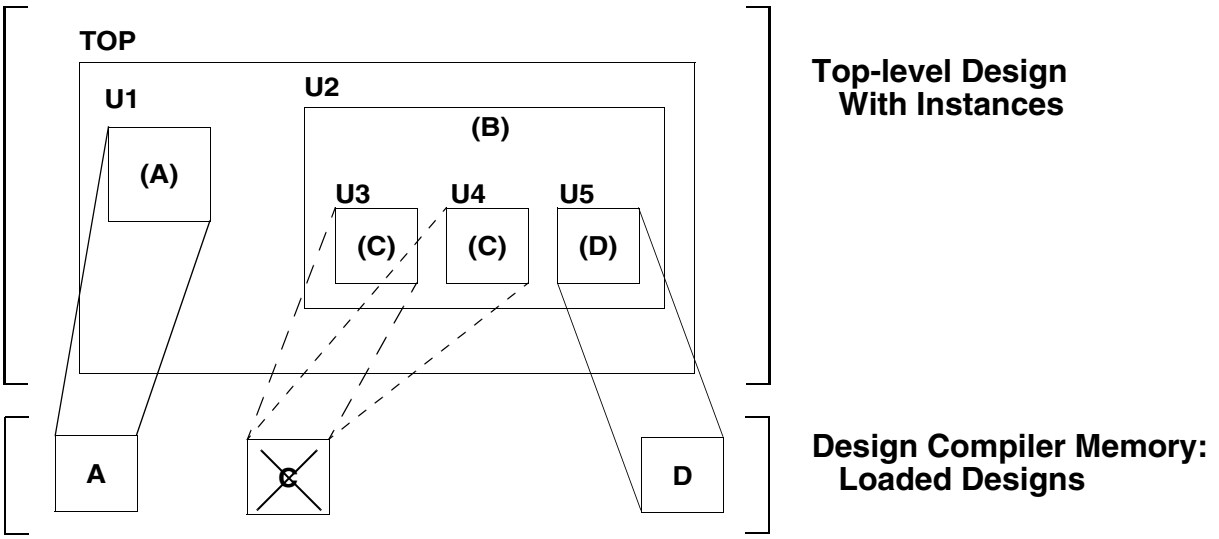
1. Characterize the subdesign's instance that has the worst-case environment.
2. Compile the referenced subdesign.
3. Use the `set_dont_touch` command to set the `dont_touch` attribute on all instances that reference the compiled subdesign.
4. Compile the entire design.

For example, the following command sequence resolves the multiple instances of design C in design TOP by using the compile-once-don't-touch method (assuming U2/U3 has the worst-case environment). In this case, no copies of the original subdesign are loaded into memory.

```
dc_shell> current_design top
dc_shell> characterize U2/U3
dc_shell> current_design C
dc_shell> compile
dc_shell> current_design top
dc_shell> set_dont_touch {U2/U3 U2/U4}
dc_shell> compile
```

Figure 8-5 shows the result of running this command sequence. The X drawn over the C design, which has already been compiled, indicates that the `dont_touch` attribute has been set. This design is not modified when the top-level design is compiled.

Figure 8-5 Compile-Once-Don't-Touch Results



The compile-once-don't-touch method has the following advantages:

- Compiles the reference design one time
- Requires less memory than the uniquify method
- Takes less time to compile than the uniquify method

The principal disadvantage of the compile-once-don't-touch method is that the characterization might not apply well to all instances. Another disadvantage is that you cannot ungroup objects that have the `dont_touch` attribute.

Ungroup Method

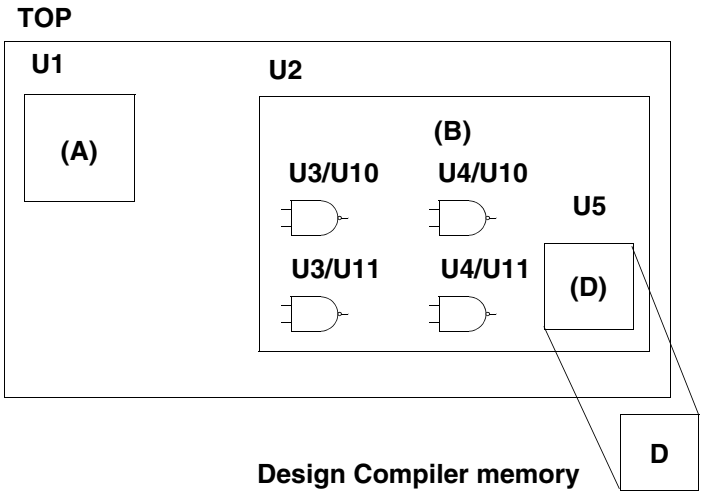
The ungroup method has the same effect as the uniquify method (it makes unique copies of the design), but in addition, it removes levels of hierarchy. This method uses the `ungroup` command to produce a flattened netlist. For details about the `ungroup` command, see Removing Levels of Hierarchy.

After ungrouping the instances of a subdesign, you can recompile the top-level design. For example, the following command sequence uses the ungroup method to resolve the multiple instances of design C in design TOP:

```
dc_shell> current_design B
dc_shell> ungroup {U3 U4}
dc_shell> current_design top
dc_shell> compile
```

Figure 8-6 shows the result of running this command sequence.

Figure 8-6 Ungroup Results



The ungroup method has the following characteristics:

- Requires more memory and takes longer to compile than the compile-once-don't-touch method
- Provides the best synthesis results

The obvious drawback in using the ungroup method is that it removes the user-defined design hierarchy.

Preserving Subdesigns

The `set_dont_touch` command preserves a subdesign during optimization. It places the `dont_touch` attribute on cells, nets, references, and designs in the current design to prevent these objects from being modified or replaced during optimization.

Use the `set_dont_touch` command on subdesigns that you do not want optimized with the rest of the design hierarchy. The `dont_touch` attribute does not prevent or disable timing through the design.

Note:

Any interface logic model present in your design is automatically marked as `dont_touch`. Also, the cells of an interface logic model are marked as `dont_touch`. For information about interface logic models, see Using Hierarchical Models.

Preserving Cells, References, and Designs

When you use the `set_dont_touch` command on cells, references, and designs in the current design, consider the following:

- Setting the `dont_touch` attribute on a hierarchical cell sets an implicit `dont_touch` on all cells below that cell.
- Setting the `dont_touch` attribute on a library cell sets an implicit `dont_touch` on all instances of that cell.
- Setting the `dont_touch` attribute on a reference sets an implicit `dont_touch` on all cells using that reference during subsequent optimizations of the design.
- Setting the `dont_touch` attribute on a design has an effect only when the design is instantiated within another design as a level of hierarchy. In this case, the `dont_touch` attribute on the design implies that all cells under that level of hierarchy are subject to the `dont_touch` attribute. Setting `dont_touch` on the top-level design has no effect because the top-level design is not instantiated within any other design.
- You cannot manually or automatically ungroup objects marked as `dont_touch`. That is, the `ungroup` command and the compile `-ungroup_all` and `-auto_ungroup` options have no effect on objects marked as `dont_touch`.

Preserving Nets

When you use the set_dont_touch command on nets in the current design, consider the following:

- By default, if you set a dont_touch attribute on a net, Design Compiler sets an implicit dont_touch only on mapped combinational cells that are connected to that net. If the net is connected only to unmapped cells, optimization might remove the net.

  The dont_touch attribute is ignored on nets that have unmapped cells on them. During compilation, Design Compiler issues warnings for dont_touch nets connected to unmapped cells (generic logic).

- You can preserve specific nets throughout the compilation process by setting the enable_keep_signal_dt_net variable to true before using the set_dont_touch command on the net. When you do this, the dont_touch attribute on the net sets an implicit size_only attribute on logic connected to that net even if logic connected to it is unmapped and not combinational.

  Warning:

  Preserving specific nets throughout compilation can cause quality of results (QoR) degradation. If you preserve a net that is in the critical path, the QoR degradation can be severe. This net preservation functionality is intended to facilitate verification. It should not be used when you are trying to achieve final QoR goals.

  Design Compiler issues warnings during compilation to indicate nets that are user-preserved.

  In the following example, the enable_keep_signal_dt_net variable is enabled to preserve the my_net net throughout compilation.

  1. Make sure the net to be preserved is present in the design and then use the enable_keep_signal_dt_net variable and the set_dont_touch command to preserve the net:

     ```
     dc_shell> get_nets *my_net*
     dc_shell> set enable_keep_signal_dt_net true
     dc_shell> set_dont_touch [get_nets *my_net*] true
     ```

  2. Run the compile_ultra command:

     ```
     dc_shell> compile_ultra
     ```

     Check for OPT-154 warning messages at the beginning of compilation to indicate that the net is preserved:

     ```
     Warning: Preserving net 'my_net'. Expect QoR impact. (OPT-154)
     ```

  3. Make sure the net is preserved after compilation:

     ```
     dc_shell> get_nets *my_net*
     ```

Removing a dont_touch Setting

To remove the `dont_touch` attribute, use the `remove_attribute` command or the `set_dont_touch` command set to `false`.

Understanding the Compile Cost Function

The compile cost function consists of design rule costs and optimization costs. By default, Design Compiler prioritizes costs in the following order:

1. Design rule costs
    a. Connection class
    b. Multiple port nets
    c. Maximum transition time
    d. Maximum fanout
    e. Maximum capacitance
    f. Cell degradation
2. Optimization costs
    a. Maximum delay
    b. Minimum delay
    c. Maximum power
    d. Maximum area

The compile cost function considers only those components that are active in your design. Design Compiler evaluates each cost function component independently, in order of importance.

When evaluating cost function components, Design Compiler considers only violators (positive difference between actual value and constraint) and works to reduce the cost function to zero.

The goal of Design Compiler is to meet all constraints. However, by default, it gives precedence to design rule constraints because design rule constraints are functional requirements for designs. Using the default priority, Design Compiler fixes design rule violations even at the expense of violating your delay or area constraints.

You can change the priority of the maximum design rule costs and the delay costs by using the `set_cost_priority` command to specify the ordering. You must run the `set_cost_priority` command before running the `compile` command.

You can disable evaluation of the design rule cost function by using the `-no_design_rule` option when running the `compile_ultra` command or `compile` command.

You can disable evaluation of the optimization cost function by using the `-only_design_rule` option when running the `compile_ultra command or compile` command.

For more information about the compile cost function, see the *Design Compiler Optimization Reference Manual*.

Performing Design Exploration

In design exploration, you use the default synthesis algorithm to gauge the design performance against your goals. To invoke the default synthesis algorithm, use the `compile` command with no options:

```
dc_shell> compile
```

The default compile uses the `-map_effort medium` option of the `compile` command. The default area effort of the area recovery phase of the compile is the specified value of the `map_effort` option. You can change the area effort by using the `-area_effort` option.

If the performance violates the timing goals by more than 15 percent, you should consider whether to refine the design budget or modify the HDL code.

Performing Design Implementation

The default compile generates good results for most designs. If your design meets the optimization goals after design exploration, you are finished. If not, try the techniques described in the following sections:

- Optimizing High-Performance Designs
- Optimizing for Maximum Performance
- Optimizing for Minimum Area
- Optimizing Datapaths

Optimizing High-Performance Designs

For high-performance designs that have significantly tight timing constraints, you can invoke a single DC Ultra command, `compile_ultra`, for better quality of results (QoR). This command allows you to apply the best possible set of timing-centric variables or commands during compile for critical delay optimization as well as improvement in area QoR. Because `compile_ultra` includes all compile options and starts the entire compile process, no separate `compile` command is necessary.

By default, if the dw_foundation.sldb library is not in the synthetic library list but the DesignWare license has been successfully checked out, the dw_foundation.sldb library is automatically added to the synthetic library list. This behavior applies to the current command only. The user-specified synthetic library and link library lists are not affected.

In addition, all DesignWare hierarchies are, by default, unconditionally ungrouped in the second pass of the compile. You can prevent this ungrouping by setting the `compile_ultra_ungroup_dw` variable to `false` (the default is `true`).

To use the `compile_ultra` command, you will need a DC Ultra license and a DesignWare Foundation license.

For more information, see the *Design Compiler Optimization Reference Manual*.

Optimizing for Maximum Performance

If your design does not meet the timing constraints, you can try the following methods to improve performance:

- Creating Path Groups
- Fixing Heavily Loaded Nets
- Automatically Ungrouping Hierarchies on the Critical Path
- Performing a High-Effort Compile
- Performing a High-Effort Incremental Compile

Creating Path Groups

By default, Design Compiler groups paths based on the clock controlling the endpoint (all paths not associated with a clock are in the default path group). If your design has complex clocking, complex timing requirements, or complex constraints, you can create path groups to focus Design Compiler on specific critical paths in your design.

Use the `group_path` command to create path groups. The `group_path` command allows you to

- Control the optimization of your design
- Optimize near-critical paths
- Optimize all paths

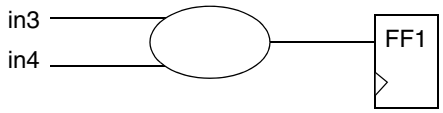
Controlling the Optimization of Your Design

You can control the optimization of your design by creating and prioritizing path groups, which affect only the maximum delay cost function. By default, Design Compiler works only on the worst violator in each group.

Set the path group priorities by assigning weights to each group (the default weight is 1.0). The weight can be from 0.0 to 100.0.

For example, Figure 8-7 shows a design that has multiple paths to flip-flop FF1.

Figure 8-7 Path Group Example



To indicate that the path from input in3 to FF1 is the highest-priority path, use the following command to create a high-priority path group:

```
dc_shell> group_path -name group3 -from in3 -to FF1/D -weight 2.5
```

Optimizing Near-Critical Paths

When you add a critical range to a path group, you change the maximum delay cost function from worst negative slack to critical negative slack. Design Compiler optimizes all paths within the critical range.

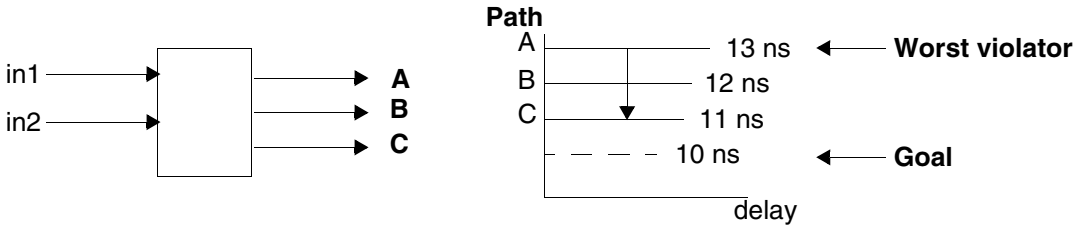
Specifying a critical range can increase runtime. To limit the runtime increase, use critical range only during the final implementation phase of the design, and use a reasonable critical range value. A guideline for the maximum critical range value is 10 percent of the clock period.

Use one of the following methods to specify the critical range:

- Use the `-critical_range` option of the `group_path` command.
- Use the `set_critical_range` command.

For example, Figure 8-8 shows a design with three outputs, A, B, and C.

Figure 8-8 Critical Range Example



Assume that the clock period is 20 ns, the maximum delay on each of these outputs is 10 ns, and the path delays are as shown. By default, Design Compiler optimizes only the worst violator (the path to output A). To optimize all paths, set the critical delay to 3.0 ns. For example,

```
create_clock -period 20 clk
set_critical_range 3.0 $current_design
set_max_delay 10 {A B C}
group_path -name group1 -to {A B C}
```

Optimizing All Paths

You can optimize all paths by creating a path group for each endpoint in the design. Creating a path group for each endpoint enables total negative slack optimization but results in long compile runtimes.

Use the following script to create a path group for each endpoint.

```
set endpoints \
      [add_to_collection [all_outputs] \
      [all_registers -data_pins]]
foreach_in_collection endpt $endpoints {
   set pin [get_object_name $endpt]
   group_path -name $pin -to $pin
}
```

Fixing Heavily Loaded Nets

Heavily loaded nets often become critical paths. To reduce the load on a net, you can use either of two approaches:

- If the large load resides in a single module and the module contains no hierarchy, fix the heavily loaded net by using the `balance_buffer` command. For example, enter

```
source constraints.con
compile_ultra
balance_buffer -from [get_pins buf1/Z]
```

  Note:

  The `balance_buffer` command provides the best results when your library uses linear delay models. If your library uses nonlinear delay models, the second approach provides better results.

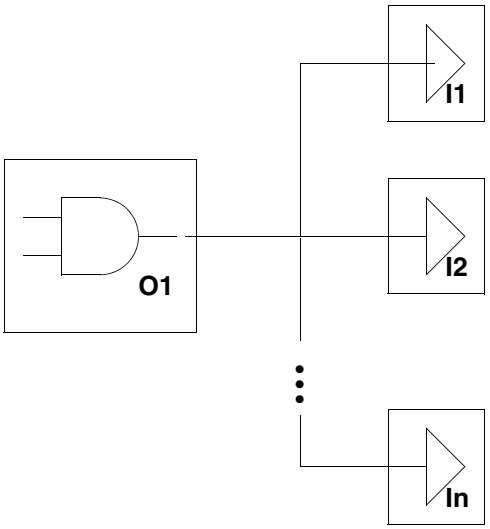
- If the large loads reside across the hierarchy from several modules, apply design rules to fix the problem. For example,

```
source constraints.con
compile_ultra
set_max_capacitance 3.0
compile_ultra -only_design_rule
```

In rare cases, hierarchical structure might disable Design Compiler from fixing design rules.

In the sample design shown in Figure 8-9, net O1 is overloaded. To reduce the load, group as many of the loads (I1 through In) as possible in one level of hierarchy by using the `group` command or by changing the HDL. Then you can apply one of the approaches.

Figure 8-9 Heavily Loaded Net



Automatically Ungrouping Hierarchies on the Critical Path

Automatically ungrouping hierarchies during compile can often improve performance. Ungrouping removes hierarchy boundaries and allows Design Compiler to optimize over a larger number of gates, generally improving timing. You use delay-based automatic ungrouping to ungroup hierarchies along the critical path.

Design Compiler provides the following automatic ungrouping strategies:

- Area-based automatic ungrouping

  The `compile_ultra` command performs area-based automatic ungrouping before initial mapping. The tool estimates the area for unmapped hierarchies and removes small subdesigns; the goal is to improve area and timing quality of results. Because the tool performs automatic ungrouping at an early stage, it has a better optimization context. Additionally, datapath extraction is enabled across ungrouped hierarchies. These factors improve the timing and area quality of results.

- Delay-based automatic ungrouping

  During delay optimization, the `compile_ultra` command performs delay-based automatic ungrouping. It ungroups hierarchies along the critical path and is used essentially for timing optimization.

- When you use the `-spg` option with the `compile_ultra` command, Design Compiler Graphical ungroups additional hierarchies to improve QoR.

For information about controlling automatic ungrouping, see the *Design Compiler Optimization Reference Manual*.

Performing a High-Effort Compile

The optimization result depends on the starting point. Occasionally, the starting point generated by the default compile results in a local minimum solution, and Design Compiler quits before generating an optimal design. A high-effort compile might solve this problem.

The high-effort compile uses the `-map_effort high` option of the `compile` command on the initial compile (on the HDL description of the design).

```
dc_shell> elaborate my_design
dc_shell> compile -map_effort high
```

A high-effort compile pushes Design Compiler to the extreme to achieve the design goal. A high-effort compile invokes the critical path resynthesis strategy to restructure and remap the logic on and around the critical path.

This compile strategy is CPU intensive, especially when you do not use the incremental compile option, with the result that the entire design is compiled using a high map effort.

Performing a High-Effort Incremental Compile

You can often improve compile performance of a high-effort compile by using the incremental compile option. Also, if none of the previous strategies results in a design that meets your optimization goals, a high-effort incremental compile might produce the desired result.

An incremental compile (`-incremental_mapping` compile option) allows you to incrementally improve your design by experimenting with different approaches. An incremental compile performs only gate-level optimization and does not perform logic-level optimization. The resulting design's performance is the same or better than the original design's.

This technique can still require large amounts of CPU time, but it is the most successful method for reducing the worst negative slack to zero. To reduce runtime, you can place a `dont_touch` attribute on all blocks that already meet timing constraints.

```
dc_shell> dont_touch noncritical_blocks
dc_shell> compile -map_effort high -incremental_mapping
```

This incremental approach works best for a logic library that has many variations of each logic cell.

Incremental compile supports adaptive retiming, that is, `compile_ultra -incremental -retime`.

Optimizing for Minimum Area

If your design has timing constraints, these constraints always take precedence over area requirements. For area-critical designs, do not over constrain your design.

If your design does not meet the area constraints, you can try the following methods to reduce the area:

- Disable total negative slack optimization
- Optimize across hierarchical boundaries

Disabling Total Negative Slack Optimization

By default, Design Compiler prioritizes total negative slack over meeting area constraints. This means Design Compiler performs area optimization only on those paths that have positive slack.

To change the default priorities (prioritize area over total negative slack), use the `-ignore_tns` option when setting the area constraints.

```
dc_shell> set_max_area -ignore_tns max_area
```

Optimizing Across Hierarchical Boundaries

Design Compiler respects levels of hierarchy and port functionality (except when automatic ungrouping of small hierarchies is enabled). To fine-tune the area, you can leave the hierarchy intact and enable boundary optimization. For greater area reduction, you might have to remove hierarchical boundaries.

The `compile_ultra` command performs the following types of boundary optimization by default:

- Propagation of constants across the hierarchy
- Propagation of equal and opposite information across the hierarchy
- Propagation of unconnected port information across the hierarchy
- Pushing of inverters across the hierarchy (phase inversion)

For information about controlling boundary optimization, see the *Design Compiler Optimization Reference Manual*.

Hierarchy Removal

Removing levels of hierarchy by ungrouping gives Design Compiler more freedom to share common terms across the entire design. You can ungroup specific hierarchies before optimization by using the `set_ungroup` command to designate which cells you want ungrouped. You can also use the automatic ungrouping capability to ungroup small hierarchies during optimization. In this case, you do not specify the hierarchies to be ungrouped.

For details about ungrouping hierarchies, see Removing Levels of Hierarchy and the *Design Compiler Optimization Reference Manual*.

Optimizing Datapaths

Datapath design is commonly used in applications that contain extensive data manipulation, such as 3-D, multimedia, and digital signal processing (DSP). Datapath extraction transforms arithmetic operators, such as addition, subtraction, and multiplication, into datapath blocks to be implemented by a datapath generator. This transformation improves the QoR by utilizing the carry-save arithmetic technique.

DC Ultra uses the datapath generator to build arithmetic components for optimal QoR considering the bit-level timing context during optimization. To use this improved datapath generator with the `compile` command, the dw_foundation.sldb library must be listed in the synthetic library. If necessary, use the `set synthetic_library dw_foundation.sldb` command. A DesignWare license is required.

DC Ultra enables datapath extraction and explores various datapath and resource-sharing options during compile. DC Ultra datapath optimization provides the following benefits:

- Shares datapath operators
- Extracts the datapath
- Explores better solutions that might involve a different resource-sharing configuration
- Allows the tool to make better tradeoffs between resource sharing and datapath optimization

DC Ultra datapath optimization is enabled by default when you use `compile_ultra`.

For more information about datapath synthesis, see the *Design Compiler Optimization Reference Manual*.

Specifying Target Library Subsets

Design Compiler allows you to restrict optimization of a design block using a subset of the target library. Normally, optimization can select any library cell from the target library. However, you can restrict the library cells to be used in a particular block or filter the target library cells on a block-by-block basis with the `set_target_library_subset` command. For example, you can omit specific double-height cells in some blocks even though they are mixed with other needed library cells in the target library. The subset restriction only applies to new cells that are created or mapped during optimization. It does not affect cells that are already mapped unless Design Compiler encounters a reason to modify the cells during optimization.

When you use the `set_target_library_subset` command, the subset applies to the specified blocks and their subblocks. Applying the `set_target_library_subset` command to a hierarchical cell or to the top-level design enforces the library restriction on all lower cells in the hierarchy, except for those cells that have a different library subset constraint explicitly set on them. A subset at a lower level overrides any subset specified at a higher level. Design Compiler does not incrementally refine the upper subset.

To restrict a list of blocks or top-level cells to a specific target library subset during optimization, use the `-object_list` option. The cells are instances of hierarchical designs, and the subset restriction applies to these cells and their child instances. The target library subset cannot be specified on leaf-level cells. If you do not specify the `-object_list` option, Design Compiler sets the target library subset on the current design.

To specify a list of libraries that are available to optimize the identified design instances, use the *`library_list`* argument. These libraries must also be specified by the `target_library` variable. If you do not specify the *`library_list`* argument, all the libraries listed in the `target_library` variable can be used.

To specify library cells that cannot be used for optimization even if they are inside the libraries specified by the *`library_list`* argument, use the `-dont_use` option. You can specify the library cells by name or by collection in a space-separated list. You can also use a wildcard (*).

Use the `-only_here` option to specify library cells that can be used for optimization within the block but cannot be used in other blocks (unless those blocks also list the library cell in an `-only_here` option). This essentially applies a `-dont_use` option on these cells for all other target library subsets in the design, including the target library subset at the top level, unless the subsets also contain an `-only_here` option for these cells. You can specify the library cells by name or by collection in a space-separated list. You can also use a wildcard (*).

If the `-dont_use` cell list and the `-only_here` cell list include the same cell name after wildcard expansion, the `-only_here` cell list takes precedence, and Design Compiler reports this with an information message.

If you specify a library name with the -dont_use and -only_here options, Design Compiler ignores the specified library and reports an information message. For example, if you run the following command, Design Compiler issues a message saying that library name lib1 is irrelevant and is ignored:

```
set_target_library_subset -top -dont_use {AN2 */OR* lib1/NOR2}
```

In this case, Design Compiler restricts the use of all library cells named NOR2.

You cannot override a dont_use attribute if it is set in a library. If the dont_use attribute is set on a target library cell, it cannot be used even if listed in the -only_here cell list. Design Compiler issues a warning if you apply the -only_here option to a target library cell that has a dont_use attribute set on it. If you want to make target library cells selectively available for use with the set_target_library_subset command, you must first remove the dont_use attribute from the library with the remove_attribute command.

You cannot ungroup a subdesign if the set_target_library_subset command is set on it. Similarly, the tool's auto ungrouping capability cannot ungroup the subdesign.

Removing Target Library Subsets

To remove a target library subset constraint from the design or a design instance, use the remove_target_library_subset command with the appropriate library list and cell list, or use the reset_design command.

Checking Target Library Subsets

To check for errors and conflicts introduced by target library subsets, use either the check_mv_design -target_library_subset or check_target_library_subset command. The commands check for the following conditions:

- Conflicts between target library subsets and the global target_library variable
- Conflicts between operating condition and target library subset
- Conflicts between the library cell of a mapped cell and target library subset

The check_mv_design -target_library_subset command searches the design for any cells that do not follow the rules for the subset. If encountered, Design Compiler reports them as warnings. They are not reported as errors because they might have already existed in the design before the subset was specified. Using the check_mv_design -target_library_subset command can be helpful to identify issues if you run the command before optimization and then run the command again after optimization to see if you get the same results.

Reporting Target Library Subsets

To find out which target library subsets have been defined both for the hierarchical cells and at the top level, use the `report_target_library_subset` command with the appropriate library cell list.

Reports that are generated by reporting commands, such as `report_cell` and `report_timing`, include a `td` attribute, indicating that a cell specified with the `-dont_use` option or the `-only_here` option is used somewhere in the design.

Examples

The following example shows how to set the target library for the design to the full set of libraries “lib1.db lib2.db” but restrict the library cells used in block u1 to the cells in library lib2.db:

```
dc_shell-topo> set target_library "lib1.db lib2.db"
dc_shell-topo> set_target_library_subset "lib2.db" -object_list \
 [get_cells u1]
```

The following example restricts the library cells avoid1 and avoid2 so they will not be used in blocks u1 and u2:

```
dc_shell-topo> set_target_library_subset -object_list "u1 u2" \
  -dont_use "avoid1 avoid2"
```

The following example restricts library cell SPECIAL so it can only be used in blocks u1 and u2:

```
dc_shell-topo> set_target_library_subset -object_list "u1 u2" \
 -only_here "SPECIAL"
```

The following example achieves the same effect as the previous example. The second `set_target_library_subset` command gives no restrictions and opens up the entire target library in blocks u1 and u2, including SPECIAL.

```
dc_shell-topo> set_target_library_subset -top -dont_use "SPECIAL"
dc_shell-topo> set_target_library_subset -object_list "u1 u2"
```

Specifying Library Subsets for Sequential Cells

Design Compiler in topographical mode allows you to restrict the mapping and optimization of sequential cells in a design to a user-specified subset of library cells in a target library. You define the subset of library cells by using the `define_libcell_subset` command. The command specifies a list of library cells and defines them as a family if they meet the following criteria: The library cells cannot belong to another family; they must have the same functional identification; they must be sequential library cells. Multiple subsets are allowed.

After the library cells are grouped into a subset family, they are not available for general mapping during the compile run. Design Compiler topographical uses the library cells specified in the subset exclusively to map the sequential cells that you specify with the `set_libcell_subset` command.

Use the `object_list` argument with the `set_libcell_subset` command to specify the list of sequential cells to be optimized. The cells must be either unmapped or instantiated as one of the library cells in the library cell subset. Use the `-family_name` option to specify the library subset of cells defined by the `define_libcell_subset` command that you are using during optimization.

When you specify the `define_libcell_subset` and `set_libcell_subset` commands, Design Compiler restricts all optimizations, including cell sizing and cell swapping, to the cells specified within the library cell subset. The subset restriction applies only to cells that are created, mapped, or modified during optimization. The subset does not affect any unoptimized portions of the design.

In the following example, the `define_libcell_subset` command groups logic library cells SDFLOP1 and SDFLOP2 into a family called “specialflops.” Next, the `set_libcell_subset` command sets the library cell subset for instances reg01 and reg02 to the user-defined cell family, specialflops, which consists of the cells SDFLOP1 and SDFLOP2:

```
dc_shell-topo> define_libcell_subset -libcell_list {SDFLOP1 SDFLOP2} \
                                     -family_name specialflops
dc_shell-topo> set_libcell_subset -object_list [get_cells {reg01 reg02}] \
                                  -family_name specialflops
```

Reporting Library Cell Subsets

Use the `report_libcell_subset` command to report information about the library cell subset that is specified for the sequential cells. It reports the library subset information that is explicitly user defined.

The following example reports the library cell subset family for the reg01 sequential cell:

```
dc_shell-topo> report_libcell_subset -object_list [get_cells reg01]

****************************************
Report : library cell subset
Design : chip
Version: G-2012.06
Date   : Wed Mar 21 16:16:58 2012
****************************************

Libcell Family        Libcells
--------------------------------------------------------------------------------

specialflops          sff0 sff1 sff2
--------------------------------------------------------------------------------

Object                Reference           Libcell Family
--------------------------------------------------------------------------------

reg01                 sff0                specialflops
--------------------------------------------------------------------------------

1
```

Removing Library Cell Subsets

The `remove_libcell_subset` command removes a library cell subset constraint from specified sequential cells or removes a user-defined library cell subset.

A library cell can only belong to one subset; therefore, if you need to change the subset of a library cell, you should remove the existing subset definition first by using the `remove_libcell_subset` command.

In the following example, the library subset constraint is removed from the reg01 and reg02 instances:

```
dc_shell-topo> remove_libcell_subset -object_list [get_cells {reg01 reg02}]
```

In the following example, the user-defined specialflops library cell subset is removed:

```
dc_shell-topo> remove_libcell_subset -family_name specialflops
```

Specifying Link Library Subsets

Design Compiler allows you to restrict the selection of library cells so they are chosen from a subset of the libraries specified by the `link_library` variable. Normally, library cells are selected from any library in the `link_library` variable. However, when you use the `set_link_library_subset` command, Design Compiler restricts the selection of library cells, which can resolve ambiguity among libraries with the same voltage, temperature, and process.

The command resolves ambiguity among libraries that are characterized for the same voltage, temperature, and process, but that are characterized differently for other known arbitrary parameters. For example, in a multivoltage or multicorner design, the tool would normally determine a cell's timing and power from a library in the `link_library` whose voltage, temperature, and process match the cell. If more than one library satisfies these conditions, the tool issues an "Ambiguous Libraries" warning (MV-086). You can use the `set_link_library_subset` command to identify which library is appropriate for a particular block or scenario.

The `set_link_library_subset` command does not override existing library selection rules; it augments them. The tool still matches voltage, temperature, and process within the libraries listed in the subset. The link library subset provides an additional filter on the libraries to be considered. You could, for example, use the command to make macro libraries unambiguous by specifying just the relevant macro library, without the need to list all the other libraries in the link library.

To specify a list of cells for which the link library subset will be used, use the `set_link_library_subset` command with the `-object_list` option. The cells must be instances of hierarchical designs, macros, or pads. The subset restriction applies to these cells and their child instances.

To specify a subset of libraries belonging to the `link_library` variable, use the `set_link_library_subset` command with the *`library_list`* argument. For any reference name that appears in a library of the subset, only library cells within the subset are used. For reference names that do not appear in the subset, library cells are selected from any library in the `link_library` variable, as usual.

Note:

The `set_link_library_subset` command does not work the same way as the `set_target_library_subset` command. You do not need to specify anything to make optimization work correctly with the link library subsets. Link library subsets are taken into account when Design Compiler decides which library cells have suitable characterizations. The `set_target_library_subset` command restricts the library cells to be used in a particular block or filters the target library cells on a block-by-block basis. However, the `set_target_library_subset` command does not select individual characterizations.

Reporting Link Library Subsets

To report the link library subset on the design based on the specified options, use the `report_link_library_subset` command.

The following example reports the link library subset for the top-level design and for the u1 instance:

```
dc_shell-topo> report_link_library_subset -top -object_list [get_cells u1]
```

Removing Link Library Subsets

To remove a link library subset from the design or a design instance, use the `remove_link_library_subset` or `reset_design` command.

The following example removes the link library subset from the top-level design and from the u1 instance:

```
dc_shell-topo> remove_link_library_subset -object_list [get_cells u1] -top
```

Examples

The following example has two libraries, lib1.db and lib2.db, with the same voltage, temperature, and process. Ordinarily this would result in an “Ambiguous Libraries” warning, but it does not because the ambiguity is resolved. Library values are taken from lib1.db everywhere in the design, except for the hierarchy under cell u1, which uses values from lib2.db.

```
dc_shell-topo> set_app_var link_library "* lib1.db lib2.db"
dc_shell-topo> set_link_library_subset "lib1.db" -top
dc_shell-topo> set_link_library_subset "lib2.db" -object_list [get_cells u1]
```

The following example uses two macro libraries, extracted as models by PrimeTime with different parasitics in effect.

```
dc_shell-topo> set_app_var link_library "* stdlib.db macro_corner1.db \
               macro_corner2.db"
dc_shell-topo> create_scenario A
dc_shell-topo> set_link_library_subset "macro_corner1.db" \
               -object_list [get_cells u1/my_macro]
dc_shell-topo> create_scenario B
dc_shell-topo> set_link_library_subset "macro_corner2.db" \
               -object_list [get_cells u1/my_macro]
```

9

Using Hierarchical Models

Hierarchical models are used in Design Compiler to reduce the number of design objects and the memory requirements when the tool performs top-level optimization on large designs. The term *hierarchical models* refers to block abstractions, physical hierarchies, and interface logic models (ILMs).

To learn about generating and using hierarchical models in Design Compiler, see

- Overview of Hierarchical Models
- Block Abstraction Hierarchical Flow
- Interface Logic Model Hierarchical Flow

For information about using hierarchical models in a bottom-up compile flow, see Performing a Bottom-up (Hierarchical) Compile.

Overview of Hierarchical Models

Hierarchical models are used in Design Compiler to reduce the number of design objects and the memory requirements when the tool performs top-level optimization on large designs. For ILMs, the gate-level netlist for a block is modeled by a partial gate-level netlist that contains only the required interface logic of the block and possibly the logic that you manually associate with the interface logic. All other logic is removed.

Block abstractions are an extension to interface logic models. Unlike an ILM where the internal logic is removed, all the logic is retained and only the interface logic is loaded when using the block abstraction.

Design Compiler can create ILMs and block abstractions, and it can use ILMs and block abstractions created either in Design Compiler or in IC Compiler. However, IC Compiler can only use ILMs and block abstractions created in IC Compiler. For optimal correlation and alignment between the tools, use IC Compiler block abstractions or IC Compiler ILMs to model the physical blocks in both tools.

Note:

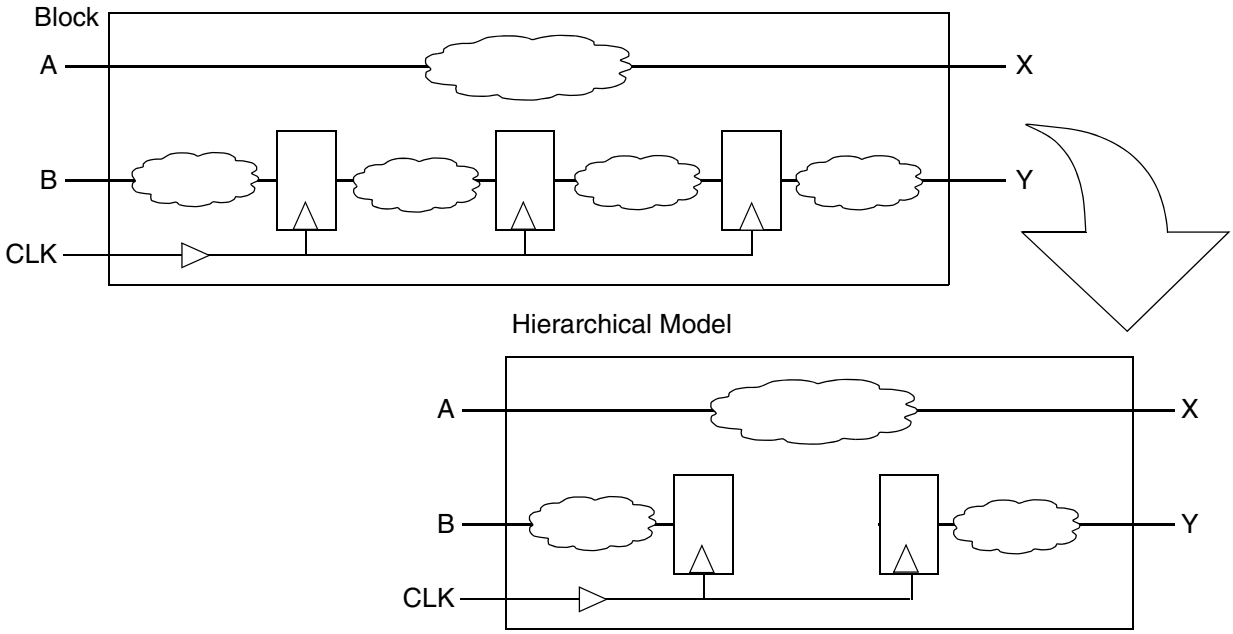
You cannot combine block abstractions and ILMs in the same flow.

Figure 9-1 shows a block and its hierarchical model, where the logic is preserved between

- The input port and the first register of each timing path
- The last register of each timing path and the output port

Logic associated with pure combinational input-port-to-output-port timing paths (A to X) is also preserved. Clock connections to the preserved registers are kept as well. The register-to-register logic is discarded.

Figure 9-1 A Block and Its Hierarchical Model



Design Compiler supports the use of hierarchical models in a variety of flows, including

- Multicorner-multimode

  Design Compiler automatically detects the presence of multiple corners and multiple modes and retains the interface logic involved in the interface timing paths of each scenario.

- Designs with multiple levels of physical hierarchy

  You can use nested block abstractions to model designs with multiple levels of physical hierarchy. For more information about nested block abstractions, see Creating Block Abstractions for Nested Blocks.

  You can use nested ILMs to model designs with multiple levels of physical hierarchy. For more information about nested ILMs, see Creating ILMs for Blocks With Nested ILMs.

Information Used in Hierarchical Models

By default, hierarchical models include the following netlist objects:

- Leaf cells and macro cells in the four critical timing paths that lead from input ports to output ports (combinational input-to-output paths), input ports to edge-triggered registers, and edge-triggered registers to output ports
- Abstracted clock trees that include clock paths to interface registers, minimum and maximum clock paths that could also be connected to noninterface registers, and the clock tree synthesis exceptions that are a part of these clock paths

  Design Compiler treats generated clocks like clock ports. Design Compiler retains interface registers that are driven by a generated clock. However, internal registers driven by generated clocks are not retained. When the design contains generated clocks, the logic along the timing path between the master clock and the generated clock is also retained.
- Clock-gating circuitry—if it is driven by external ports
- Logic along the timing path between a master clock and any generated clocks derived from it
- Side-load cells for all nets

  Side-load cells are cells that can affect the timing of an interface path but are not directly part of the interface logic.

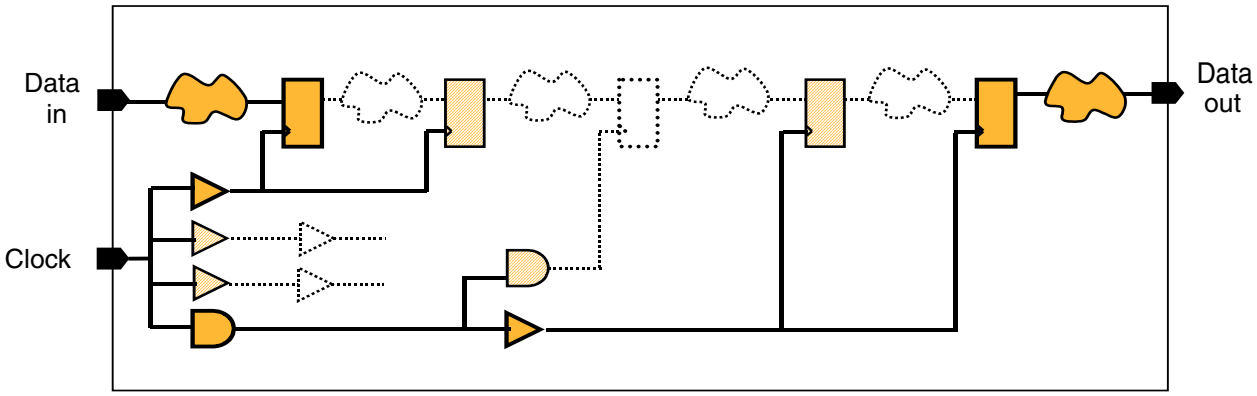
Note:
By default, latches are treated as combinational logic.

Apart from the netlist objects, the following information is also stored in a hierarchical model:

- Cell or pin location information
- Parasitic information
- Timing constraints
- Pin-to-pin delay for internal nets
- Power consumption

Figure 9-2 shows the elements contained in a hierarchical model, emphasizing the clock tree. The darkly shaded elements are the elements on the clock path; the lightly shaded elements are the side-load cells. Unshaded elements are not included in the model.

Figure 9-2 Hierarchical Model



Using Hierarchical Models in a Multicorner-Multimode Flow

You can apply multicorner-multimode constraints to a hierarchical model and use those constraints in a top-level design.

The following requirements apply to using blocks that are modeled using a hierarchical model with multicorner-multimode scenarios:

- For each top-level multicorner-multimode scenario, an identically named scenario must exist in each of the blocks used in the design.

  If there is a mismatch, use the `select_block_scenario` command to map the scenarios in the blocks to the top-level design. A block can have additional scenarios that are not used at the top level.

  The syntax for using the `select_block_scenario` command is

  ```
  select_block_scenario
     [-scenarios top_scenarios]
     [-block_references list_design_names]
     -block_scenario block_scenario_name | -reset
  ```

  To reset user-specified multicorner-multimode scenario mapping, use the `-reset` option.

- You can use a top-level design with multicorner-multimode scenarios with blocks that do not have multicorner-multimode scenarios. This does not require the use of the `select_block_scenario` command.

- By default, in a top-level design without multicorner-multimode scenarios, only blocks without multicorner-multimode scenarios can be used.

  To use multicorner-multimode blocks with non-multicorner-multimode top-level designs, specify the scenario mapping by using the `select_block_scenario` command.

- For each TLUPlus file, a block stores the extraction data at the specified temperature, which is an operating condition. For accurate results during parasitic extraction at the top level, the TLUPlus files and temperature corners at the top level should match the TLUPlus and temperature corners used during block-level implementation. However, you can use additional TLUPlus and temperatures at the top level with loss in accuracy.

Viewing Hierarchical Models in the GUI

In the Design Compiler GUI, a hierarchical model instance is displayed as a level of physical hierarchy: a rectangular or rectilinear shape with a fill pattern. The View Settings panel provides additional control over the block abstraction or ILM display.

- To display the hierarchical models, select the Cell > ILM/Block Abstraction visibility option.
- To display the hierarchical model pins, select the Pin visibility option.
- To expand the hierarchical model to show the leaf cells, pins, nets, and macros within, type or select a positive integer in the “Level” box.
- To reset the display to the default view, the unexpanded view, type or select 0 in the “Level” box.

For more information about using the GUI to view physical hierarchy blocks, see the “Examining Physical Hierarchy Blocks” topic in Design Vision Help.

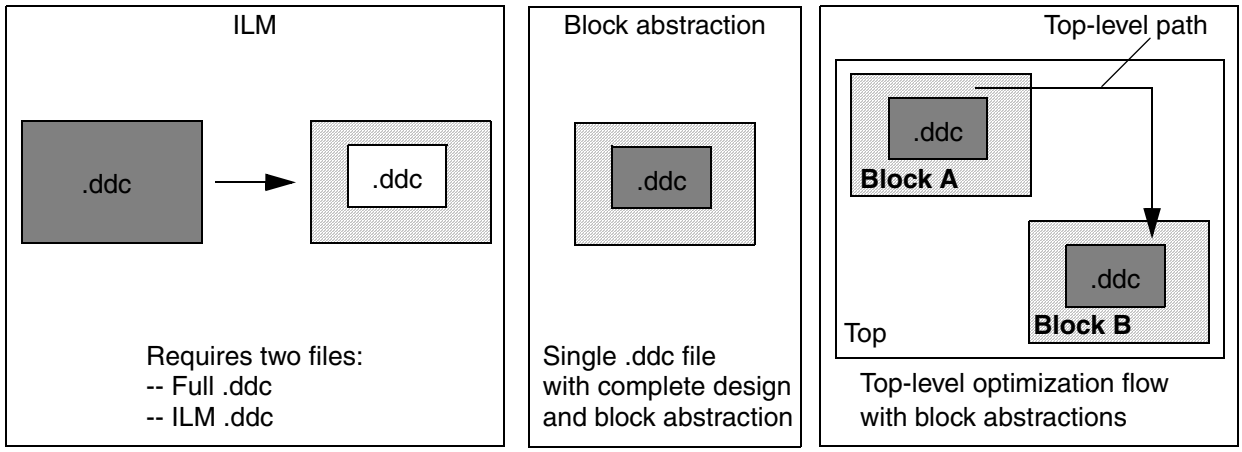
Block Abstraction Hierarchical Flow

A top-level flow using interface logic models (ILMs) allows optimization of only the top-level logic. Block abstractions are an extension of interface logic models. Unlike an ILM where the internal logic is removed, all the logic is retained in a block abstraction, but only the interface logic is loaded. This supports faster top-level design closure by allowing optimization of block-level interface logic during top-level synthesis.

When you create and save block abstractions, they are stored in a single .ddc file in the same file that includes the complete design netlist. You do not need to maintain two .ddc files, one for interface logic and the other for the full netlist, as you need to do for ILMs.

Figure 9-3 illustrates the difference between an ILM and a block abstraction.

Figure 9-3 ILM and Block Abstraction Comparison



By default, block abstractions are not modified or optimized. If you are using Design Compiler in topographical mode, you can optimize the block abstractions using top-level transparent interface optimization. Transparent interface optimization is the process that optimizes the interface logic within the blocks during top-level synthesis to achieve faster timing closure.

Transparent interface optimization provides the following benefits:

- Supports faster top-level design closure by performing concurrent optimization of top-level logic and block-level interfaces.
- Reduces iterations between top-level and block-level implementation.

Design Compiler can create ILMs and block abstractions and can use ILMs and block abstractions created either in Design Compiler or in IC Compiler. However, IC Compiler can only use ILMs and block abstractions created in IC Compiler. For information about creating a block abstraction in IC Compiler, see the *IC Compiler Implementation User Guide*.

Note:

You cannot combine block abstractions and ILMs in the same flow.

The following sections describe the block abstraction hierarchical flow:

- Creating and Saving Block Abstractions
- Setting Top-Level Implementation Options and Transparent Interface Optimization
- Resetting Implementation Options
- Loading Block Abstractions
- Reporting Implementation Options

- Reporting Block Abstractions
- Querying Block Abstractions
- Checking Block Abstractions
- Performing Top-Level Synthesis
- Saving Optimized Block Abstractions After Top-Level Synthesis
- Limitations

Creating and Saving Block Abstractions

To generate a block abstraction for the current design, use the `create_block_abstraction` command at the end of synthesis. The `create_block_abstraction` command adds the block abstraction information to the design in memory.

To save the block abstraction, you must use the `write_file` command immediately after creating the block abstraction:

```
dc_shell-topo> create_block_abstraction
dc_shell-topo> write_file -hierarchy -format ddc -output ${DESIGN_NAME}.mapped.ddc
```

The .ddc file contains both the complete design and the Design Compiler block abstraction.

You can create block abstractions in either Design Compiler or Design Compiler in topographical mode. However, you must use topographical mode to optimize block abstractions, and only block abstractions created in topographical mode are optimized.

To specify the list of block references that you want to link to block abstractions and integrated with the top-level design, and to optimize the block abstractions using transparent interface optimization, use the `set_top_implementation_options` command as described in Setting Top-Level Implementation Options and Transparent Interface Optimization.

Information Used in Block Abstractions

When you use the `create_block_abstraction` command, the tool

- Identifies the interface logic.

  The following netlist objects are part of interface logic:

  - All cells, pins, and nets in timing paths from input ports to registers or output ports.
  - All cells, pins, and nets in timing paths to output ports from registers or input ports.
  - Any logic in the connection from a master clock to generated clocks.

  - The clock trees that drive interface registers, including any logic in the clock tree.
  - The longest and shortest clock paths from the clock ports.
  - The side-load cells of all non-ideal and non-DRC disabled nets.
- Ignores an input or inout port if the percentage of the total registers in the transitive fanout of the port is greater than or equal to the threshold percentage that is specified by using the `abstraction_ignore_percentage` variable.

  The following netlist objects are retained for ports that are ignored:

  - Connections to the already identified interface logic of the other ports.
  - Minimum and maximum critical timing paths.
- Ignores any case analysis settings. Any case analysis settings that are required must be specified at the top level by using the `set_case_analysis` command.
- Assumes all latches found in interface logic are potential borrowers; thus, all logic from I/O ports to flip-flops or output ports are identified as belonging to interface logic.
- Extracts and stores detailed parasitics as part of the abstraction information for each specified TLUPlus file and temperature.
- Calculates power consumption of the design and stores that information in attributes in the design. This information is used by the `report_power` command during the final assembly step of the entire chip.

To include additional leaf cells and nets in the block abstraction, use the `create_block_abstraction` command with the `-include` option. Logical hierarchical cells and pins are ignored.

If either of the following conditions exist, the entire block is included in the block abstraction:

- The number of leaf cells included exceeds 95 percent of the total leaf cells in the block.
- The number of nets included exceeds 95 percent of the total nets in the block.

Block Abstractions for Multicorner-Multimode Usage

The `create_block_abstraction` command automatically detects the presence of multiple scenarios (multiple modes or corners) and determines the interface logic for each scenario. The interface logic identified for each scenario is retained as the interface logic of the block abstraction. If an interface timing path is disabled in one scenario but enabled in another, the path is included in the interface logic.

Setting Top-Level Implementation Options and Transparent Interface Optimization

To use block abstractions at the top level, use the `set_top_implementation_options` command with the `-block_references` option to specify the list of block references that you want to link to the block abstraction. Do this for both Design Compiler block abstractions and IC Compiler block abstractions.

Use the `set_top_implementation_options` command before reading the .ddc file with the block abstraction into memory; otherwise, the full block netlist is loaded. When linking, the tool loads only the interface logic to the top level.

When using IC Compiler block abstractions, use the `set_top_implementation_options` command before running any `link` command in Design Compiler. The IC Compiler block abstractions are automatically read from the list of Milkyway reference libraries while linking the top-level design.

By default, block abstractions are not modified or optimized. If you are using Design Compiler in topographical mode, you can optimize the block abstractions using top-level transparent interface optimization. Transparent interface optimization is a process that optimizes the interface logic within the blocks during top-level synthesis to achieve faster timing closure.

You can enable transparent interface optimization when using the `compile_ultra` or `compile_ultra -incremental` command by using the `set_top_implementation_options -optimize_block_interface true` command and specifying the list of blocks whose interface logic should be optimized during top-level synthesis. During top-level synthesis, Design Compiler links the block abstractions to the top-level design, optimizes and modifies each block's interface logic, updates the top-level interface logic, and performs full optimization.

Optimization of block abstractions is limited to cell sizing only; therefore, the `-size_only_mode` option has no effect. Transparent interface optimization is available only in Design Compiler in topographical mode for block abstractions created in topographical mode. Transparent interface optimization cannot be performed on block abstractions created in IC Compiler. These can only be optimized in IC Compiler.

The following example sets the top-level implementation options for the blk1 and blk2 blocks:

```
dc_shell-topo> set_top_implementation_options -block_references {blk1 blk2}
```

The following example enables transparent interface optimization for the blk1 block:

```
dc_shell-topo> set_top_implementation_options -block_references blk1 \
                  -optimize_block_interface true
```

You can use multiple `set_top_implementation_options` commands to specify the options for all of the block abstractions at the top level.

The command settings in Example 9-1 and Example 9-2 are equivalent.

Example 9-1 Specifying Options for Block Abstractions in One Command

```
set_top_implementation_options -block_references {blk1 blk2}
```

Example 9-2 Specifying Options for Block Abstractions in Multiple Commands

```
set_top_implementation_options -block_references {blk1}
set_top_implementation_options -block_references {blk2}
```

To verify the options set by the `set_top_implementation_options` command and ensure that the block abstractions are loaded correctly, see Reporting Implementation Options.

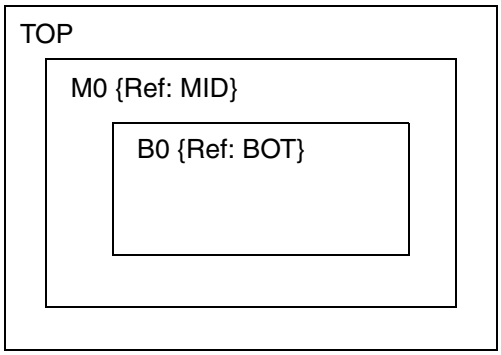
Creating Block Abstractions for Nested Blocks

You can use the `create_block_abstraction` command to create block abstractions for blocks that include nested block abstractions. The command does not create block abstractions with other types of physical hierarchy.

When you use nested block abstractions, you must specify the references of the blocks instantiated in all levels of the hierarchy, as shown:

```
dc_shell-topo> set_top_implementation_options -block_references {MID BOT}
```

Figure 9-4 Block Abstraction With Nested Physical Hierarchy



You can use transparent interface optimization to optimize the interface logic at any level.

For multicorner-multimode designs, you need to specify scenario mapping from the top level to each of the levels by using the `select_block_scenario` command.

Note:

Visibility of nested block abstractions in the layout view is not yet available.

Resetting Implementation Options

You can reset the top implementation options by using the `-reset` option with the `set_top_implementation_options` command. This causes the full design netlist to be loaded for the block instead of loading the block abstraction.

The following example resets the top implementation options for the blk2 block. In this example, the block abstraction for blk1 is loaded, and the full netlist for blk2 is loaded:

```
dc_shell-topo> set_top_implementation_options -block_references {blk1 blk2}
dc_shell-topo> set_top_implementation_options -block_references {blk2} -reset
```

Loading Block Abstractions

Use the `read_ddc` command after you have used the `set_top_implementation_options` command to load Design Compiler block abstractions to be linked to the top-level design:

```
dc_shell-topo> read_ddc {DesignA.mapped.ddc DesignB.mapped.ddc}
```

The following message indicates that the Design Compiler block abstraction is being loaded while reading in the block .ddc file:

```
Information: Reading block abstraction for design 'blk1' and its hierarchy. (DDC-27)
```

Make sure that your Milkyway reference libraries include the IC Compiler block abstraction designs. IC Compiler block abstractions are automatically loaded from the list of Milkyway reference libraries when linking the top-level design.

The following message indicates that the IC Compiler block abstraction is being loaded while linking the top-level design:

```
Information: Auto loading blk2.CEL (version 1) from Milkyway library blk2.mw ...
(ILM-102)
```

After a block abstraction is loaded, you can perform the tasks discussed in the following sections to ensure the block abstraction is ready for top-level synthesis:

- Reporting Implementation Options
- Reporting Block Abstractions
- Querying Block Abstractions
- Checking Block Abstractions

Reporting Implementation Options

To verify the options set by the `set_top_implementation_options` command and ensure that the block abstractions are correctly loaded, use the `report_top_implementation_options` command. The `report_top_implementation_options` command reports the options as shown in the following example:

```
dc_shell-topo> report_top_implementation_options

****************************************
Report    : Top implementation options
...
****************************************

------------------------------------------------------------------------
Block       Optimize    Optimize       Size Only   Host
Reference   Interface   Shared Logic   Mode        Options
------------------------------------------------------------------------
block1      true        false          off         --
block2      true        false          off         --
------------------------------------------------------------------------
```

Reporting Block Abstractions

To report block abstractions that are linked to the current design, use the `report_block_abstraction` command. The command reports the following information:

- Block reference name
- Block instance name
- Block version
- Full path of the library from which the block was loaded
- Date and time of the last modification of the block
- Cell count and compression statistics for each of the blocks
- Cell count and compression statistics for the top-level design with blocks
- TLUPlus settings
- Parasitics data of the block
- User-defined multicorner-multimode scenario mappings between the top level and blocks
- UPF information

The `report_block_abstraction` command reports all levels of the hierarchy for nested block abstractions.

The following example shows a block abstraction report. The report includes a Design Compiler topographical block abstraction (blk1) and an IC Compiler block abstraction (blk2).

```
*************************************
Report  : Block Abstraction
Design  : top
...
*************************************

###############################
Top-level details
###############################

------------------------------------------------------------------------
Block Instance               Reference      Orient    Location
------------------------------------------------------------------------
sub/blk1                     blk1           0         (23.78, 1127.96)
sub/blk2                     blk2           0         (809.74, 25.88)
------------------------------------------------------------------------

Top design leaf cell count statistics:
--------------------------------------
Block instance count             : 2
Top only cell count              : 184148
Top + interface logic cell count : 239527
Top + Block cell count           : 320803
Compression percentage           : 25.34

Top-level and block scenario mapping:
-------------------------------------
There is no user-specified scenario mapping available.

###############################
Block reference : blk1
###############################

File name: blk1.ddc
Last modified(mm/dd/yyyy hr:min:sec): 3/6/2012 0:42:46

Leaf cell count statistics:
---------------------------
Interface logic cell count       : 19142
Block cell count                 : 46219
Compression percentage           : 58.58

TLU+ files used:
----------------
Block scenario #0: timing (mapped from top-level scenario timing)
Min TLU+ file           : min.tlup
Max TLU+ file           : max.tlup
Tech2ITF mapping file   : tech.map

Block scenario #1: leakage (mapped from top-level scenario leakage)
Min TLU+ file           : min.tlup
```

```
Max TLU+ file           : max.tlup
Tech2ITF mapping file   : tech.map

##############################
Block reference : blk2
##############################

File name: blk2.mw/CEL/blk2:1
Last modified(mm/dd/yyyy hr:min:sec): 3/5/2012 21:44:9

Leaf cell count statistics:
---------------------------
Interface logic cell count       : 36237
Block cell count                 : 90436
Compression percentage           : 59.93

TLU+ files used:
----------------
Block scenario #0: timing (mapped from top-level scenario timing)
Min TLU+ file           : min.tlup
Max TLU+ file           : max.tlup
Tech2ITF mapping file   : tech.map

Block scenario #1: leakage (mapped from top-level scenario leakage)
Min TLU+ file           : min.tlup
Max TLU+ file           : max.tlup
Tech2ITF mapping file   : tech.map
1
```

Querying Block Abstractions

To create a collection of block abstractions, use the `get_cells` command:

```
dc_shell-topo> get_cells -hierarchical -filter is_block_abstraction==true
{blk1 blk2}
```

Checking Block Abstractions

Use the `check_block_abstraction` command to check the readiness of a block abstraction before optimization. The `check_block_abstraction` command checks the blocks linked to the top-level design to ensure they are ready for top-level synthesis.

The command checks for the following:

- Scenario consistency in multicorner-multimode blocks.
- The existence of `dont_touch` settings on blocks and cells.

  All objects must have the `dont_touch` attribute so they cannot be modified.

The following example checks the blocks in the current design and reports the errors to be resolved:

```
dc_shell-topo> check_block_abstraction
Error: No scenario data loaded from block reference 'blk1'. (ILM-151)
Error: dont_touch attribute incorrectly set to false on block design 'blk2'. (ILM-121)
0
```

You can also use the `compile_ultra -check_only` command to verify that the design and libraries have all the data that `compile_ultra` requires to run successfully. The `-check_only` option is only available in Design Compiler in topographical mode.

Performing Top-Level Synthesis

After you create the block abstraction, specify the settings linking it to the top-level design, enable transparent interface optimization (if you are using topographical mode and want to optimize the block abstraction), run a report to check the top implementation settings, run a report to get details about the block abstraction, and check the readiness of the block abstraction, you can perform top-level synthesis.

The following script shows part of a top-level synthesis flow with transparent interface optimization enabled. The top-level design contains the DesignA and DesignB blocks and their corresponding block abstractions, DesignA.mapped.ddc and DesignB.mapped.ddc.

```
set BLOCK_ABSTRACTION_DESIGNS "DesignA DesignB"
set_top_implementation_options -block_references ${BLOCK_ABSTRACTION_DESIGNS} \
-optimize_block_interface true
... #Read in the top-level RTL
read_ddc {DesignA.mapped.ddc DesignB.mapped.ddc}
current_design top
link
report_block_abstraction
get_cells -hierarchical -filter is_block_abstraction==true
report_top_implementation_options
...# Read in constraints
check_block_abstraction
compile_ultra -check_only
compile_ultra
```

For the complete top-level hierarchical flow using block abstractions, see Performing a Bottom-up (Hierarchical) Compile.

Saving Optimized Block Abstractions After Top-Level Synthesis

When you save the top-level design after optimization, block abstractions are not written to the top.ddc file. If you enabled transparent interface optimization, you need to save the updated blocks. Use the `write_file` command to save each optimized block abstraction to a separate .ddc file. You cannot write more than one block abstraction design in a single

output .ddc file. The `write_file` command merges the changes with the original .ddc design and writes the complete block as a new .ddc file.

The `-hierarchy` and `-output` options are required with the `write_file` command when you write out block abstractions at the top level, and you must provide only the top-level block abstraction design name as an argument.

The following script shows how to save the top-level design and the optimized block abstractions into separate .ddc files:

```
write_file -hierarchy -format ddc -output top.mapped.ddc
foreach design ${BLOCK_ABSTRACTION_DESIGNS}{
  write_file -hierarchy -format ddc \
             -output ${design}.mapped_tio.ddc \
             ${design}
}
```

Limitations

Block abstractions have the following limitations:

- You cannot combine ILMs and block abstractions in the same flow.
- IC Compiler does not support the loading of ILMs and block abstractions that were created in Design Compiler.
- You cannot view nested block abstractions in the layout view.

Interface Logic Model Hierarchical Flow

An interface logic model (ILM) is a structural model that describes the interface logic of a block. Logic that is not required for modeling boundary timing is discarded, reducing the memory and CPU time needed to optimize the block. The ILM retains the cells whose timing is affected by or affects the external environment of the block; therefore, ILMs are highly accurate timing representations of the original block. Names of cells and data paths are retained in the model, which is helpful for debugging.

The use of ILMs in Design Compiler is described in the following sections:

- Creating ILMs
- Validating ILMs
- Reporting Information About ILMs
- Using ILMs

Creating ILMs

This section describes how to use the `create_ilm` command to create ILMs and how to control the type and quantity of logic in an ILM in the following subsections:

- Overview
- Creating and Saving an ILM
- Creating ILMs for Multiply Instantiated Designs
- Creating ILMs for Multicorner-Multimode Usage
- Creating ILMs for IEEE 1801 Unified Power Format (UPF) Flows
- Creating ILMs for Rotated and Mirrored Instances
- Creating ILMs for Blocks With Nested ILMs
- Controlling the Logic Included in an ILM
- Storing Parasitic Information in an ILM

Overview

To create an ILM, use the `create_ilm` command. The various options of this command allow you to control how the interface logic is modeled. When you create an ILM, the `create_ilm` command,

- Identifies the interface logic to be included in the model. You can specify various command options to control the logic that is included as part of a block's interface.
- Extracts the logic and builds the model. Certain command options can be used to include side-load cells, which is technically not part of the interface logic, and to include physical design information in the model.
- Automatically marks the ILM with the `is_interface_model attribute`.

  Note:
  In the topographical mode, the tool does not support the `propagate_ilm` command. However, it propagates information about ILM blocks by default.
- Does not create an ILM and issues an error message if the current design contains sequential elements but a clock is not defined.
- Creates an ILM and issues a warning message if the current design does not contain any sequential elements. However, this ILM is the same size as the original design.
- Automatically applies the `dont_touch` attribute to the ILM cells.

- Maintains the hierarchy of the original netlist being modeled as an ILM. You can flatten an ILM by using the `ungroup -all -flatten` command.
- Allows propagated clocks to retain side loads if the option to keep side loads is used. Generated clocks are treated like clock ports. Interface registers driven by the generated clock are retained in the model. Registers driven by the generated clock that are in internal register-to-register paths are not included in the model.

Note:

For ILMs created by IC Compiler, the Milkyway design library ILM view name must match the design name. For the CEL views created by IC Compiler, the Milkyway design library CEL view name must match the design name.

Creating and Saving an ILM

To create an ILM,

1. Read each mapped block design that you want to model and link each design.
2. Identify all clocks by using commands such as `create_clock` and `create_generated_clock`.
3. Use the `write_interface_timing` command to create a timing report for the original netlist. This timing report is compared to the timing report for the ILM to help verify consistency between the ILM and the original netlist.

   If you are using multiple scenarios, see Example 9-3 for the flow to verify consistency between the original netlist and the ILM.
4. Create the ILM by using the `create_ilm` command for each block you chose.

   After command execution, the design in memory is the ILM for the block. It has the same name as the original design and becomes the current design.

   Note:

   To create a customized model that selectively includes (or excludes) certain cells, nets, and pins, use the `create_ilm -identify_only` and `create_ilm -extract_only` commands. The `create_ilm -extract_only` command does not identify new interface logic elements. For details, see Controlling the Logic Included in an ILM.

   Use the `get_ilm_objects` command to report the objects that have been identified as interface logic objects. You can repeat this step, changing option settings as needed, until you obtain the desired collection of interface objects. You can then use the `create_ilm -extract_only` command to extract the currently identified interface logic objects and build the ILM with them.
5. To create a timing report for the ILM, use the `write_interface_timing` command. Compare this timing report with the timing report for the netlist to help verify the consistency between the ILM and the original netlist.

6. Compare the original netlist and ILM interface timing reports by using the `compare_interface_timing` command. This command checks for consistency between the ILM and the original netlist. For example, the following command compares the report for the original netlist in the original.wit file with the report for the ILM in the ilm.wit file and writes the comparison report to the check.cit file.

   ```
   compare_interface_timing original.wit ilm.wit -output check.cit
   ```

7. Save the model of the block design by using the `write -format ddc` command. Design Compiler supports only the .ddc format for writing out the ILMs.

   By default, the `write_file` command saves only the top-level design. Use the `-hierarchy` option to save the entire design. If you do not use the `-output` option to specify the output file name, the `write_file -format ddc` command creates a file called top_design.ddc, where *top_design* is the name of the current design.

   To save an ILM to a file named design.ILM.ddc:

   ```
   write_file -format ddc -hierarchy -output design.ILM.ddc
   ```

   To ensure that the created ILM is correct, before saving the ILM,

   - Check the log file to ensure there are no errors.
   - Check the return status of the `create_ilm` command. If the command returned a 1, the command succeeded. This is the preferred method of checking because it can be scripted.

Creating ILMs for Multiply Instantiated Designs

Design Compiler can create ILMs for multiply instantiated designs without relying on the `uniquify` command.

When you create an ILM for a block-level design that contains multiple instantiations of a hierarchical cell, you do not need to run the `uniquify` command on the subdesign first. By default, the `compile` command uniquifies the hierarchical cells. You can then create an ILM for the mapped block and use it in the top-level design.

At the top level, multiply instantiated ILMs do not need to be uniquified with the `uniquify` command. Note, however, that running the `compile` command at the top level does not automatically uniquify any multiply instantiated ILMs.

Creating ILMs for Multicorner-Multimode Usage

If you are using a multicorner-multimode flow, the `create_ilm` command automatically detects the presence of multiple corners and multiple modes and retains the interface logic involved in the critical interface timing paths for each scenario.

You must create an ILM by using the `create_ilm -compact none -traverse_disabled_arcs` command if the following conditions exist:

- The ILM is instantiated multiple times in the top-level design.
- One instance of the ILM is used in a mode that is different from the other instances of the same ILM in the top-level scenario.

Do not use different modes of the same ILM reference block at the same time in a multiply instantiated module design in Design Compiler.

Creating ILMs for IEEE 1801 Unified Power Format (UPF) Flows

If you are using a UPF flow, the `create_ilm` command automatically retains the following UPF objects:

- All power-switch cell instances, irrespective of whether they are part of the interface logic or not, for proper supply net connectivity and always-on synthesis
- All retention registers, irrespective of whether they are part of the interface logic or not
- Level-shifter cells and isolation cells that are part of the interface logic
- Control logic for the retained power-switch cells, retention registers, isolation cells, and level-shifter cells

When Design Compiler removes a UPF netlist object, it either removes or updates the associated UPF constraint. The tool ensures that the UPF strategies stored in the hierarchical model are consistent with the UPF netlist objects.

When using hierarchical models with UPF at the top level in Design Compiler, you must propagate the UPF objects to the top level using the `propagate_constraints -power_supply_data` command.

Creating ILMs for Rotated and Mirrored Instances

Design Compiler in topographical mode supports rotated and mirrored ILMs. You do not need to create a separate ILM for each orientation of a block because Design Compiler supports all orientations of the ILM. Design Compiler propagates the leaf cell locations and the port locations of each ILM instance to the top-level design, based on the location and orientation of the ILM instances.

All eight orientations—north, flipped (mirrored) north, south, flipped (mirrored) south, east, flipped (mirrored) east, west, and flipped (mirrored) west—are allowed. The default orientation is N (north with 0-degree rotation).

You must define the orientation of an ILM block in the floorplan. When an ILM block is instantiated in the top-level design, the physical area that it occupies in the core area depends on the orientation of the block.

Creating ILMs for Blocks With Nested ILMs

A nested ILM is an ILM that is contained in another ILM. When you run the `create_ilm` command on a block that contains nested ILMs, the tool retains only the interface logic for the current block. Logic from a nested ILM is retained only if it is part of the interface paths for the current block.

In the following example, the MID block contains an instance of the BOT block. Figure 9-5 shows the ILM for the BOT block.

Figure 9-5 Interface Logic Model for the BOT Block

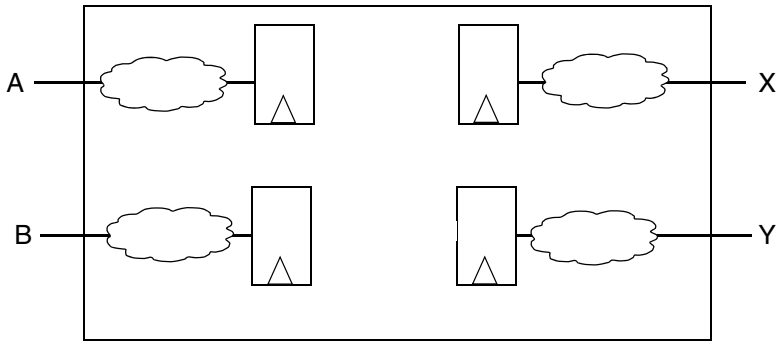
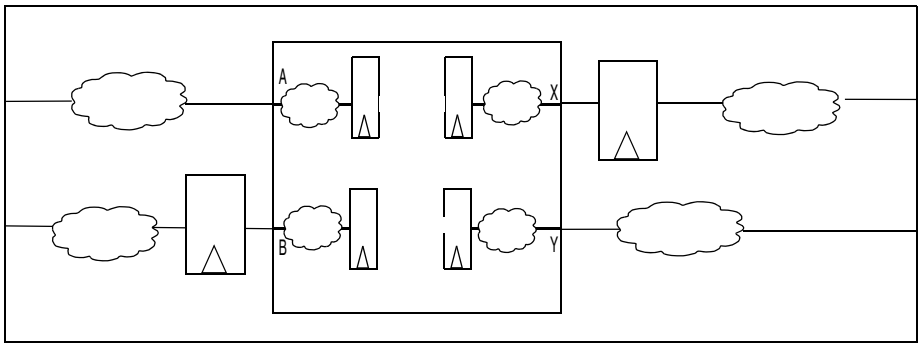


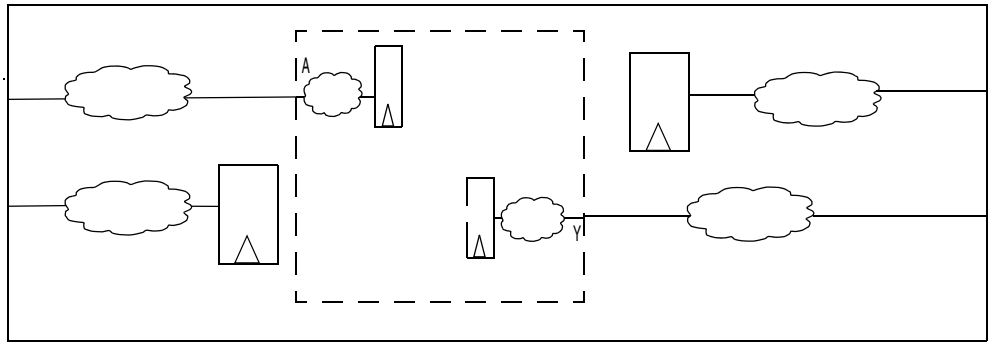
Figure 9-6 shows the MID block, which contains the ILM for the BOT block.

Figure 9-6 MID Block That Contains the ILM for the BOT Block



When you run the `create_ilm` command on the MID block, the tool discards the logic connected to the B and X pins of the BOT ILM, because it is not part of the interface logic for the MID block. Figure 9-7 shows the ILM for the MID block. The dotted line indicates the logical hierarchy.

Figure 9-7 Interface Logic Model for the MID Macro



Controlling the Logic Included in an ILM

This section describes, in the following topics, the various options of the `create_ilm` command that are used to control the logic retained in an ILM when creating it:

- Changing the Compaction Level
- Retaining Additional Logic
- Additional Controls for Noncompact ILMs

Changing the Compaction Level

The compaction level controls the amount of logic retained in the ILM. Specify the compaction level by using the `-compact` option when you run the `create_ilm` command. Table 9-1 describes the various arguments that are available with the `-compact` option.

Table 9-1 The create_ilm Command Options for Compaction

Option	Description
`-compact all` (default)	Performs compaction on all input paths and all output paths and results in the smallest ILM, while still maintaining timing accuracy. The created ILM contains the interface logic only for the four critical timing paths (minimum rise, minimum fall, maximum rise, and maximum fall) from each input port and to each output port. Additionally, if there are input-to-output combinational paths in a block, the minimum rise, minimum fall, maximum rise, and maximum fall paths for input-to-output combinational paths are included in the compact ILM. This is the default ILM.

Table 9-1 The create_ilm Command Options for Compaction (Continued)

`-compact output`	Performs compaction only on the output paths and retains all interface logic on the input paths, resulting in a larger ILM.
`-compact none`	Disables compaction and retains all interface logic on both the input paths and the output paths and results in the largest ILM. If your design does not have complete and correct SDC files, you must use this option because compaction is slack-based for each port.

The amount of compaction also depends on the design style.

- Designs with registered inputs and outputs will have the greatest reduction in size when the original netlist is compared to its ILM netlist.
- Some design types, such as pure combinational logic blocks or latch-based designs, do not show much reduction. The interface logic for such designs tends to contain much of the original design.

Retaining Additional Logic

Design Compiler provides several options that enable you to retain logic in addition to the default interface logic. Table 9-2 describes these options.

Table 9-2 The create_ilm Command Options for Retaining Additional Logic

Option	Description
`-keep_full_clock_tree`	By default, Design Compiler retains abstracted clock trees that include the minimum and maximum clock paths, the clock path to the interface registers, and the clock tree synthesis exceptions that are a part of these paths. To keep the entire clock trees, use the `-keep_full_clock_tree` option.
`-keep_macros`	By default, Design Compiler retains only the macro cells that are part of the interface logic. To retain all macros in the design, use the `-keep_macros` option. A cell is considered to be a macro cell if it is specified as a macro in the physical library or if it is a large black box cell.
`-include_all_logic`	By default, Design Compiler retains only the interface logic. To retain all the design logic in the ILM, use the `-include_all_logic` option. You should use this option only when you need to retain the entire logic in the ILM, such as for a clock generator block or for a very small block.

Table 9-2 The create_ilm Command Options for Retaining Additional Logic (Continued)

Option	Description
`-must_connect_ports`	Use the `-must_connect_ports` option to specify a list of ports, such as scan enable or reset ports, that must be connected to already identified logic for functional correctness of the design.
`-case_controlled_ports`	If you apply case-analysis constraints on a block, the disabled logic is removed from the ILM even if the port is specified as a must-connect port because that logic is usually not needed. However, if the case-analysis values on an input or inout port change when linked to the top-level design, you must retain all interface logic that is associated with this port. To retain all the interface logic on case-controlled ports, use the `-case_controlled_ports` option to specify the affected ports. When you use this option, the ILMs that are created are independent of any block-level case-analysis constraints placed on these ports. After creating an ILM that is case-analysis independent for the specified ports, apply case-analysis constraints from the top level to the ILM cell as desired for the particular application.

Controlling Side-Load Cells

By default, Design Compiler retains the side-load cells for all nets. Table 9-3 describes the various arguments of the `create_ilm -include_side_load` command for including side-load cells in the ILM.

Table 9-3 ILM Creation Command Options for Controlling Side-Load Cells

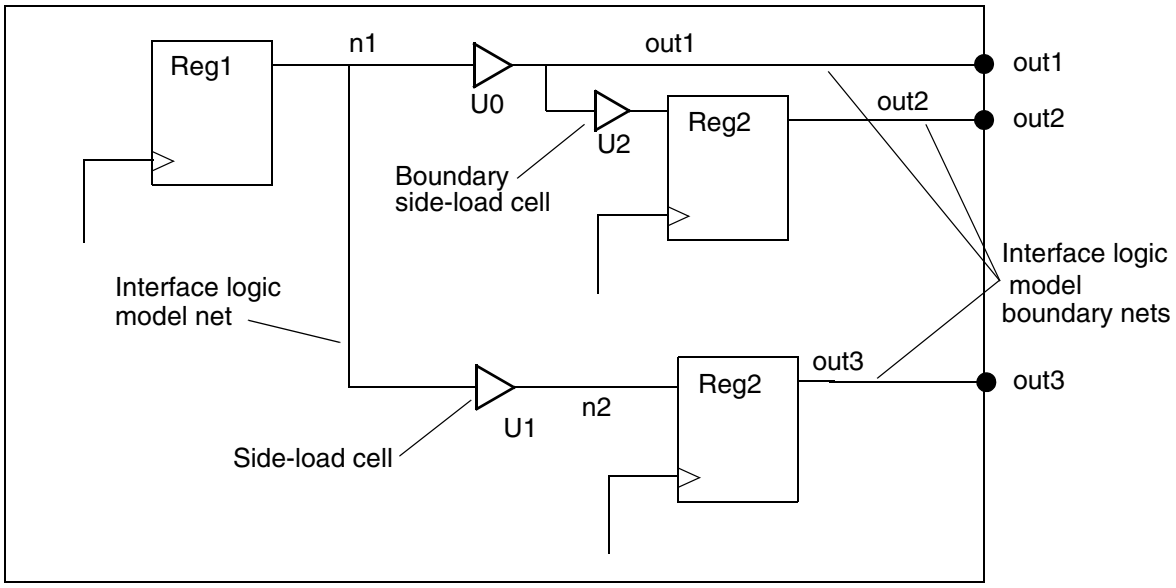
Option	Description
`-include_side_load all` (default)	This option includes all side-load cells. Although using this option increases the ILM size, it can improve the timing accuracy of the ILM.
`-include_side_load boundary`	This option includes only boundary side-load cells.
`-include_side_load none`	This option includes no side-load cell. This option creates the smallest ILM, but it is also the least accurate.

Note:

User-controlled handling of side-load cells can affect the timing of an interface path, even when side-load cells are not in the interface logic.

Figure 9-8 shows an example of a side-load cell and a boundary side-load cell. In this figure, cell U1 is a side-load cell because it places a load capacitance on the interface logic net n1. Cell U2 is a boundary side-load cell because it places a load capacitance on the boundary interface logic net out1. Net out1 is considered a boundary net because it is directly connected to the out1 output port.

Figure 9-8 Side-Load Cell and Boundary Side-Load Cell



Controlling the Number of Latch Levels

By default, Design Compiler assumes that all transparent latches found in the interface logic are potential time borrowers. Path tracing continues from the input ports through these latches until an edge-triggered register is encountered. Similarly, path tracing continues from the output registers through the latches to the output ports.

Control the number of latch levels included in the model by using the `-latch_level` option when you run the `create_ilm` command:

```
dc_shell> create_ilm -latch_level 1
```

Table 9-4 ILM Creation Command Option for Controlling Latch Levels

Option	Description
`-latch_level`	Control the number of latch levels included in the model.

If your design does not have any time borrowing across the latches, set the `-latch_level` option to 0. In this case, the first latch encountered in a timing path is treated as an

edge-triggered cell and is considered an endpoint for a timing path from an input port or a startpoint for a timing path to an output port.

Figure 9-9 shows a default ILM and latch levels. Figure 9-10 shows the results of specifying `-latch_level 1` and `-latch_level 0`.

Figure 9-9 Interface Logic Model and Latch Levels

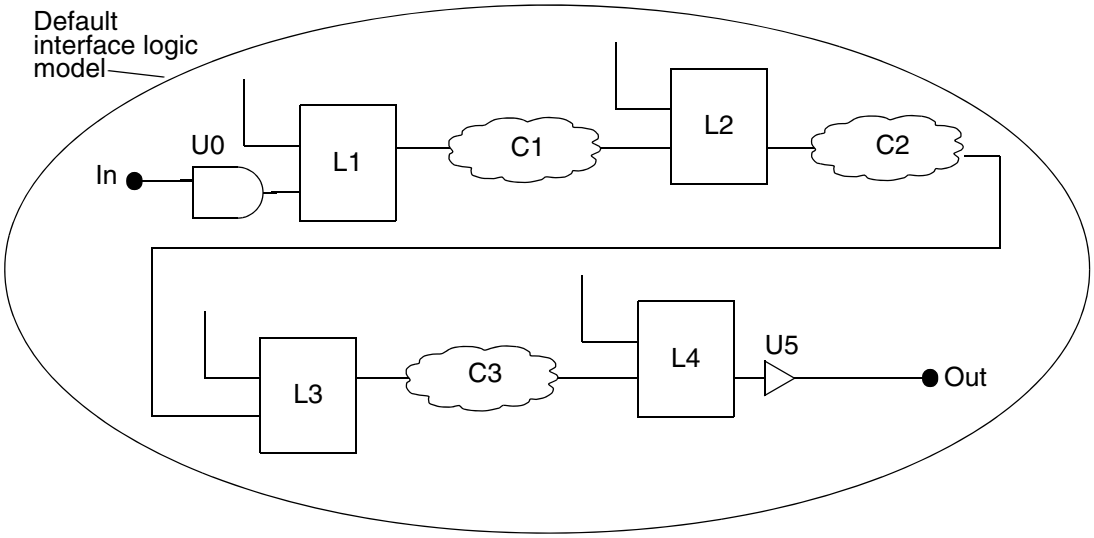
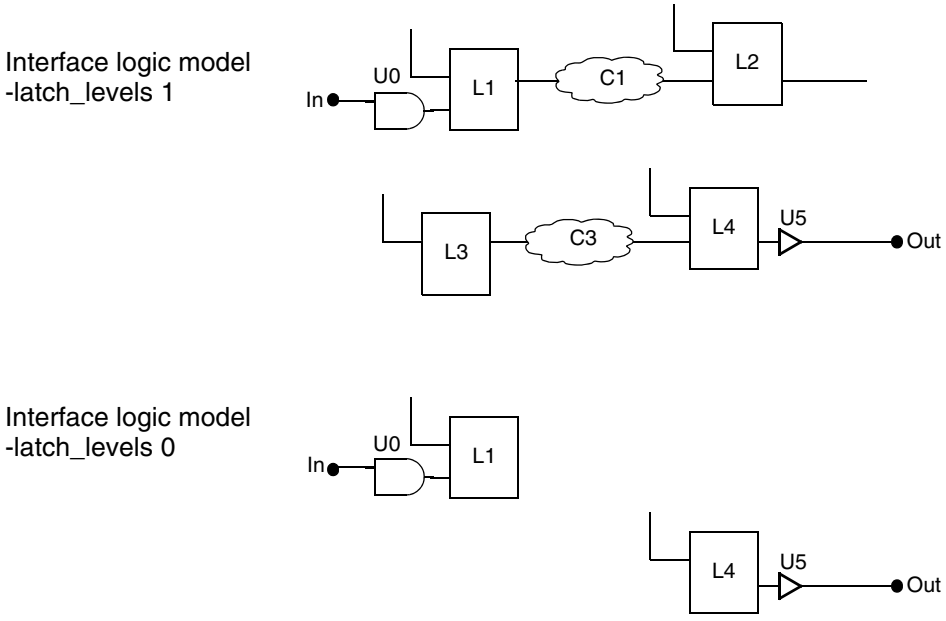


Figure 9-10 Results of Specifying -latch_level 1 and -latch_level 0



Additional Controls for Noncompact ILMs

When creating a noncompact ILM by using the `create_ilm -compact none` command, you can also control the retention of the following logic:

- The fanin and fanout of chip-level networks
- The logic disabled by timing exceptions

Retaining the Fanin and Fanout of Chip-Level Networks

By default, when you generate a noncompact ILM, Design Compiler ignores high-fanout input and inout ports. A port is considered high-fanout if the percentage of the total registers in the design in the transitive fanout of the port is greater than or equal to the value set by the `abstraction_ignore_percentage` variable, which has a default of 25. Typically these ports are associated with a chip-level network, such as a scan-enable, set, or reset network.

If you want more control over which ports are ignored, use the `-no_auto_ignore` option and explicitly specify the ignored ports by using the `-ignore_ports` option.

```
dc_shell> create_ilm -compact none \
   -no_auto_ignore -ignore_ports [get_ports scan_enable]
```

Table 9-5 ILM Creation Command Option for Retaining Fanin and Fanout

Option	Description
`-ignore_ports`	Specify the ports to ignore when identifying the interface logic. This option applies only to noncompact ILMs.

The connectivity from all ignored ports, whether automatically ignored or explicitly ignored, to already-identified interface logic is retained. Minimum and maximum paths and connectivity from all ports are also included in the model.

Retaining Logic on Disabled Paths

By default, when you generate a noncompact ILM, Design Compiler removes the interface logic that is disabled by the `set_case_analysis` command that you apply on the block because that logic is usually not needed.

To retain the disabled logic, use the `-traverse_disabled_arcs` option with the `create_ilm -compact none` command.

```
dc_shell> create_ilm -compact none -traverse_disabled_arcs
```

Table 9-6 ILM Creation Command Option for Retaining Logic on Disabled Paths

Option	Description
`-traverse_disabled_arcs`	Retain the interface logic from disabled timing arcs and pins. This option applies only to noncompact ILMs.

Note:
The `create_ilm` command ignores the `-traverse_disabled_arcs` option unless you also specify the `-compact none` option.

If you use the `-traverse_disabled_arcs` option, you must apply the timing exceptions from the top-level design onto the block because constraints are not automatically propagated from the block level to the top level. Not doing so could lead to including additional timing paths that would need to be disabled for timing analysis when the ILMs are used at the top level.

Storing Parasitic Information in an ILM

When you create an ILM in Design Compiler topographical mode, the virtual placement and parasitics information is stored with the ILM by default. The information is automatically propagated to the top-level when using the ILM for top-level synthesis.

Validating ILMs

Validate an ILM against the original gate-level netlist or compare any two valid timing models in Design Compiler by using the following commands:

- `write_interface_timing`
- `compare_interface_timing`

The `write_interface_timing` command generates an ASCII report containing interface timing information for a gate-level netlist or an ILM. The command performs an implicit timing update, if necessary.

Table 9-7 lists the options supported by the `write_interface_timing` command.

Table 9-7 Options Supported by the write_interface_timing Command

Option	Description
file_name	Specifies the name of the file to which the interface timing information is written.
`-ignore_ports`	Specifies a list of ports that are excluded from the timing file. By default, all ports are included.
`-nosplit`	Prevents line-splitting.
`-significant_digits`	Specifies the number of digits to the right of the decimal point that are reported following the method used in the `report_timing` command. Allowed values are 0-13. The default is 3.

The `write_interface_timing` command writes a report on the interface timing of a specified netlist or model. The report contains the following sections:

- Worst Slack: This section contains the slack values of the critical paths either starting from input ports or ending at output ports. For each critical path, it reports the slack values for the following arc types: minimum rise, minimum fall, maximum rise, and maximum fall.
- Type: This section contains maximum or minimum arc values for the critical paths either starting from input ports or ending at output ports. For each path timing relationship, it reports one of the following possible arc types: minimum rise, minimum fall, maximum rise, and maximum fall.

The `compare_interface_timing` command compares two interface timing files called the reference file and the comparison file, previously generated by the `write_interface_timing` command. The result is “pass” if the timing parameter values in the two files are the same or within a specified tolerance. The result is “fail” if both the columns have data and the tolerance is exceeded. If either file format does not conform to the `write_interface_timing` command standard, the command issues a warning message.

Table 9-8 lists the command options supported by the `compare_interface_timing` command.

Table 9-8 Options Supported by the compare_interface_timing Command

Option	Description
`ref_timing_file`	Specifies the name of the timing file used as the reference in the comparison.
`cmp_timing_file`	Specifies the name of the timing file compared in the comparison.
`-output`	Specifies the name of the output file to which the results of the comparison are written.
`-absolute_tolerance`	Specifies an absolute error tolerance for time data. The default is “0.1” timing unit of the current design unit.
`-nosplit`	Prevents line-splitting.
`-significant_digits`	Specifies the number of digits to the right of the decimal point that are reported following the method used in the `report_timing` command. Allowed values are 0-13. The default is 3.

Example 9-3 shows the flow for creating and validating ILMs in a multi-scenario environment. When using multiple scenarios, you must create interface timing reports for all your netlists before you create interface timing reports for your scenario ILMs, as shown in the example.

Example 9-3 Creating and Validating ILMs in a Multi-Scenario Environment

```
#For each scenario, write out the original interface timing before
#creating the ILM
foreach scenario [all_scenarios] {
  current_scenario ${scenario}
  write_interface_timing -nosplit ${scenario}.interface_timing.rpt
}
create_ilm
foreach scenario [all_scenarios] {
  current_scenario ${scenario}

#For each scenario, write out the interface timing after creating the ILM
  write_interface_timing -nosplit ${scenario}.ILM.interface_timing.rpt

#Now verify that the ILM interface timing matches the original design
#using compare_interface_timing
  compare_interface_timing \
           ${scenario}.interface_timing.rpt \
           ${scenario}.ILM.interface_timing.rpt \
```

```
            -output ${scenario}.compare_interface_timing.rpt
}
write -format ddc -hierarchy -output ${design}.ILM.ddc
```

Example 9-4 shows an example of the report generated by the `compare_interface_timing` command.

Example 9-4 compare_interface_timing Report

```
From        To              Type            Worst Slack       Difference     Status
                                           Ref      Model
-------------------------------------------------------------------------------
tdi         INPUTS          max_rise      1.283     1.345     -0.062         PASS
tdi         INPUTS          max_fall      1.364     1.389     -0.035         PASS
tdi         INPUTS          min_rise      0.033     0.032      0.001         PASS
tdi         INPUTS          min_fall      0.026     0.027     -0.001         PASS
OUTPUTS     DataSdram[6]    max_rise      2.081     2.037      0.044         PASS
OUTPUTS     DataSdram[6]    max_fall      2.132     2.031      0.101         FAIL
OUTPUTS     DataSdram[6]    min_rise      1.329     1.328      0.001         PASS
OUTPUTS     DataSdram[6]    min_fall      1.347     1.345      0.002         PASS
```

Debugging compare_interface_timing Failures

If the report generated by the `compare_interface_timing` command contains FAIL compare points, debug the causes of the timing differences.

ILMs reduce the subblock size by retaining only the timing information needed for top-level integration and optimization. By reducing the subblock size, runtime and memory consumption are typically reduced. Some of the logic removed by the default ILM flow can affect the interface timing path loading, but this effect is considered to be minimal. However, it is possible that the missing logic could insert unacceptably large timing differences.

Some examples of these kinds of mismatches are explained in the following subsections. You can detect these mismatches by running the `compare_interface_timing` command.

Missing Input or Output Delay Constraints

Ensure that all the input and output delay constraints have been specified by using the `set_input_delay` and `set_output_delay` commands before creating the ILM.

Ignored Ports

When creating ILMs, use the `create_ilm -ignore_ports` command to remove all logic related to certain ports. Disable the logic only if you use the `create_ilm -no_auto_ignore` command. Although ILMs do not include port logic, ILM verification does not allow automatic removal. You must manually specify those ignored ports when you run the `write_interface_timing` command; otherwise, the `compare_interface_timing` command fails. Make sure that you ignore the same ports when you run the `write_interface_timing` command.

Clock Trees

To reduce the model size, the exact clock-tree topology is not retained when generating the ILM. Missing loads in an ILM clock net can lead to timing differences that are detected by the `compare_interface_timing` command. If the differences are too large to be acceptable, consider generating the ILM by using the `create_ilm -keep_full_clock_tree` command.

Transparent Latches and Time Borrowing

By default, the `create_ilm` command includes all latches that are treated as combinational elements. All paths are included until an output port or edge-triggered register data pin is found.

If you use the `create_ilm -latch_level` command, it is possible to have some timing mismatches with respect to the original design. This is detected when you run the `compare_interface_timing` command. If the mismatch is too large, regenerate the ILM without using the `-latch_level` option.

Side-Load Cells

By default, only side-load cells that are boundary cells are included in an ILM. These are cells on nets that are connected to the ports of the model. Even though these cells are not involved in any interface logic timing paths, they might affect the timing of an interface path.

This mismatch is detected when you run the `compare_interface_timing` command. If the mismatch is too large to be acceptable, regenerate the ILM block by using the `create_ilm -include_side_load` command.

Reporting Information About ILMs

The `report_ilm` command reports information either about ILMs present in the design or about the current ILM. You can use this command from the top-level design in which the ILMs are instantiated or from the current ILM. By using the `report_ilm` command, you can print the following details for each ILM:

- The full path of the library from which the ILMs are loaded, including the name of the ILM and the library
- The date and time when the ILMs were created
- The `create_ilm` command options that were used to create an ILM

  The options include the explicit options that you have set and also the implicit options set or adjusted by the tool
- Leaf cell count and compression statistics of the ILM

- Top-level and ILM multicorner-multimode scenario mapping that you specify
- TLUPlus file settings of an ILM that include minimum and maximum TLUPlus files, minimum and maximum emulation TLUPlus files, and the technology-file-to-ITF mapping file

  For multicorner-multimode designs, the TLUPlus file settings are listed for each scenario. The command also includes TLUPlus signature file information and parasitic data information in the report.

  Note:
  To check the block-level scenario-related information in multicorner-multimode designs, run the `report_ilm` command after creating the scenarios at the top level.
- UPF information such as power domain name, scope, elements, supply connections, isolation strategies, retention strategies, and power switches

When you use the `report_ilm` command to report on the current ILM, the following information is not included in the report:

- ILM instance name
- ILM location and the orientation in the top-level design
- Top cell count and compression statistics
- Top-level and ILM multicorner-multimode scenario mapping that you specify

Example 9-5 shows a sample of the block-level information reported by the `report_ilm` command.

Example 9-5 Block-Level Information Reported by report_ilm

```
##############################
Block Reference: my_ilm_ref
##############################

Options used in ILM creation:
------------------------------------------------------------------------
Option                      Value used*
------------------------------------------------------------------------
verbose                     on
identify_only               off
extract_only                off
include_all_logic           on
no_auto_ignore              on
keep_macros                 off
keep_boundary_cells         off
keep_full_clock_tree        off
keep_parasitics             on
include_xtalk               on
latch_level                 not specified
include_side_load           boundary
```

```
ignore_ports                not specified
compact                     all
traverse_disabled_arcs      off
case_controlled_ports       not specified
must_connect_ports          not specified
scenarios                   not specified
-------------------------------------------------------------------------
* Some of the options may have been automatically set/reset by the tool
Leaf cell count statistics:
---------------------------
 ILM cell count           : 11503
 Block cell count         : 274399
 Compression percentage   : 95.8
TLU+ files used:
----------------
ILM scenario #0: ilm_scenario1 (mapped from top-level scenario
top_scenario1)
 Max TLU+ file            : TLU_Plus/max.tluplus
 Tech2ITF mapping file    : TLU_Plus/mapfile
ILM scenario #1: scenario2 (mapped from top-level scenario scenario2)
 Max TLU+ file            : TLU_Plus/max.tluplus
 Tech2ITF mapping file    : TLU_Plus/mapfile
```

In addition, the following top-level design details are also printed:

- Location and orientation of each ILM instance
- Leaf cell count and compression statistics of the top-level design

Example 9-6 shows an example of the top-level information reported by the `report_ilm` command.

Example 9-6 Top-Level Information Reported by report_ilm

```
-------------------------------------------------------------------------
ILM Instance                  Reference      Orient    Location
-------------------------------------------------------------------------
my_ilm_inst                   my_ilm_ref       0      (43.40, 30.80)
-------------------------------------------------------------------------
Top design leaf cell count statistics:
-------------------------------------
 ILM instance count      : 1
 Top only cell count     : 134256
 Top + ILM cell count    : 453367
 Top + Block cell count  : 7713418
 Compression percentage  : 94.12
Top-level and ILM scenario mapping:
-------------------------------------------------------------------------
Top-level scenario         Reference ILM       ILM scenario
-------------------------------------------------------------------------
top_scenario1                my_ilm_ref      ilm_scenario1
-------------------------------------------------------------------------
(Only user-specified mappings are reported)
```

You can use the commands listed in Table 9-9 to provide additional information about ILMs.

Table 9-9 Commands That Report Information About Interface Logic Models

Command	Reports
`get_ilms`	The ILM blocks that are defined as part of the current design. You can prevent this command from issuing messages by using the `-quiet` option.
`get_ilm_objects`	Objects in the current design that are identified as belonging to interface logic.
`report_area`	Statistics for both the original netlist and the ILM netlist to let you know how much reduction occurred for each of the design objects and the total area reduction for ILM designs.
`report_design`	Information that the design is an ILM.

Using ILMs

Design Compiler ILMs are stored in .ddc files, while IC Compiler ILMs are stored in a Milkyway reference library that is linked to the current Milkyway design library.

Linking ILMs to the Top-Level Design

Use the following guidelines to allow Design Compiler to link to block-level ILMs when performing top-level synthesis:

- If you are using IC Compiler ILMs, add the Milkyway library containing the IC Compiler ILM view for the block to the list of Milkyway reference libraries used for top-level synthesis.
- If you are using Design Compiler ILMs, read in the ILM .ddc files for the blocks before linking the top-level design.
- Use the `list_designs -show_file` command to check the source of the block-level designs in memory.

Design Compiler automatically links in ILMs that are referenced in the top-level design. During the link process, Design Compiler first searches for each reference name in the libraries specified in the `link_library` variable. If references remain unresolved after this search, Design Compiler searches the Milkyway reference libraries for ILM views that match the unresolved reference names.

Note:

If a .db file corresponding to a macro (an extracted timing model) is specified in the `link_library`, the link process chooses that model and does not search for the ILM in the Milkyway library because higher precedence is given to explicitly specified files.

The link process also incorporates automatic propagation of ILM data, such as placement and UPF constraints, to the top-level design.

Applying Top-Level Constraints

You can apply timing constraints to the top-level design by using the `read_sdc` command to read the golden SDC file.

When you apply the golden SDC file to the top-level design with ILMs, you might get errors and warnings because constraints are set on objects that exist in the hierarchical blocks, but not in their ILMs. You must verify each error and warning to ensure that it can safely be ignored. To prevent these errors and warnings, you can remove these constraints from the

golden SDC file to create an ILM-compatible SDC file; however, you must be careful not to eliminate any valid constraints.

During block-level implementation, if you are using timing exceptions or case-analysis constraints, you must set the same timing exceptions and case-analysis constraints at the top level. If the constraints and exceptions are not set, the ILM might not correctly represent the block-level design.

For example, if the case-analysis set at the top level on an ILM boundary pin does not match the case-analysis value set on the corresponding port of the block, Design Compiler reports the ILM-123 error message indicating the mismatch. This error occurs because the ILM might not have the timing paths corresponding to the top-level constraints.

You might also get this error when the ILM boundary pin has a case-analysis value set at the top level, but the corresponding port of the block from which the ILM was created does not have a case-analysis value.

In a compact ILM that is created with the `-case_controlled_ports` option of the `create_ilm` command, you can change the case-analysis setting at the top level, irrespective of whether the value at the block-level is 0, 1, or not defined.

In a noncompact ILM, you can change the case-analysis setting at the top-level if the case-analysis value at the block-level is not defined. However, if the case-setting value at the block level is either 0 or 1, you can change the value at the top level if the ILM is created using either of the following `create_ilm` command options:

- `-case_controlled_ports`
- `-traverse_disabled_arcs`

Using the propagate_constraints Command

The timing and other constraint data is stored in the ILM. You can use the `propagate_constraints` command to

- Propagate block-level operating conditions to the current design in a multivoltage design flow, using the `propagate_constraints -operating_conditions` command.
- Propagate power supply information to the current design in a UPF flow, using the `propagate_constraints -power_supply_data` command.

  Note:

  As the UPF data from the ILM is automatically propagated to the top level, you do not need to propagate it explicitly.
- Propagate case-analysis constraints to the current design, using the `propagate_constraints -case_analysis` command.

The `propagate_constraints` command can propagate other timing constraints as well.

Due to limitations of the `propagate_constraints` command, you might have to apply clocks and constraints from the top level instead of propagating them from the subblocks.

An example of this limitation occurs when you use the `propagate_constraints` command on a design in which two or more instances of ILM blocks share a common clock name, such as when there are multiple instances of an ILM block or when there are different ILM blocks that share a common clock name. The `propagate_constraints` command cannot propagate these same name clocks to the top-level design. In addition, top-level latency information is partially or entirely lost in this situation.

10

Using Design Compiler Topographical Technology

Topographical technology enables you to accurately predict post-layout timing, area, and power during RTL synthesis without the need for timing approximations based on wire load models. It uses Synopsys' placement and optimization technologies to drive accurate timing prediction within synthesis, ensuring better correlation with the final physical design. This technology is built in as part of the DC Ultra feature set and is available only by using the `compile_ultra` command in topographical mode.

Design Compiler topographical mode requires a DC Ultra license and a DesignWare license. A Milkyway-Interface license is also necessary if the Milkyway flow is used; this license is automatically included with the DC Ultra license.

To learn how to use Design Compiler in topographical mode, see

- Power Optimization in Topographical Mode
- Compile Flows in Topographical Mode
- Supported Commands, Command Options, and Variables
- Using the Design Compiler Graphical Tool
- Optimizing Multicorner-Multimode Designs in Design Compiler Graphical

Overview of Topographical Technology

In ultra deep submicron designs, interconnect parasitics have a major effect on path delays; accurate estimates of resistance and capacitance are necessary to calculate path delays. In topographical mode, Design Compiler leverages the Synopsys physical implementation solution to derive the “virtual layout” of the design so that the tool can accurately predict and use real net capacitances instead of statistical net approximations based on wire load models. If wire load models are present, they are ignored.

In addition, the tool updates capacitances as synthesis progresses. That is, it considers the variation of net capacitances in the design by adjusting placement-derived net delays based on an updated virtual layout at multiple points during synthesis. This approach eliminates the need for overconstraining the design or using optimistic wire load models in synthesis. The accurate prediction of net capacitances drives Design Compiler to generate a netlist that is optimized for all design goals including area, timing, test, and power. It also results in a better starting point for physical implementation.

Topographical technology supports all synthesis flows, including the following:

- Test-ready compilation flow (basic scan and DFT MAX adaptive scan)
- Clock-gating flow
- Register retiming

See Also

- About DC Ultra
- Running a Synthesis Flow

Starting Design Compiler Topographical Mode

To use the Design Compiler topographical features, you must run dc_shell in topographical mode. At the prompt, enter

```
dc_shell -topographical
```

In topographical mode, the dc_shell command-line prompt appears as

```
dc_shell-topo>
```

To query for topographical mode, run the `shell_is_in_topographical_mode` command. The command returns 1 if the topographical mode is active; otherwise it returns 0. Topographical technology is Tcl based. When you run the `compile_ultra` command in this mode, the Design Compiler topographical features are automatically used. All `compile_ultra` command options are supported.

Note:

Design Compiler in topographical mode supports only a subset of dc_shell commands, command options, and variables.

For more information, see Supported Commands, Command Options, and Variables.

Inputs and Outputs in Design Compiler Topographical Mode

Figure 10-1 shows the inputs and outputs in Design Compiler topographical mode. The sections that follow provide more information.

Figure 10-1 Inputs and Outputs in Design Compiler Topographical Mode

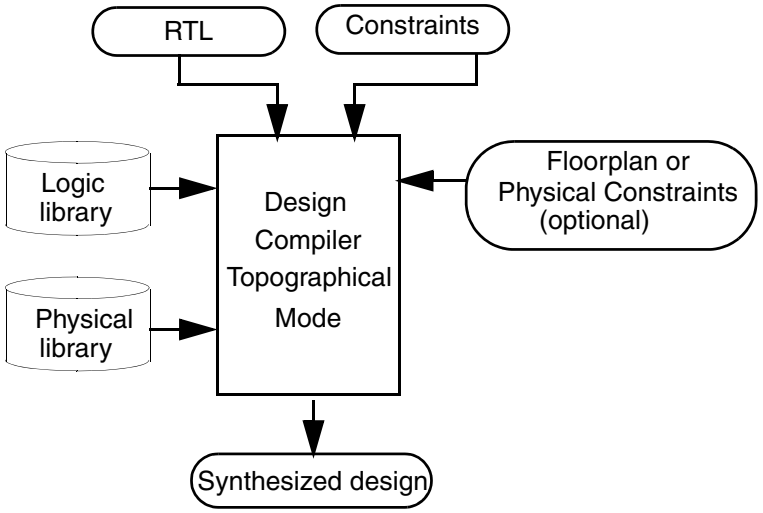


Table 10-1 and Table 10-2 describe the inputs and outputs in topographical mode.

Table 10-1 Inputs in Design Compiler Topographical Mode

Input	Description
Design	RTL or gate-level netlist.
Constraints	Timing and optimization constraints. Note that if you specify wire load models, Design Compiler ignores them.
Logic library	Liberty format (.lib or .db). For more information, see Specifying Logic Libraries.
Physical library	Milkyway format. For more information, see Specifying Physical Libraries. Note: The .pdb format is not supported.

Table 10-1 Inputs in Design Compiler Topographical Mode (Continued)

Input	Description
Floorplan or physical constraints	High-level physical constraints that determine items such as core area and shape, port location, macro location and orientation, voltage areas, placement blockages, and placement bounds. For more information, see Using Floorplan Physical Constraints.

Table 10-2 Outputs in Design Compiler Topographical Mode

Output	Description
.ddc	This format contains back-annotated net delays and constraints. Subsequent topographical mode sessions restore virtual layout data. The .ddc format is recommended for subsequent topographical mode optimizations and verification.
Milkyway	This format contains back-annotated net delays and constraints. The Milkyway format cannot be read back into topographical mode for subsequent optimizations. This format is recommended if you intend to use Synopsys tools for the back-end flow.
ASCII	This format does not contain back-annotated delays or Synopsys Design Constraints (SDC). Use the `write_sdc` command to write out the SDC and the `write_parasitics` command to write out parasitics. This format is recommended only if you intend to use a third-party tool.

Note:

You cannot use the Milkyway format to store design data for unmapped designs or non-uniquified designs. Before you use the `write_milkyway` command, run the following command:

```
uniquify -force -dont_skip_empty_designs
```

Specifying Libraries

In topographical mode, Design Compiler requires both logic libraries and physical libraries. Topographical mode also supports synthesis with black boxes. For more details about black box support, see Support for Black Boxes.

Note:

Occasionally, IC Compiler adds support for new technology file attributes in the physical libraries that are not yet supported in Design Compiler. In these cases, a TFCHK-009

error message is issued in Design Compiler but not in IC Compiler when using the same technology file. If the new attributes are safe to ignore, you can set the `ignore_tf_error` variable to `true`, which enables the tool to ignore the unsupported attributes. For details, see Fixing Errors Caused by New Unsupported Technology File Attributes.

Specifying Logic Libraries

Design Compiler topographical mode uses the same logic libraries as the Design Compiler wire load mode. Use the following commands to set up logic libraries:

```
set_app_var search_path "$search_path ./libraries"
set_app_var link_library "* max_lib.db"
set_app_var symbol_library "max_lib.sdb"
set_app_var target_library "max_lib.db"
```

For more information about logic libraries, see Working With Libraries.

Note:

Because multivoltage designs use power domains, these designs usually require that certain library cells are marked as always-on cells and certain library cell pins are marked as always-on library cell pins. These always-on attributes are necessary to establish any always-on relationships between power domains. For more information, see the *Power Compiler User Guide*.

Specifying Physical Libraries

You use the Milkyway design library to specify physical libraries and save designs in Milkyway format. The inputs required to create a Milkyway design library are the Milkyway reference library and the Milkyway technology file.

The Milkyway reference library contains the physical representation of standard cells and macros. In topographical mode, the Milkyway reference library uses the FRAM abstract view to store information. The reference library also defines the placement unit tile (the width and height of the smallest placeable instance and the routing directions).

The Milkyway technology file (.tf), contains technology-specific information required to route a design. Design Compiler will automatically derive routing layer directions if your Milkyway library file is missing this information. Derived routing layer directions are saved in the .ddc file. You can override the derived routing layer direction by using the `set_preferred_routing_direction` command. To report all routing directions, use the `report_preferred_routing_direction` command.

For information about creating a Milkyway design library, see Using a Milkyway Database and the Milkyway documentation.

Verifying Library Consistency

Consistency between the logic library and the physical library is critical to achieving good results. Before you process your design, ensure that your libraries are consistent by running the `check_library` command.

```
dc_shell-topo> check_library
```

The `check_library` command provides the following capabilities:

- Checks the integrity of individual logical and physical libraries.
- Checks consistency between logic libraries.
- Checks consistency between logic libraries and physical libraries.
- Checks consistency between physical libraries and technology files.

By default, the `check_library` command performs consistency checks between the logic libraries specified in the `link_library` variable and the physical libraries in the current Milkyway design library. You can also explicitly specify logic libraries (by using the `-logic_library_name` option) or Milkyway reference libraries (by using the `-mw_library_name` option). If you explicitly specify libraries, these override the default libraries.

By default, the `check_library` command performs the following consistency checks between the logic library and the physical library:

- Verify that all cells exist in both the logic library and the physical library, including any physical-only cells.
- Verify that the pins on each cell are consistent between the logic library and the physical library.

  This validation includes consistency in naming, direction, and type (including power and ground pins).

  Note:

  If the logic library does not contain `pg_pin` definitions, Design Compiler uses the power and ground pins as defined in the physical library.

You can also perform the following consistency checks, by setting the appropriate options with the `set_check_library_options` command:

- Verify that the area for each cell (except pad cells) is consistent between the logic library and the physical library.

  ```
  dc_shell-topo> set_check_library_options -cell_area
  ```

- Verify that the cell footprints are consistent between the logic library and the physical library.

```
dc_shell-topo> set_check_library_options -cell_footprint
```

- Verify that the same bus naming style is used in the logic library and the physical library.

```
dc_shell-topo> set_check_library_options -bus_delimiter
```

Specify the option for each check that you would like to enable, or to enable all consistency checks, use the `-logic_vs_physical` option.

To reset the library checks to the default settings (cell name and pin consistency checks), run the `set_check_library_options -reset` command.

To see the enabled library consistency library checks, run the `report_check_library_options -logic_vs_physical` command.

If the `check_library` command reports any inconsistencies, you must fix these inconsistencies before you process your design. For more information about logic libraries, see the Library Compiler documentation. For more information about physical libraries, see the Milkyway documentation.

Using TLUPlus Files for RC Estimation

TLUPlus files contain resistance and capacitance look-up tables and model ultra deep submicron (UDSM) process effects. If TLUPlus files are available or are used in your back-end flow, it is recommended that you use them for RC estimation in Design Compiler topographical mode.

TLUPlus files provide more accurate capacitance and resistance data, thereby improving correlation with back-end results.

You use the `set_tlu_plus_files` command to specify TLUPlus files. In addition, use the `-tech2itf_map` option to specify a map file, which maps layer names between the Milkyway technology file and the process Interconnect Technology Format (ITF) file. For example,

```
dc_shell-topo> set_tlu_plus_files -max_tluplus $max_tlu_file \
   -min_tluplus $min_tlu_file -tech2itf_map $prs_map_file
```

You can use the `check_tlu_plus_files` command to check TLUPlus settings.

For more information about the map file, see the Milkyway documentation. To ensure that you are using the TLUPlus files, check the `compile_ultra` log for the following message:

```
****************************************************
Information: TLU Plus based RC computation is enabled.
(RCEX-141)
****************************************************
```

You can use the `extract_rc` command to perform 2.5D extraction. The command calculates delays based on the Elmore delay model and can update back-annotated delay

and capacitance numbers on nets. Use this command after the netlist has been edited. If you used the `set_tlu_plus_files` command to specify the TLUPlus technology files, the tool performs extraction based on TLUPlus technology. Otherwise, the tool performs extraction using the extraction parameters in your physical library. Use the `set_extraction_options` command to specify the parameters that influence extraction and the `report_extraction_options` command to report the parameters that influence the post-route extraction.

Support for Black Boxes

Design Compiler in topographical mode supports synthesis with black boxes. The following types of black boxes are supported:

- Functionally unknown black boxes

  These are cells where the logic functionality is not known. Examples include the following types of cells:

  - Macro cells
  - Empty hierarchy cells or black-boxed modules
  - Unlinked or unresolved cells

- Logical black boxes

  These are cells that do not link to a cell in the logic library. This type of black box is categorized under functionally unknown black box cells. Examples include the following types of cells:

  - Empty hierarchy cells or black-boxed modules
  - Unlinked or unresolved cells

  You can define the timing for logical black box cells by using Design Compiler in wire load mode or topographical mode, as described in Defining Timing in Quick Timing Model Format.

- Physical black boxes

  These are cells that do not have physical representation. Examples include the following types of cells:

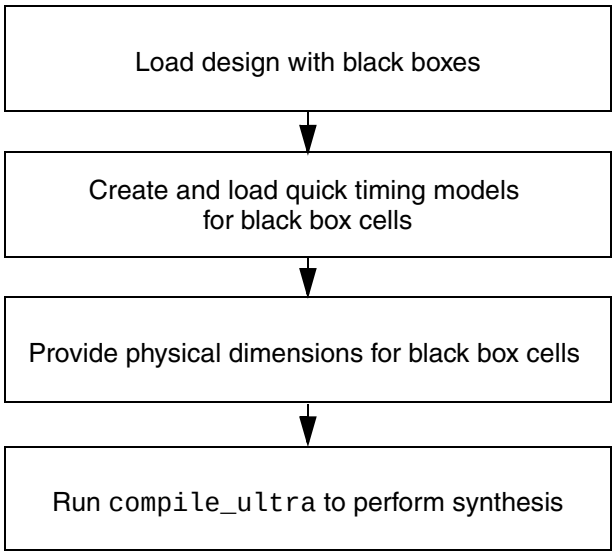
  - Empty hierarchy cells or black-boxed modules
  - Unlinked or unresolved cells

  Topographical technology allows you to define the physical dimensions for physical black box cells, as described in Defining Physical Dimensions.

If the physical representation of a cell is not available with the user-supplied physical libraries, Design Compiler in topographical mode defines it automatically. For more information, see Automatic Creation of Physical Library Cells.

Figure 10-2 shows the flow for defining the timing and physical dimensions for black box cells and synthesizing the design.

Figure 10-2 Supported Black Box Flow



You can visually examine black boxes in your floorplan by viewing them in the Design Vision layout window. You can control the visibility, selection, and display style properties of black-box cells in the active layout view by setting options on the View Settings panel. Black boxes are visible and enabled for selection by default. For more information about controlling the visibility of black boxes in the active layout view, see Design Vision Help.

Defining Timing in Quick Timing Model Format

Design Compiler allows you to define timing for logical black box cells. You can create logic library cells and provide a timing model for the new cells in the quick timing model format. A quick timing model is an approximate timing model that is useful early in the design cycle to describe the rough initial timing of a black box. You create a quick timing model for a black box by specifying the model ports, the setup and hold constraints on the inputs, the clock-to-output path delays, and the input-to-output path delays. You can also specify the loads on input ports and the drive strength of output ports. Design Compiler saves the timing models in .db format, which can then be used during synthesis.

To create a quick timing model for a simple black box in Design Compiler in wire load mode or in topographical mode, perform the following steps:

1. Create a new model using the `create_qtm_model` command:

```
dc_shell-topo> create_qtm_model BB
```

   BB is the model name.

2. Specify the technology information, such as the name of the logic library, the maximum transition time, the maximum capacitance, and the wire load information.

```
dc_shell-topo> set_qtm_technology -library library_name
dc_shell-topo> set_qtm_technology -max_transition trans_value
dc_shell-topo> set_qtm_technology -max_capacitance cap_value
```

3. Specify global parameters, such as setup and hold characteristics.

```
dc_shell-topo> set_qtm_global_parameter -param setup -value
setup_value
dc_shell-topo> set_qtm_global_parameter -param hold -value hold_value
dc_shell-topo> set_qtm_global_parameter -param clk_to_output \
   -value cto_value
```

4. Specify the ports using the `create_qtm_port` command:

```
dc_shell-topo> create_qtm_port -type input A
dc_shell-topo> create_qtm_port -type input B
dc_shell-topo> create_qtm_port -type output OP
```

5. Specify delay arcs using the `create_qtm_delay_arc` command:

```
dc_shell-topo> create_qtm_delay_arc -name A_OP_R -from A \
   -from_edge rise -to OP -value 0.10 -to_edge rise
dc_shell-topo> create_qtm_delay_arc -name A_OP_F -from A \
   -from_edge fall -to OP -value 0.11 -to_edge fall
dc_shell-topo> create_qtm_delay_arc -name B_OP_R -from B \
   -from_edge rise -to OP -value 0.15 -to_edge rise
dc_shell-topo> create_qtm_delay_arc -name B_OP_F -from B \
   -from_edge fall -to OP -value 0.16 -to_edge fall
```

6. (Optional) Generate a report that shows the defined parameters, the ports, and the timing arcs in the quick timing model.

```
dc_shell-topo> report_qtm_model
```

7. Save the quick timing model using the `save_qtm_model` command:

```
dc_shell-topo> save_qtm_model
```

8. Write out the .db file for the quick timing model using the `write_qtm_model` command:

```
dc_shell-topo> write_qtm_model -out_dir QTM
```

9. Add the .db file to the `link_library` variable to load the quick timing model:

```
dc_shell-topo> lappend link_library "QTM/BB.db"
```

Table 10-3 lists the quick timing model commands you can use to define timing for black box cells.

Table 10-3 Commands for Defining Timing in Black Box Cells

To do this:	Use this command:
Create a quick timing model clock	`create_qtm_clock`
Define setup and hold arcs	`create_qtm_constraint_arc`
Create the delay arcs for a quick timing model	`create_qtm_delay_arc`
Create a drive type in a quick timing model description	`create_qtm_drive_type`
Create a generated clock in a quick timing model	`create_qtm_generated_clock`
Create the insertion delay on the clock port for a quick timing model	`create_qtm_insertion_delay`
Create a load type for a quick timing model description	`create_qtm_load_type`
Begin defining a quick timing model	`create_qtm_model`
Create a path type in a quick timing model	`create_qtm_path_type`
Create a quick timing model port	`create_qtm_port`
Report details about the current quick timing model	`report_qtm_model`
Save the quick timing model	`save_qtm_model`
Set a global setup, hold, or clock parameter	`set_qtm_global_parameter`
Set drive on a port	`set_qtm_port_drive`
Set load on a port	`set_qtm_port_load`
Set various technology parameters	`set_qtm_technology`
Write the quick timing model .db file	`write_qtm_model`

Defining Physical Dimensions

Design Compiler allows you to set the size for physical black box cells based on an estimation of the objects that it will contain when replaced with real logic. You can also define the base unit area for gate equivalence calculations for estimating the size of black boxes. The following sections discuss these strategies for defining the physical dimensions of black box cells:

- Estimating the Size of Black Boxes
- Determining the Gate Equivalent Area

Estimating the Size of Black Boxes

To set the size and shape for physical black box cells based on an estimation of the objects that it will contain, use the `estimate_fp_black_boxes` command.

To set the width and height, use the `-sm_size` option:

```
dc_shell-topo> estimate_fp_black_boxes -sm_size size
```

To create a rectilinear black box, use the `-polygon` option:

```
dc_shell-topo> estimate_fp_black_boxes -polygon polygon_area
```

The following example estimates a black box named alu1 and specifies it as a soft macro with a size of 100x100 and a utilization of 0.7:

```
dc_shell-topo> estimate_fp_black_boxes \
  -sm_size {100 100} -sm_util 0.7 \
  [get_cells -filter "is_physical_black_box==true" alu1]
```

The following example estimates a black box named alu1 and specifies it as a rectilinear soft macro:

```
dc_shell-topo> estimate_fp_black_boxes \
  -polygon {{1723.645 1925.365} {1723.645 1722.415} \
  {1803.595 1722.415} {1803.595 1530.535} \
  {799.915 1530.535} {799.915 1925.365} \
  {1723.645 1925.365}}\
  [get_cells -filter "is_physical_black_box==true" alu1]
```

In the following example, the size of the black box named U1 is estimated from the size of hard macro ram16x128:

```
dc_shell-topo> estimate_fp_black_boxes -hard_macros ram16x128 U1
```

Determining the Gate Equivalent Area

Design Compiler also allows you to define the base unit area for gate equivalence calculations for estimating the size of black boxes. Use the `set_fp_base_gate` command to

specify either a library leaf cell area or a user-specified cell area as the base unit area to be used for gate equivalence calculations.

To specify a gate from the library as the base unit area to be used for the calculations, use the `-cell` option:

```
dc_shell-topo> set_fp_base_gate -cell UNIT
```

In this example, UNIT specifies the reference name of the library leaf cell to be used as the base unit area for gate equivalence calculations.

To specify the cell area in square microns as the base unit area to be used for the calculations, use the `-area` option:

```
dc_shell-topo> set_fp_base_gate -area 10
```

In this example, the base gate area is set to 10 square microns.

Identifying Black Box Cells

Design Compiler sets the following attributes on black box cells:

- `is_black_box`

  When you specify the `is_black_box` attribute with the `get_cells` command, Design Compiler identifies functionally unknown black boxes:

  ```
  dc_shell-topo> get_cells -hier -filter "is_black_box==true"
  ```

- `is_logical_black_box`

  When you specify the `is_logical_black_box` attribute with the `get_cells` command, Design Compiler identifies the logical black boxes in the design:

  ```
  dc_shell-topo> get_cells -hier -filter "is_logical_black_box==true"
  ```

- `is_physical_black_box`

  When you specify the `is_physical_black_box` attribute with the `get_cells` command, Design Compiler identifies the physical black boxes in the design:

  ```
  dc_shell-topo> get_cells -hier -filter "is_physical_black_box==true"
  ```

Automatic Creation of Physical Library Cells

If the physical representation of a cell is not available with the user-supplied physical libraries, Design Compiler defines it automatically. The tool can create physical library cells for the following:

- Logic library cells (leaf cells and macros)
- Empty hierarchy cells or black-boxed modules
- Unlinked or unresolved cells
- Unmapped cells

The tool issues the following warning message when it creates physical library cells:

```
Warning: Created physical library cell for logical library
%s. (OPT-1413)
```

Using Floorplan Physical Constraints

To create, import, reset, save, and report physical constraints, see

- Physical Constraints Overview
- Extracting Physical Constraints From IC Compiler Using the write_def Command
- Importing Physical Constraints From an IC Compiler DEF Formatted File
- Exporting Physical Constraints From IC Compiler Using the write_floorplan Command
- Importing Physical Constraints From an IC Compiler write_floorplan Formatted File
- Manually Defining Physical Constraints
- Including Physical-Only Cells
- Specifying Relative Placement
- Placement Options: Magnet Placement
- Resetting Physical Constraints
- Saving Physical Constraints Using the write_floorplan Command
- Reporting Physical Constraints

Physical Constraints Overview

The principal reason for using floorplan constraints in topographical mode is to improve timing correlation with the post-place-and-route tools, such as IC Compiler, by considering floorplanning information during optimizations. Design Compiler topographical mode supports high-level physical constraints such as die area, core area and shape, port locations, cell locations and orientations, keepout margins, placement blockages, preroutes, bounds, vias, tracks, voltage areas, and wiring keepouts.

You can examine most of these objects visually in your floorplan by using the Design Vision layout window. For more information about using the GUI to view physical constraints, see the “Viewing the Floorplan” topic in Design Vision Help.

You provide physical information to Design Compiler topographical mode in the following ways:

- You can export the physical data from IC Compiler by using the `write_def` command within IC Compiler and can import this data into Design Compiler by using the `extract_physical_constraints` command.

  For details, see

  - Extracting Physical Constraints From IC Compiler Using the write_def Command
  - Importing Physical Constraints From an IC Compiler DEF Formatted File

- You can export the physical data from IC Compiler by using the `write_floorplan` command within IC Compiler and import this data into Design Compiler by using the `read_floorplan` command.

  For details, see

  - Exporting Physical Constraints From IC Compiler Using the write_floorplan Command
  - Importing Physical Constraints From an IC Compiler write_floorplan Formatted File

- You can create physical constraints manually.

  For details, see Manually Defining Physical Constraints.

If you do not provide any physical constraints from a floorplanning tool, Design Compiler uses the following default physical constraints:

- Aspect ratio of 1.0 (that is, a square placement area)
- Utilization of 0.6 (that is, forty percent of empty space in the core area)

Extracting Physical Constraints From IC Compiler Using the write_def Command

To improve timing, area, and power correlation between Design Compiler and IC Compiler, you can read your mapped Design Compiler netlist into IC Compiler, create a basic floorplan in IC Compiler, export this floorplan from IC Compiler, and read the floorplan back into Design Compiler.

To export floorplan information from IC Compiler for use in Design Compiler topographical mode, you can use the `write_def` command in IC Compiler. The `write_def` command exports physical design data and writes this data to a Design Exchange Format (DEF) file which you can read into Design Compiler topographical mode. Example 10-1 uses the `write_def` command to write physical design data to the my_physical_data.def file.

Example 10-1 Using the write_def Command to Extract Physical Data From IC Compiler

```
icc_shell> write_def -version 5.7 -rows_tracks_gcells -macro -pins \
      -blockages -specialnets -vias -regions_groups -verbose \
      -output my_physical_data.def
```

Importing Physical Constraints From an IC Compiler DEF Formatted File

To learn about the imported DEF physical constraints, incremental DEF extraction, and port and macro name matching considerations, see

- Importing a DEF Floorplan Overview
- Imported DEF Constraints
- Extracting Physical-Only Cells From a DEF File
- Importing Incremental DEF Information Using the extract_physical_constraints Command
- Matching Names of Macros and Ports

Importing a DEF Floorplan Overview

To import floorplan information from a DEF file, use the `extract_physical_constraints` command. This command imports physical information from the specified DEF file and applies these constraints to the design. The applied constraints are saved in the .ddc file and do not need to be reapplied in a new topographical mode session when you read in the .ddc.

The following command shows how you can import physical constraints from multiple DEF files:

```
dc_shell-topo> extract_physical_constraints \
```

```
                                {des_1.def des2.def … des_N.def}
```

For information on incremental extraction from DEF files, see Importing Incremental DEF Information Using the extract_physical_constraints Command.

Note:

When you use the `extract_physical_constraints` command to read a DEF file, Design Compiler in topographical mode automatically resolves the site name in the floorplan to match the name of the tile in the Milkyway reference libraries. Milkyway reference libraries usually have the site name set to unit by default. However, you might still need to use the `mw_site_name_mapping` variable to define the name mappings if the site dimension does not match unit or if there is more than one site type being used in the DEF. For example,

```
dc_shell-topo> set mw_site_name_mapping { {def_site_name1 mw_ref_lib_site_name} \
               {def_site_name2 mw_ref_lib_site_name} }
```

Multiple pairs of values can be specified for the `mw_site_name_mapping` variable.

Imported DEF Constraints

To learn about the physical constraints that are imported from the DEF file with the `extract_physical_constraints` command, see

- Die Area
- Placement Area
- Macro Location and Orientation
- Hard, Soft, and Partial Placement Blockages
- Wiring Keepouts
- Placement Bounds
- Port Location
- Preroutes
- Site Array Information
- Vias
- Routing Tracks
- Keepout Margins

To visually inspect your extracted physical constraints, use the layout view in the Design Vision layout window. All physical constraints extracted from the DEF file are automatically added to the layout view.

Note:

Voltage areas are not defined in a DEF file. Therefore, for multivoltage designs, you have to use the `create_voltage_area` command to define voltage areas for the tool. If you are using IC Compiler as your floorplanning tool, you can use the `write_floorplan` and `read_floorplan` commands to obtain floorplan information that automatically includes voltage areas. You do not need to define voltage areas manually when using these commands.

Die Area

The Design Compiler topographical mode command, `extract_physical_constraints`, supports automatic die area extraction from the DEF file. DEF files are generated by floorplanning tools and used by Design Compiler to import physical constraints. The tool can extract both rectangular and rectilinear die areas that are defined in the DEF file. The die area is also known as the cell boundary. The die area represents the silicon boundary of a chip and encloses all objects of a design, such as pads, I/O pins, and cells.

The following example shows a die area definition in a DEF file:

```
DEF
   UNITS DISTANCE MICRONS 1000 ;
   DIEAREA ( 0 0 ) ( 0 60000 ) ( 39680 60000 ) ( 39680 40000 ) \
           ( 59360 40000 ) ( 59360 0 ) ;
```

The following example shows how Design Compiler in topographical mode translates the DEF definition into Tcl when you write out your physical constraints using the `write_floorplan -all` command:

```
create_die_area -polygon { { 0.000 0.000 } { 0.000 60.000 } \
       { 39.680 60.000 } { 39.680 40.000 } { 59.360 40.000 } \
       { 59.360 0.000 } { 0.000 0.000 } }
```

Placement Area

Placement area is computed as the rectangular bounding box of the site rows.

Macro Location and Orientation

When you use the `extract_physical_constraints` command, for each cell with a location and the FIXED attribute specified in the DEF, Design Compiler sets the location on the corresponding cell in the design. Example 10-2 shows DEF macro location and orientation information.

Note:

E = east rotation and W = west rotation

Example 10-2 DEF Macro Location and Orientation Information

```
COMPONENTS 2 ;
    - macro_cell_abx2  + FIXED ( 4350720 8160 ) E ;
    - macro_cell_cdy1 + FIXED ( 4800 8160 ) W ;
END COMPONENTS
```

The Tcl equivalent commands are shown in Example 10-3.

Example 10-3 Tcl Equivalent Macro Location and Orientation Information

```
set_cell_location macro_cell_abx2 -coordinates { 4350.720 8.160 } \
        -orientation E -fixed
set_cell_location macro_cell_cdy1 -coordinates { 4.800 8.160 } \
        -orientation W -fixed
```

Hard, Soft, and Partial Placement Blockages

The `extract_physical_constraints` command can import hard, soft, and partial placement blockages defined in the DEF file.

Example 10-4 shows DEF hard placement blockage information.

Example 10-4 DEF Hard Placement Blockage Information

```
BLOCKAGES 50 ;
...
    - PLACEMENT RECT ( 970460 7500 ) ( 3247660 129940 )
...
END BLOCKAGES
```

The Tcl equivalent command is shown in Example 10-5.

Example 10-5 Tcl Equivalent Hard Placement Blockage Information

```
create_placement_blockage -name def_obstruction_23 \
                      -bbox { 970.460 7.500 3247.660 129.940 }
```

For a soft placement blockage, if your extracted DEF information is as shown in Example 10-6 (DEF version 5.6) or Example 10-7 (DEF version 5.7), then the Tcl equivalent command is shown in Example 10-8.

Note:

If your floorplanning tool creates a DEF file with DEF version 5.6, you need to manually add the `#SNPS_SOFT_BLOCKAGE` pragma to specify a soft blockage, as shown in Example 10-6.

Example 10-6 DEF Version 5.6 Soft Placement Blockage Information

```
BLOCKAGES 50 ;
...
- PLACEMENT RECT ( 970460 7500 ) ( 3247660 129940 ) ; #SNPS_SOFT_BLOCKAGE
...
END BLOCKAGES
```

Example 10-7 DEF Version 5.7 Soft Placement Blockage Information

```
BLOCKAGES 50 ;
...
- PLACEMENT + SOFT RECT ( 970460 7500 ) ( 3247660 129940 ) ;
...
END BLOCKAGES
```

Example 10-8 Tcl Equivalent Soft Placement Blockage Information

```
create_placement_blockage -name def_obstruction_23 \
                      -bbox { 970.460 7.500 3247.660 129.940 } \
                      -type soft
```

Example 10-9 shows DEF partial placement blockage information. The Tcl equivalent command is shown in Example 10-10.

Note:
DEF versions prior to version 5.7 did not support partial blockages.

Example 10-9 DEF Partial Placement Blockage Information

```
BLOCKAGES 50 ;
...
- PLACEMENT + PARTIAL 80 RECT ( 970460 7500 ) ( 3247660 129940 ) ;
...
END BLOCKAGES
```

Example 10-10 Tcl Equivalent Partial Placement Blockage Information

```
create_placement_blockage -name def_obstruction_23 \
                      -bbox { 970.460 7.500 3247.660 129.940 } \
                      -type partial \
                      -blocked_percentage 80
```

Wiring Keepouts

For wiring keepouts defined in the DEF, Design Compiler creates wiring keepouts on the design.

Example 10-11 shows DEF wiring keepout information.

Example 10-11 DEF Wiring Keepout Information

```
BLOCKAGES 30 ;
...
   - LAYER METAL6 RECT ( 0 495720 ) ( 4050 1419120 );
...
END BLOCKAGES
```

Placement Bounds

If REGIONS defining bounds exist in the DEF, `extract_physical_constraints` imports placement bounds. Also, if there are cells in the related GROUP attached to the region,

these cells are fuzzy matched with the ones in the design, and the matched cells are attached to the bounds in the following two ways:

- If there are regions in the design with the same name as in the DEF, the cells in the related group are attached to the region by the `update_bounds` command in incremental mode.
- If the region does not exist in the design, it is created with the same name as in the DEF file by applying the `create_bounds` command; matched cells in the related group are also attached.

Example 10-12 shows imported placement bounds information.

Example 10-12 DEF Placement Bounds Information

```
REGIONS 1 ;
 - c20_group ( 201970 40040 ) ( 237914 75984 ) + TYPE FENCE ;
END REGIONS

GROUPS 1 ;
 - c20_group
     cell_abc1
     cell_sm1
     cell_sm2
    + SOFT
    + REGION c20_group ;
END GROUPS
```

The Tcl equivalent commands are shown in Example 10-13.

Example 10-13 Tcl Equivalent Placement Bounds Information

```
create_bounds \
  -name "c20_group" \
  -coordinate {201970 40040 237914 75984} \
  -exclusive \
  {cell_abc1 cell_sm1 cell_sm2}
```

Port Location

When you use the `extract_physical_constraints` command, for each port with the location specified in the DEF, Design Compiler sets the location on the corresponding port in the design.

Example 10-14 shows imported port location information.

Example 10-14 DEF Port Information

```
PINS 2 ;
   -Out1 + NET Out1 + DIRECTION OUTPUT + USE SIGNAL +
      LAYER M3 (0 0) (4200 200) + PLACED (80875 0) N;

   -Sel0 + NET Sel0 + DIRECTION INPUT + USE SIGNAL +
      LAYER M4 (0 0)(200 200) + PLACED (135920 42475) N;
END PINS
```

The Tcl equivalent commands are shown in Example 10-15.

Example 10-15 Tcl Equivalent Port Information

```
set_port_location Out1 -coordinate {80.875 0.000} \
         -layer_name M3 -layer_area {0.000 0.000 4.200 0.200}

set_port_location Sel0 -coordinate {135.920 0.000} \
         -layer_name M4 -layer_area {0.000 0.000 0.200 0.200}
```

Ports with changed names and multiple layers are supported. Example 10-16 shows DEF information for such a case.

Example 10-16 DEF Port Information

```
PINS 2 ;
   - sys_addr\[23\].extra2 + NET sys_addr[23] + DIRECTION INPUT + USE SIGNAL +
      LAYER METAL4 ( 0 0 ) ( 820 5820 ) + FIXED ( 1587825 2744180 ) N ;
   - sys_addr[23] + NET sys_addr[23] + DIRECTION INPUT + USE SIGNAL + LAYER
      METAL3 ( 0 0 ) ( 820 5820 ) + FIXED ( 1587825 2744180 ) N ;
END PINS
```

The corresponding Tcl commands are shown in Example 10-17.

Example 10-17 Tcl Equivalent Port Information

```
set_port_location sys_addr[23] -coordinate { 1587.825 2744.180 } \
 -layer_name METAL3 -layer_area {0.000 0.000 0.820 5.820}
set_port_location sys_addr[23] -coordinate { 1587.825 2744.180 }\
 -layer_name \
METAL4 -layer_area {0.000 0.000 0.820 5.820} -append
```

Port orientation is also supported. Example 10-18 shows DEF information for such a case.

Example 10-18 DEF Port Information

```
PINS 1;
   - OUT + NET OUT + DIRECTION INPUT + USE SIGNAL
      + LAYER m4 ( -120 0 ) ( 120 240 )
      + FIXED ( 4557120 1726080 ) S ;
END PINS
```

The corresponding Tcl commands are shown in Example 10-19.

Example 10-19 Tcl Equivalent Port Information

```
set_port_location OUT -coordinate { 4557.120 1726.080 } -layer_name m4 \
                     -layer_area {-0.120 -0.240 0.120 0.000}
```

Preroutes

Design Compiler extracts preroutes that are defined in the DEF file.

Example 10-20 shows imported preroute information.

Example 10-20 DEF Preroute Information

```
SPECIALNETS 2 ;
- vdd
 + ROUTED METAL3 10000 + SHAPE STRIPE ( 10000 150000 ) ( 50000 * )
 + USE POWER ;
...
END SPECIALNETS
```

The Tcl equivalent commands are shown in Example 10-21.

Example 10-21 Tcl Equivalent Preroute Information

```
create_net_shape -no_snap -type path -net vdd -datatype 0 -path_type 0 \
         -route_type pg_strap -layer METAL3 -width 10.000 \
         -points {{10.000 150.000} {50.000 150.000}}
```

Site Array Information

Design Compiler imports site array information that is defined in the DEF file. Site arrays in the DEF file define the placement area.

Example 10-22 shows imported site array information.

Example 10-22 DEF Site Array Information

```
ROW ROW_0 core 0 0 N DO 838 BY 1 STEP 560 0;
```

The Tcl equivalent commands are shown in Example 10-23.

Example 10-23 Tcl Equivalent Site Array Information

```
create_site_row -name ROW_0 -coordinate {0.000 0.000}\
               -kind core -orient 0 -dir H -space 0.560 -count 838
```

Vias

The `extract_physical_constraints` command extracts vias that are defined in the DEF file. Vias are stored in the .ddc file in the same way as other physical constraints.

Example 10-24 shows how Design Compiler in topographical mode translates the DEF definition into Tcl when you write out your physical constraints using the `write_floorplan -all` command.

Example 10-24 Tcl Equivalent Via Information

```
create_via -no_snap -type via -net VDD -master VIA67 -route_type pg_strap \
      -at {746.61 2279} -orient N
```

```
create_via -no_snap -type via_array -net VDD -master FATVIA45 -route_type pg_strap \
      -at {1491.79 2127.8} -orient N -col 5 -row 2 -x_pitch 0.23 -y_pitch 0.23
```

See Also

- Creating Vias

Routing Tracks

The `extract_physical_constraints` command extracts any track information that is defined in the DEF file. Tracks define the routing grid for standard cell-based designs. They can be used during routing, and track support can enhance congestion evaluation and reporting in Design Compiler in topographical mode to make congestion routing more precise and match more closely with IC Compiler. Track information is stored in the .ddc file in the same way as other physical constraints. If you have a floorplan with track information, such as track type and location, the track information is passed to IC Compiler during floorplan exploration.

The following example shows track data in a DEF file:

```
TRACKS X 330 DO 457 STEP 660 LAYER METAL1 ;
TRACKS Y 280 DO 540 STEP 560 LAYER METAL1 ;
```

Example 10-25 shows how Design Compiler in topographical mode translates the DEF definition into Tcl when you write out your physical constraints using the `write_floorplan -all` command.

Example 10-25 Tcl Equivalent Track Information

```
create_track \
   -layer metal1 \
   -dir X \
   -coord 0.100 \
   -space 0.200 \
   -count 11577 \
   -bounding_box {{0.000 0.000} {2315.400 2315.200}}
create_track \
   -layer metal1 \
   -dir Y \
   -coord 0.200 \
   -space 0.200 \
   -count 11575 \
   -bounding_box {{0.000 0.000} {2315.400 2315.200}}
```

See Also

- Creating Routing Tracks

Keepout Margins

The `extract_physical_constraints` command extracts keepout margins that are defined in the DEF file. Keepout margins are stored in the .ddc file in the same way as other physical constraints.

The following example shows a keepout margin definition in a DEF file:

```
COMPONENTS 2 ;
   - U542 OAI21XL + FIXED ( 80000 80000 ) FN + HALO 10000 10000 50000 50000 ;
   - U543 OAI21XL + FIXED ( 10000 20000 ) FN + HALO SOFT 15000 15000 15000 15000 ;
END COMPONENTS
```

See Also

- Creating Keepout Margins

Extracting Physical-Only Cells From a DEF File

To extract physical-only cells from a DEF file, use the `extract_physical_constraints` command with the `-allow_physical_cells` option.

See Also

- Including Physical-Only Cells

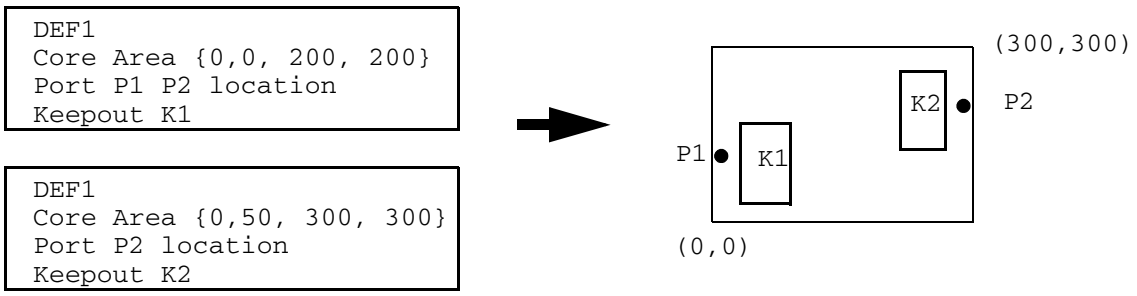
Importing Incremental DEF Information Using the extract_physical_constraints Command

By default, the `extract_physical_constraints` command runs in incremental mode. That is, if you use the command to process multiple DEF files, the command preserves existing physical annotations on the design. In incremental mode, the placement area is imported based on the current core area and site rows in the DEF whereas, in non-incremental mode, the placement area is imported based on the site rows in the DEF. Conflicts are resolved as follows:

- Physical constraints that can have only one value are overwritten by the value from the latest DEF file. That is, port location and macro location are overwritten.
- Physical constraints that can have accumulated values are recomputed. That is, core area can be recomputed based on the existing value and the site row definitions in the latest DEF file. Placement keepouts from different DEF files are accumulated and the final keepout geometry is computed internally during synthesis.

Figure 10-3 shows an example of incremental extraction performed by the `extract_physical_constraints` command.

Figure 10-3 Incremental Extraction with the extract_physical_constraints command




```
extract_physical_constraints DEF1.def DEF2.def
```

To disable incremental mode, use the `-no_incremental` option of the `extract_physical_constraints` command.

Matching Names of Macros and Ports

By default, when the `extract_physical_constraints` command applies physical constraints in topographical mode, it uses an intelligent name matching algorithm to match macros and ports in the DEF file with macros and ports in memory. The command uses the intelligent name matching capability when it does not find an exact match.

The `extract_physical_constraints` command reads the DEF files generated from a netlist that could have different object names than the netlist in memory. These name mismatches can be caused by automatic ungrouping and the `change_names` command. Typically, hierarchy separators and bus notations are sources of these mismatches.

For example, automatic ungrouping by the `compile_ultra` command followed by the `change_names` command might result in the forward slash (/) separator being replaced with an underscore (_) character. Therefore, a macro named *a/b/c/macro_name* in the RTL might be named *a/b_c_macro_name* in the mapped netlist, which is the input to the back-end tool. When extracting physical constraints from the DEF file, the `extract_physical_constraints` command automatically resolves these name differences by using an intelligent name matching algorithm.

To disable intelligent name matching, you can use the `-exact` option of the `extract_physical_constraints` command. This option allows you to specify that the objects in the netlist in memory be matched exactly with the corresponding objects in the DEF file.

When you use the `-verbose` option of the `extract_physical_constraints` command, the tool displays an informational message:

```
Information: Fuzzy match cell %s in netlist with instance
%s in DEF.
```

By default, the following characters are considered equivalent:

- Hierarchical separators { / _ . }

  For example, a cell named *a.b_c/d_e* is automatically matched with the string *a/b_c.d/e* in the DEF file.

- Bus notations { [ ] __ ( ) }

  For example, a cell named *a [4] [5]* is automatically matched with the string *a_4__5_* in the DEF file.

To define the rules used by the intelligent name matching algorithm, use the `set_fuzzy_query_options` command.

For more information, see *Using Tcl With Synopsys Tools*.

Exporting Physical Constraints From IC Compiler Using the write_floorplan Command

To improve timing, area, and power correlation between Design Compiler and IC Compiler, you can read your mapped Design Compiler netlist into IC Compiler, create a basic floorplan in IC Compiler, export this floorplan from IC Compiler, and read the floorplan back into Design Compiler.

To export floorplan information from IC Compiler for use in Design Compiler topographical mode, you can use the `write_floorplan` command in IC Compiler. The `write_floorplan` command writes a Tcl script that you read into Design Compiler to re-create elements of the floorplan of the specified design.

Example 10-26 uses the `write_floorplan` command to write out all placed standard cells to a file called placed_std.fp.

Example 10-26 Using write_floorplan to Export Physical Data From IC Compiler

```
icc_shell> write_floorplan -placement {io terminal hard_macro soft_macro}\
                           -create_terminal -row -create_bound \
                           -preroute -track floorplan_for_DC.fp
```

Importing Physical Constraints From an IC Compiler write_floorplan Formatted File

The `write_floorplan` command generates a Tcl script that contains commands that describe floorplan information for the current design or a design that you specify. You can use this file to re-create elements of the floorplan. Use the `read_floorplan` command to read the script generated by the `write_floorplan` command in IC Compiler and restore the floorplan information.

You can also use the `source` command to import the floorplan information. However, the `source` command reports errors and warnings that are not applicable to Design Compiler in topographical mode. The `read_floorplan` command removes these unnecessary errors and warnings. In addition, you need to enable fuzzy name matching manually when you use the `source` command. The `read_floorplan` command automatically enables fuzzy name matching.

The imported physical constraints are automatically saved to your .ddc file. To save the physical constraints in a separate file, use the `write_floorplan` command after extraction. The command saves the floorplan information so you can read the floorplan back into Design Compiler.

To visually inspect your imported physical constraints, use the layout view in the Design Vision layout window.

Physical Constraints Imported From the Floorplan File

The following physical constraints are imported from the floorplan file:

- Voltage areas
- Die Area

  The die area represents the silicon boundary of a chip and encloses all objects of a design, such as pads, I/O pins, and cells.
- Placement Area
- Macro Location and Orientation

  For each cell with a location and the FIXED attribute specified, Design Compiler sets the location on the corresponding cell in the design.
- Hard and Soft Placement Blockages

  For defined placement blockages, Design Compiler creates placement blockages on the design.
- Wiring Keepouts

Wiring keepout information is imported from the floorplan file. The `create_route_guide` command creates a wiring keepout.

- Placement Bounds

  Placement bounds are extracted from the floorplan file in the following two ways:

  1. If there are regions in the design with the same name as in the floorplan file, the cells in the related group are attached to the region by the `update_bounds` command in incremental mode.
  2. If the region does not exist in the design, it is created with the same name as in the floorplan file by applying the `create_bounds` command. Matched cells in the related group are also attached.

- Port Locations

  For each port with the location specified in the floorplan file, Design Compiler sets the location on the corresponding port in the design.

  Ports with changed names and multilayers are supported.

- Preroutes

  Design Compiler imports preroutes that are defined in the floorplan file. A preroute is represented with the `create_net_shape` command.

- User Shapes

  Design Compiler imports user shapes that are defined in the floorplan file and includes them in the preroute section.

- Site Array Information

  Design Compiler extracts site array information that is defined in the floorplan file. Site arrays define the placement area.

- Vias

  Design Compiler imports vias that are defined in the floorplan file and includes them in the preroute section.

- Tracks

  Design Compiler imports any track information that is defined in the floorplan file.

For information about floorplan modification issues and port and macro name matching considerations, see

- Incremental Floorplan Modifications Using the read_floorplan Command
- Matching Names of Macros and Ports

Incremental Floorplan Modifications Using the read_floorplan Command

The `read_floorplan` command should not be used to perform incremental floorplan modifications. The `read_floorplan` command imports the entire floorplan information exported from IC Compiler with the `write_floorplan` command and overwrites any existing floorplan. The `write_floorplan` command usually includes commands to remove any existing floorplan.

Matching Names of Macros and Ports

By default, when the `read_floorplan` command applies physical constraints in topographical mode, it has an intelligent name matching capability that matches macros and ports in the floorplan file with macros and ports in memory. The command uses the intelligent name matching capability when it does not find an exact match.

The `read_floorplan` command reads from floorplan files generated from a netlist that could have different object names from the netlist in memory. These name mismatches can be caused by automatic ungrouping and the `change_names` command. Typically, hierarchy separators and bus notations are sources of these mismatches.

For example, automatic ungrouping by the `compile_ultra` command followed by `change_names` might result in the forward slash (/) separator being replaced with an underscore (_) character. Therefore, a macro named *a/b/c/macro_name* in the RTL might be named *a/b_c_macro_name* in the mapped netlist, which is the input to the back-end tool. When extracting physical constraints from the DEF file, the `extract_physical_constraints` command automatically resolves these name differences by using Design Compiler's intelligent name matching capability.

By default, the following characters are considered equivalent:

- Hierarchical separators { / _ . }

  For example, a cell named *a.b_c/d_e* is automatically matched with the string *a/b_c.d/e* in the floorplan file.

- Bus notations { [ ] __ ( ) }

  For example, a cell named *a [4] [5]* is automatically matched with the string *a_4__5_* in the floorplan file.

To define the rules used by the intelligent name matching capability, use the `set_fuzzy_query_options` command.

Note:

Using the `source` command is not recommended. However, if you use the `source` command to import your floorplan information, you need to enable fuzzy name matching

manually by setting the `fuzzy_matching_enabled` variable to `true` before you source the floorplan file, for example,

```
(set_fuzzy_query_options)
set fuzzy_matching_enabled true
source design.fp
set fuzzy_matching_enabled false
```

For more information, see *Using Tcl With Synopsys Tools*.

Reading and Writing Preroute Information for Power and Ground Nets and Physical-Only Cells

The floorplan that is written from IC Compiler using the `write_floorplan` command can include preroute information and the location of physical-only cells. However, when the floorplan file is read into Design Compiler in topographical mode, the information is not used during optimization because the power and ground nets and physical cells do not exist in the logical netlist.

To use preroute information and physical-only cells in Design Compiler, you need to create the power and ground nets and physical-only cells before reading the floorplan file. When the preroute information and the location of the physical-only cells are written out from Design Compiler by the `write_floorplan -preroute` command, the command generates a secondary floorplan. The secondary floorplan file contains Tcl commands that create the power and ground nets and physical-only cells. The secondary floorplan file is generated with an .objects suffix added to the floorplan file name.

The following example creates the floorplan files, floorplan.tcl and floorplan.tcl.objects:

```
dc_shell-topo> write_floorplan -preroute floorplan.tcl
1
```

The following example shows a floorplan.tcl.objects file:

```
if {! [sizeof_collection [get_nets -quiet -all "VDD"]]} {
create_net -power VDD
}
if {! [sizeof_collection [get_nets -quiet -all "VSS"]]} {
create_net -ground VSS
}
if {! [sizeof_collection [get_cells -quiet -all "FIL_1"]]}  {
  create_cell -only_physical FIL_1 FILLERX1
}
```

When you read the secondary floorplan file, the tool creates the power and ground nets and the physical-only cells in the logical netlist. Then, when you read the main floorplan file, the tool applies the floorplan information for the power and ground nets and the physical-only cells to the corresponding objects. Therefore, it is important that you read the secondary floorplan file before you read the main floorplan file, as shown in the following example. This

allows the tool to use the preroute information for the power and ground nets and physical-only cells during optimization.

```
dc_shell-topo> read_floorplan floorplan.tcl.objects
1
dc_shell-topo> read_floorplan floorplan.tcl
Information: Successfully applied 2 of the 2 'create_net_shape'
commands read. (DCT-143)
1
```

Manually Defining Physical Constraints

This section describes how to manually define physical constraints in the following subsections:

- Defining Physical Constraints Overview
- Defining the Die Area
- Defining the Core Placement Area With the create_site_row Command
- Defining Placement Area With the set_aspect_ratio and set_utilization Commands
- Defining Port Locations
- Defining Macro Location and Orientation
- Defining Placement Blockages
- Defining Voltage Area
- Defining Placement Bounds
- Creating Wiring Keepouts
- Creating Preroutes
- Creating User Shapes
- Defining Physical Constraints for Pins
- Creating Design Via Masters
- Creating Vias
- Creating Routing Tracks
- Creating Keepout Margins
- Computing Polygons

Defining Physical Constraints Overview

This section describes the commands you can use to define the floorplan's physical constraints manually. Design Compiler in topographical mode can read physical constraints (floorplan information) from a Tcl-based script of physical commands.

You manually define physical constraints when you cannot obtain this information from a DEF file or from the `read_floorplan` command. For details, see Importing Physical Constraints From an IC Compiler DEF Formatted File and Importing Physical Constraints From an IC Compiler write_floorplan Formatted File.

After you have manually defined your physical constraints in a Tcl script file, use the `source` command to apply these constraints. Keep the following points in mind when you manually define physical constraints:

- You must read in the design before applying user-specified physical constraints.
- You must apply user-specified physical constraints during the first topographical mode session.

Table 10-4 lists the commands that you use to set physical constraints.

Table 10-4 Commands That Define Physical Constraints

To define this physical constraint	Use these commands
Die Area For details, see Defining the Die Area.	`create_die_area`
Floorplan estimate when exact area is not known For details, see Defining Placement Area With the set_aspect_ratio and set_utilization Commands.	`set_aspect_ratio` and `set_utilization`
Exact core area For details, see Defining the Core Placement Area With the create_site_row Command.	`create_site_row`
Relative port locations For details, see Defining Relative Port Locations.	`set_port_side`
Exact port locations For details, see Defining Exact Port Locations.	`set_port_location`
Macro location and orientation For details, see Defining Macro Location and Orientation.	`set_cell_location`
Placement keepout (blockages) For details, see Defining Placement Blockages.	`create_placement_blockage`

Table 10-4 Commands That Define Physical Constraints (Continued)

To define this physical constraint	Use these commands
Voltage area For details, see Defining Voltage Area.	`create_voltage_area`
Placement bounds For details, see Defining Placement Bounds.	`create_bounds`
Wiring keepouts For details, see Creating Wiring Keepouts.	`create_route_guide`
Preroutes For details, see Creating Preroutes.	`create_net_shape`
User shapes For details, see Creating User Shapes.	`create_user_shape`
Pin physical constraints For details, see Defining Physical Constraints for Pins.	`set_pin_physical_constraints` `create_pin_guide`
Design via masters For details, see Creating Design Via Masters.	`create_via_master`
Vias For details, see Creating Vias.	`create_via`
Tracks For details, see Creating Routing Tracks.	`create_track`
Keepout margins For details, see Creating Keepout Margins.	`set_keepout_margin`
Polygons For details, see Computing Polygons.	`compute_polygons`

Defining the Die Area

Design Compiler topographical technology allows you to manually define the die area, also known as the cell boundary. The die area represents the silicon boundary of a chip, and it encloses all objects of a design, such as pads, I/O pins, and cells. There should be only one die area in a design.

Typically, you create floorplan constraints, including the die area, in your floorplanning tool, and import these physical constraints into Design Compiler topographical mode. However, if you are not using a floorplanning tool, you need to manually create your physical

constraints. The create_die_area command allows you to define your die area from within Design Compiler in topographical mode.

When using the create_die_area command, you can use the -coordinate option if your die area is a rectangle. If your die area is rectilinear, you must use the -polygon option. You can use the -polygon option to specify rectangles, but you cannot use the -coordinate option to specify rectilinear die areas. For example, the following commands create the same rectangular die area:

```
create_die_area -coordinate { 0 0 100 100 }
create_die_area -coordinate { {0 0} {100 100} }
create_die_area -polygon { {0 0} {100 0} {100 100} {0 100} }
```

You can use the get_die_area command to return a collection containing the die area of the current design. If the die area is not defined, the command returns an empty string.

Use the get_attribute command to get information about the die area, such as the object class, bounding box information, coordinates of the current design's die area, die area boundary, and die area name.

You can use the report_attribute command to browse attributes, such as object_class, bbox, boundary, and name.

For example, you can create a new die area, as shown:

```
dc_shell-topo> create_die_area -polygon {{0 0} {0 400} {200 400} {200 200} \
               {400 200} {400 0}}
```

Then, use the get_attribute command to get the following information about the die area:

- To return the object class, use the object_class attribute, as shown:

  ```
  dc_shell-topo> get_attribute [get_die_area] object_class
  die_area
  ```

- To return the bounding box information and the coordinates of the current design's die area, use the bbox attribute, as shown:

  ```
  dc_shell-topo> get_attribute [get_die_area] bbox
  {0.000 0.000} {400.000 400.000}
  ```

- To return the die area boundary, use the boundary attribute, as shown:

  ```
  dc_shell-topo> get_attribute [get_die_area] boundary
  {0.000 0.000} {0.000 400.000} {200.000 400.000} {200.000 200.000}
  {400.000 200.000} {400.000 0.000} {0.000 0.000}
  ```

- To return the die area name, use the name attribute, as shown:

  ```
  dc_shell-topo> get_attribute [get_die_area] name
  {my_design_die_area}
  ```

Defining the Core Placement Area With the create_site_row Command

You can define the core placement area with the `create_site_row` command. The core placement area is a box that contains all rows. It represents the placeable area for standard cells. The core area is smaller than the cell boundary (the die area). Pads, I/O pins, and top-level power and ground rings are found outside the core placement area. Standard cells, macros, and wire tracks are typically found inside the core placement area. There should be only one core area in a design.

You use the `create_site_row` command to create a row of sites or a site array at a specified location. A site is a predefined valid location where a leaf cell can be placed; a site array is an array of placement sites and defines the placement core area.

To create a horizontal row of 100 CORE_2H sites with a bottom-left corner located at (10,10) and rows spaced 5 units apart, enter

```
dc_shell-topo> create_site_row -count 100 -kind CORE_2H -space 5 \
                            -coordinate {10 10}
```

To report physical information, such as core area, aspect ratio, utilization, total fixed cell area, and total movable cell area, use the `report_area -physical` command.

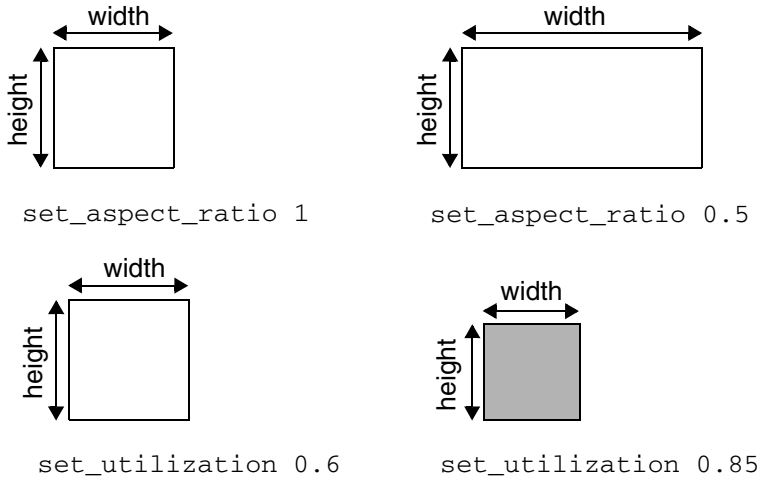
Defining Placement Area With the set_aspect_ratio and set_utilization Commands

If you have not defined your die area with the `create_die_area` command, defined a floorplan area with the `create_site_row` command, or imported floorplan information from a floorplanning tool, you can use the `set_aspect_ratio` and `set_utilization` commands to estimate the placement area.

The aspect ratio is the height-to-width ratio of a block; it defines the shape of a block. Utilization specifies how densely you want cells to be placed within the block. Increasing utilization reduces the core area.

Figure 10-4 illustrates how to use these commands.

Figure 10-4 Using the set_aspect_ratio and set_utilization Commands



Defining Port Locations

You can define constraints that restrict the placement and sizing of ports by using the `set_port_side` command or `set_port_location` command. You can specify relative or exact constraints.

To learn how to define relative and exact port locations, see

- Defining Relative Port Locations
- Defining Exact Port Locations

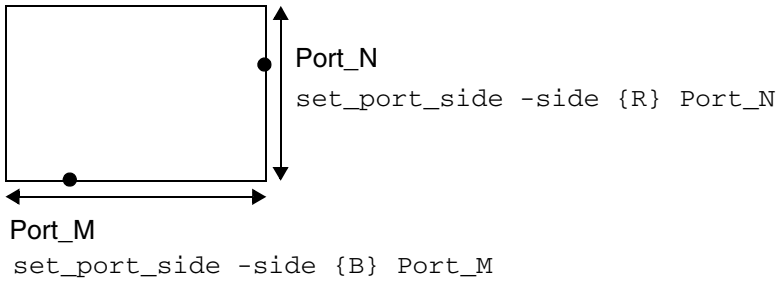
Defining Relative Port Locations

Use the `set_port_side` command to define relative port locations as follows:

```
set_port_side port_name -side {L|R|T|B}
```

Valid sides are left (L), right (R), top (T), or bottom (B). A port can be placed at any location along the specified side. If the port side constraints are provided, the ports are snapped to the specified side. Otherwise, by default, the ports are snapped to the side nearest to the port location assigned by the coarse placer. Figure 10-5 shows how you define port sides by using the `set_port_side` command.

Figure 10-5 Setting Relative Port Sides



Defining Exact Port Locations

Use the `set_port_location` command to annotate the port location on the specified port, defining exact port locations as follows:

```
set_port_location port_name -coordinate {x y}
```

where `-coordinate` is the lower left point of the port shape.

The following example annotates port Z, a top-level port in the current design, with the location at 100, 5000.

```
dc_shell-topo> set_port_location -coordinate {100 5000} Z
```

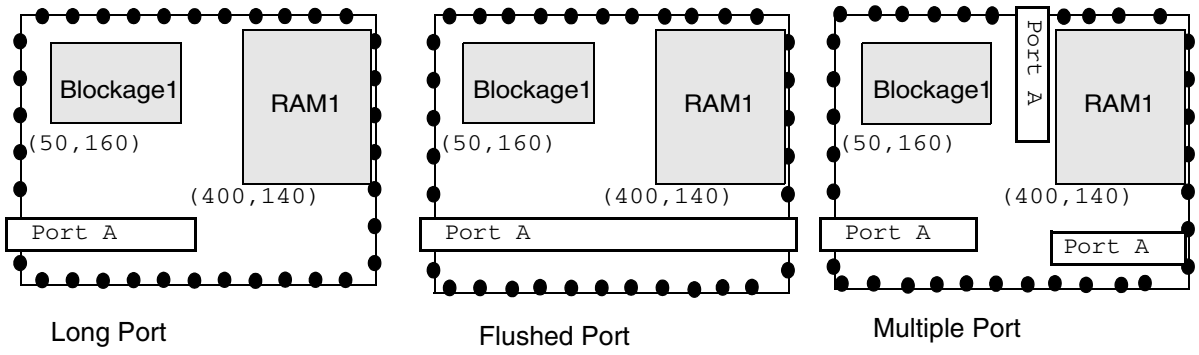
In addition, you can use the `-layer_name` option and `-layer_area` option to define metal layer geometry. For example, the following command specifies that the layer geometry is a rectangle of dimensions 10 by 20 for port Z.

```
dc_shell-topo> set_port_location -layer_name METAL1 \
                      -layer_area {-5 -10 5 10} Z
```

These options allow you to define long ports so that connections to the long port can be made along any point on the specified metal layer during virtual-layout based optimization. A long port can be an input, output, or bidirectional port. You specify a long port location by using a dimension greater than the minimum metal area. Figure 10-6 illustrates the different types of port dimensions: Long port, flushed port, and multiple port.

- A long port has the metal area defined with a single metal layer and touches only one side of the block core area boundary.
- A flushed port has the metal area defined with a single metal layer and touches the block core area boundary on two sides.
- A multiple port has more than one connection to the block and can touch the block boundary on two or more sides and can use one or more layers. To create a multiple port, use the `-append` option of the `set_port_location` command. With this option you add multiple shapes to a port and create a multiple port.

Figure 10-6 Types of Port Dimension Specifications



Defining Macro Location and Orientation

You can define exact macro locations by setting the following command:

```
set_cell_location cell_name -coordinates {x y} -fixed
                            -orientation {N|S|E|W|FN|FS|FE|FW}
```

where the `-coordinates` option specifies the coordinate of the lower left corner of the cell's bounding box. Specify the coordinates in microns relative to the block orientation. The orientation value is one of the rotations listed in the following table.

Orientation	Rotation
N (default)	Nominal orientation, 0-degree rotation (north)
S	180 degrees (south)
E	270 degrees counterclockwise (east)
W	90 degrees counterclockwise (west)
FN	Reflection in the y-axis (flipped north)
FS	180 degrees, followed by reflection in the y-axis (flipped south)
FE	270 degrees counterclockwise, followed by reflection in the y-axis (flipped east)
FW	90 degrees counterclockwise, followed by reflection in the y-axis (flipped west)

Defining Placement Blockages

Use the create_placement_blockage command to define placement blockages. The -bbox option specifies the coordinates of the bounding box of the blockage. Use the -type option to specify the type of placement blockage to be created—hard, soft, or partial.

The -type values are described as follows:

- A hard placement blockage prevents the placer from placing any standard cells or hard macros in the specified region. This is the -type option default.
- A soft placement blockage prevents the coarse placer from placing any standard cells or hard macros in the specified region. However, cells might be placed there during optimization or legalization.
- A partial placement blockage limits the amount of area used to place cells to the specified percent of the blockage area. When you create a partial blockage, you must use the -blocked_percentage option to specify the blocked percentage value.

During coarse placement, Design Compiler can evenly distribute cells outside the partial blockages and prevent the clumping of cells around their perimeters. To do this, set the placer_enable_redefined_blockage_behavior variable to true if your design contains partial blockages created by using the -type partial option.

You can define a blockage area where only buffers or inverters can be placed by specifying the -buffer_only option with the -type partial option. You must set the placer_enable_redefined_blockage_behavior variable to true when using the -buffer_only option.

The following example shows a blockage created with the -buffer_only option. The blocked percentage is 50 percent:

```
dc_shell-topo> create_placement_blockage -buffer_only -name my_blockage \
               -type partial -blocked_percentage 50 -bbox {{0 0} {100 100}}
```

The -buffer_only option cannot be used with the -type hard, -type soft, -no_hard_macro, -no_rp_group, -no_register, -no_pin, or -category options.

You can specify either to include specific library cells or cell instances in a blockage area or to exclude them from a blockage area by using the -category option with the -type partial option. You must set the placer_enable_redefined_blockage_behavior variable to true when using the -category option.

To create a predefined category of cells, you must first define an attribute for the library cells or cell instances by using the define_user_attribute command. Then apply the attribute to the desired library cells or cell instances by using the set_attribute command. After you do this, you can create a category blockage by using the attribute name as the argument to the -category option.

To prevent library cells or cell instances from being placed within the specified blockage area, set your user-defined attribute to `true` for the specific library cells or cell instances. By default, all cells are allowed within the specified blockage area. This is equivalent to the user-defined attribute set to `false`.

In the following example, Design Compiler blocks all cells except isolation (bfiso*) cells:

```
dc_shell-topo> define_user_attribute -type boolean -classes lib_cell only_iso
dc_shell-topo> set_attribute [get_lib_cell mylib/*] only_iso true
dc_shell-topo> set_attribute [get_lib_cell mylib/bfiso*] only_iso false
dc_shell-topo> create_placement_blockage -bbox {{230 795} {830 1940}} \
            -name blk_VA_edge -type partial -blocked_percentage 20 -category only_iso
```

Each `cell` or `lib_cell` class can have multiple user attributes defined for it, controlling the library cell or cell instance behavior in multiple regions. If a cell instance has attributes defined for it and its library cell has different attributes defined for it, the cell instance's attribute takes precedence.

User attributes defined on cell instances are saved when the design is written in .ddc format; however, user attributes defined on library cells are not saved when the design is written. You need to re-create the `lib_cell` user-attribute and reapply it to the library cell if you reload your design in another session.

Multiple blockages can use the same attribute. However, each blockage can accept only one attribute. The `-blocked_percentage` option setting affects all cells in the blockage area, regardless of the attribute.

The `-category` option cannot be used with the `-type hard`, `-type soft`, `-no_hard_macro`, `-no_rp_group`, `-no_register`, `-no_pin`, or `-buffer_only` options.

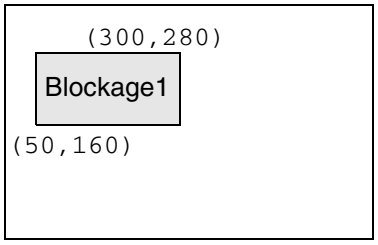
Use the `-no_rp_group` option to specify relative placement group blockages. This option prevents the placement of relative placement groups within the specified area. The option only affects cells that belong to relative placement groups. The `-no_rp_group` option must be used with the `-type partial` option, and it cannot be used with the `-no_hard_macro`, `-no_register`, and `-no_pin` options.

To restrict Design Compiler from placing any register cells within the specified partial blockage area during coarse placement, specify the `-no_register` option with the `-type partial` option. However, this restriction does not apply to optimization performed after coarse placement. If you specify the `-retime` option with the `compile_ultra` command, the registers that are moved during optimization can be placed in this area.

Figure 10-7 uses the `create_placement_blockage` command to define a hard placement blockage named Blockage1.

Figure 10-7 Defining a Hard Placement Blockage

```
create_placement_blockage\
-name Blockage1 -type hard \
-bbox {50 160 300 280}
```



You can use the `get_placement_blockages` command to return a collection of placement blockages from the current design. The command returns a collection of placement blockages if one or more blockages meet the specified criteria. If no blockages match the criteria, it returns an empty string. You can use the `get_placement_blockages` command as an argument to another command or assign its result to a variable.

To create a collection containing all blockages within a specified region, use the `-within` *`region`* option with the `get_placement_blockages` command. The region boundary can be a rectangle or a polygon. In the following example, Design Compiler returns one rectangular placement blockage based on the specified region. In the example, the first xy pair {2 2} represents the lower-left corner of the rectangle, and the second xy pair {25 25} represents the upper-right corner of the rectangle:

```
dc_shell-topo> get_placement_blockages * -within {{2 2} {25 25}}
{"PB#5389"}
```

To filter the collection with an expression, use the `-filter` *`expression`* option with the `get_placement_blockages` command. In the following example, Design Compiler returns all placement blockages that have an area greater than 900:

```
dc_shell-topo> get_placement_blockages * -filter "area > 900"
{"PB#4683"}
```

Defining Voltage Area

You define voltage areas by using the `create_voltage_area` command. This command enables you to create a voltage area on the core area of the chip. The voltage area is associated with hierarchical cells. The tool assumes the voltage area to be an exclusive, hard move bound and tries to place all the cells associated with the voltage area within the

defined voltage area, as well as place all the cells not associated with the voltage area outside the defined voltage area.

Example 10-27 uses the `create_voltage_area` command to constrain the instance INST_1 to lie within the voltage area whose coordinates are lower-left corner (100 100) upper-right corner (200 200).

Example 10-27 Using the create_voltage_area Command With the -name Option

```
dc_shell-topo> create_voltage_area -name my_design \
   -coordinate {100 100 200 200} INST_1
```

Example 10-28 uses the `create_voltage_area` command to constrain cells in power domain PD1 to lie within the voltage area whose coordinates are lower-left corner (100 100) upper-right corner (200 200).

Example 10-28 Using the create_voltage_area Command With the -power_domain Option

```
dc_shell-topo> create_voltage_area -power_domain PD1 \
   -coordinate {100 100 200 200}
```

Note:

The `-name` option and the `-power_domain` option of the `create_voltage_area` command are mutually exclusive.

In addition to the `create_voltage_area` command, the following commands are supported: `remove_voltage_area` and `report_voltage_area`

Voltage areas are automatically defined when you import your floorplan information from IC Compiler with the `read_floorplan` command. For details, see Importing Physical Constraints From an IC Compiler write_floorplan Formatted File. You need to define voltage areas manually when importing floorplan information from a DEF file.

To visually inspect your defined voltage areas, use the visual mode in the Design Vision layout window.

Defining Placement Bounds

To learn how to create and use placement bounds, see

- Placement Bounds Overview
- Creating Placement Bounds
- Order for Creating Placement Bounds
- Guidelines for Defining Placement Bounds Effectively
- Returning a Collection of Bounds
- Summary of Bounds Commands

Placement Bounds Overview

A placement bound is a rectangular or rectilinear area within which to place cells and hierarchical cells. You use the `create_bounds` command to specify placement constraints for coarse placement.

Defining a placement bound enables you to group cells such as clock-gating cells or extremely timing-critical groups of cells that you want to guarantee will not be disrupted for placement by other logic. During placement, the tool ensures that the cells you grouped remain together. Placement bounds are placement constraints.

Usually, it is best not to impose bounds constraints and instead allow the tool full flexibility to optimize placement for timing and routability. However, if QoR does not meet your requirements, you might improve QoR by using the `create_bounds` command.

To use placement bounds effectively, make the number of cells you define in placement bounds relatively small compared with the total number of cells in the design. When you define placement bounds, the solution space available to the placer to reach the optimal result gets smaller.

Creating Placement Bounds

Use the `create_bounds` command to specify placement bounds. You can specify two different types of bounds: move bounds and group bounds. Move bounds restrict the placement of cells to a specific region of the core area. Move bounds require absolute coordinates to be specified, using the `-coordinate` option. Group bounds, however, are floating region constraints, using the `-dimension` option. Cells in the same group bound are placed within a specified bound but the absolute coordinates are not fixed. Instead, they are optimized by the placer. If you do not use either the `-dimension` or `-coordinate` option, the tool creates a group bound with the bounding box computed internally by the tool.

In addition, move bounds can be soft, hard, or exclusive. Group bounds can be soft or hard.

- Soft bounds specify placement goals, with no guarantee that the cells will be placed inside the bounds. If timing or congestion cost is too high, cells might be placed outside the region. This is the default.
- Hard bounds force placement of the specified cells inside the bounds. To specify hard bounds, use the `-type hard` option along with the `-dimension` or `-coordinate` option. However, overusing hard bounds can lead to inferior placement solutions.
- Exclusive bounds force the placement of the specified cell inside the bounds. All other cells must be placed outside the bounds. To specify exclusive bounds, use the `-exclusive` option.

You can specify the following types of placement bounds:

- Soft group bound

```
create_bounds -dimension {100 100} -name foo1 INST1
```

- Soft move bound

```
create_bounds -coordinate {0 0 10 10} -name foo2 INST2
```

- Hard group bound

```
create_bounds -dimension {100 100} -type hard \
                      -name foo3 INST3
```

- Hard move bound

```
create_bounds -coordinate {0 0 10 10} -type hard \
                      -name foo4 INST4
```

- Exclusive move bound

```
create_bounds -coordinate {0 0 10 10} -type hard \
                      -name foo5 INST5
```

You can specify color for move bounds by using the following options with the `create_bounds` command:

- `-color`

  Specify the move bound color either by specifying an integer value between 0 and 63 or by specifying a color name string with one of the following values: `black`, `blue`, `green`, `cyan`, `brown`, `purple`, `red`, `magenta`, `salmon`, `orange`, `yellow`, or `white`. The default is `no_color`, meaning that no color is applied.

- `-cycle_color`

  The `-cycle_color` option allows the tool to automatically assign a move bound color. By default, this option is turned off.

Order for Creating Placement Bounds

If you impose bounds constraints, create bounds in the following order:

1. Floating group bounds where location and dimension are optimized by the tool. For these bounds, do not specify dimensions.
2. Floating group bounds with fixed dimensions. For these bounds, specify dimensions.
3. Fixed move bounds with fixed location and dimension. For these bounds, specify coordinates that define the bound.

Guidelines for Defining Placement Bounds Effectively

Use the following guidelines when you create placement bounds:

- Use soft and hard move bounds sparingly in your design.

- Avoid placing cells in more than one bound.
- Be aware that including small numbers of fixed cells in group bounds can move the bound.
- Do not use placement bounds as keepouts.
- Maintain even cell density distribution over the chip.

Returning a Collection of Bounds

You can use the `get_bounds` command to return a collection of bounds from the current design based on the criteria you specify. You can use the `get_bounds` command at the command prompt, or you can nest it as an argument to another command, such as the `report_bounds` command. In addition, you can assign the `get_bounds` result to a variable.

When issued from the command prompt, the `get_bounds` command behaves as though you have called the `report_bounds` command to report the objects in the collection. By default, it displays a maximum of 100 objects. You can change this maximum by using the `collection_result_display_limit` variable.

In the following example, Design Compiler returns all bounds that have a name starting with my_bound:

```
dc_shell-topo> get_bounds my_bound*
```

Summary of Bounds Commands

Table 10-5 summarizes the commands related to placement bounds.

Table 10-5 Summary of Bounds Commands

Command	Description
`create_bounds`	Creates rectangular and rectilinear move bounds and group bounds
`update_bounds`	Updates a bound by adding or removing contents
`get_bounds`	Returns a collection of bounds from the current design
`remove_bounds`	Removes bounds set using `create_bounds`
`report_bounds`	Reports bounds and bound IDs set using `create_bounds`

Creating Wiring Keepouts

You can create wiring keepouts by using the `create_route_guide` command. Example 10-29 uses the `create_route_guide` command to create a keepout named my_keepout_1 in the METAL1 layer at coordinates {12 12 100 100}.

Example 10-29

```
dc_shell-topo> create_route_guide -name  "my_keepout_1"  \
                                  -no_signal_layers "METAL1" \
                                  -coordinate {12 12 100 100}
```

Creating Preroutes

You can preroute a group of nets, such as clock nets, before routing the rest of the nets in the design. During global routing, the tool considers these preroutes while computing the congestion map; this map is consistent with IC Compiler. This feature also addresses correlation issues caused by inconsistent floorplan information.

To define preroutes, you use the `create_net_shape` command. You can create three different types of net shapes as shown in Figure 10-8: path, wire (horizontal and vertical), and rectangle.

Figure 10-8 Types of Preroutes

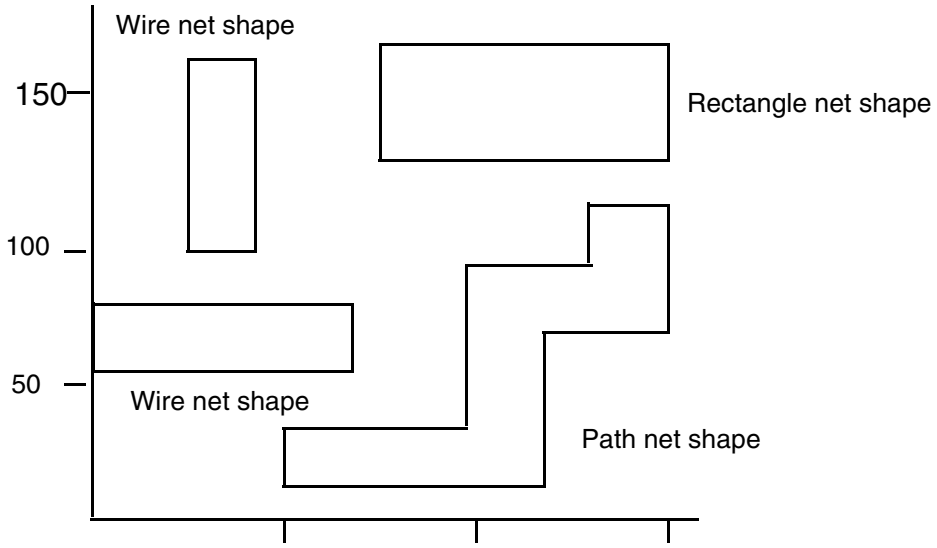


Table 10-6 shows how to use some of the `create_net_shape` command options.

Table 10-6 Using the create_net_shape Command Options to Create Preroutes

To do this	Use this option
Specify the type of net shape.	`-type path | wire | rect`
Specify the type of net.	`-net_type ground | power | clock | signal`
Specify the net on which the net shape is created, for example VSS or VDD.	`-net name`
Specify the layer on which the net shape is created.	`-layer name`
Specify the origin of the net shape for wires.	`-origin x y`
Specify the bounding box of the net shape for rectangles.	`-bbox llx lly urx ury`
Specify the point sequence of the net shape for paths.	`-points {x0 y0 x1 y1... xn yn}`
Specify the alignment type of a wire or path. Default is square.	`-path_type { square | round | extend_half_width | octagon }`
Specify the length and width for wires.	`-length real_number -width real_number`
Specify the type of route.	`-route_type {user_enter | signal_route | signal_route_global | signal_route_detail | pg_ring | pg_strap | pg_macro_io_pin_conn | pg_std_cell_pin_conn | clk_ring | clk_strap | clk_zero_skew_route | bus | shield | shield_dynamic | clk_fill_track}`
Specify the vertical orientation (Default is horizontal).	`-vertical`

The following examples show how to create the net shapes shown in Figure 10-8.

Example 10-30 uses the `create_net_shape` command to create two wire net shapes.

Example 10-30 Creating Two Wire Net Shapes

```
create_net_shape -type wire -net VSS \
      -bbox {0 60 80 80} \
      -layer METAL5 \
```

```
      -route_type pg_strap \
      -net_type ground
create_net_shape -type wire -net VSS \
      -origin {50 100} \
      -width 20 \
      -length 80 \
      -layer METAL4 -route_type pg_strap -vertical
```

Example 10-31 uses the create_net_shape command to create the path net shape.

Example 10-31 Creating a Path Net Shape

```
create_net_shape -type path -net VSS\
      -points {70 30 110 30 110 90 150 90 150 110} \
      -width 0.20 -layer M3 \
      -route_type pg_std_cell_pin_conn
```

Example 10-32 uses the create_net_shape command to create the rectangle net shape.

Example 10-32 Creating a Rectangle Net Shape

```
create_net_shape -type rect -net VSS \
      -bbox {{80 140} {160 180}} \
      -layer METAL1 -route_type pg_strap
```

Creating Preroutes for Power and Ground Nets

Design Compiler uses preroute data for congestion analysis during optimization only for preroutes that are created on nets that exist in the design. When preroute data is extracted from a DEF file by the extract_physical_constraints command, Design Compiler also extracts the power and ground nets from the DEF file and creates the preroutes on them.

Design Compiler takes the power and ground nets into account when you run the compile_ultra command. However, when preroutes are created by the create_net_shape command, you need to create the power and ground nets before creating the preroutes on them because the power and ground nets are not in the logical netlist.

You can create power and ground nets in Design Compiler in topographical mode by using the create_net command. This allows the tool to use the preroute information created by the create_net_shape command for the power and ground nets.

In the following example, the tool creates the power and ground nets and then creates preroutes on the power and ground nets:

```
dc_shell-topo> create_net -power vdd
1
dc_shell-topo> create_net -power vss
1
dc_shell-topo> create_net_shape -net vdd -points {{0 20} {80 20}} \
               -width 10 -layer M1 -route_type pg_ring -no_snap
Information: net shape with type path will be created during compile.
```

```
(DCT-020)
1
dc_shell-topo> create_net_shape -net vss -points {{20 0} {20 80}} \
               -width 10 -layer M3 -route_type pg_ring -no_snap
Information: net shape with type path will be created during compile.
 (DCT-020)
1
```

When a preroute is created on a net that does not exist in the design, Design Compiler does not use the preroute data when you run the `compile_ultra` command. It removes the preroute data.

This behavior is consistent with the `place_opt` command in IC Compiler.

If you want to preserve preroute data for nets that do not exist in the design, set the `dct_preserve_all_preroutes` variable to `true` before you run the `compile_ultra` command. When this variable is set to `true`, Design Compiler does not remove the preroutes during optimization, but it does not use the preroute data on the non-existing net during optimization. The default for the variable is `false`. After you run the `compile_ultra` command, generate the floorplan file using the `write_floorplan` command.

Creating User Shapes

You can create user shapes by using the `create_user_shape` command. A user shape is a metal shape that is not associated with a net. You can specify the following types of user shapes by using the `-type` option: wire (horizontal or vertical), path, rectangle, polygon, and trapezoid.

If you do not specify the `-type` option, the user shape type is determined by the following rules, in order of precedence:

1. If you use the `-origin` option, the user shape is a wire. The wire is horizontal unless you also specify the `-vertical` option.
2. If you use the `-bbox` option, the user shape is a wire if you also use the `-path_type`, `-route_type`, or `-vertical` options. If you do not use any of these additional options, the user shape is a rectangle.
3. If you use the `-points` option, the user shape is a path.

In the following example, Design Compiler creates a wire user shape:

```
dc_shell-topo> create_user_shape -type wire \
               -origin {0 0} -length 10 -width 2 -layer M1
```

Use the `write_floorplan -user_shape` command to save user shapes to the generated floorplan file. Alternatively, use the `write_floorplan -preroute` command to save user shapes and net shapes to the floorplan file.

The `create_user_shape` commands are saved in the Preroute section of the floorplan file, as shown in the following example:

```
#************************************************
#   SECTION: Preroutes, with number: 6
#************************************************
remove_user_shape *
create_user_shape -no_snap -type wire -layer METAL1 -datatype 0 \
-path_type 0 -width 2000 -route_type signal_route -length 3000 -origin {100 1500}
create_user_shape -no_snap -type wire -layer METAL2 -datatype 0 -path_type 0 \
-width 2000 -route_type signal_route -length 3000 -origin {100 1500}
create_user_shape -no_snap -type path -layer METAL4 -datatype 0 -path_type 0 \
-width 200 -route_type signal_route -points {{100 100}  {400 100} {400 400} {600 400}}
create_user_shape -no_snap -type rect -layer METAL5 -datatype 0 \
-route_type user_enter -bbox {{100 100} {600 600}}
create_user_shape -no_snap -type poly -layer METAL6 -datatype 0 -boundary {{100 100} \
{300 200} {600 600} {400 400} {200 400} {100 100}}
create_user_shape -no_snap -type trap -layer METAL3 -datatype 0 -boundary {{100 100} \
{600 100} {400 600} {200 600}}
```

To remove objects that are user shapes, use the `remove_user_shape` command. Use an asterisk (*) to indicate that all user shapes in the design be removed, or use the *`user_shapes`* argument to specify a list of user shapes to be removed:

```
dc_shell-topo> remove_user_shape *
```

You can use the layout window in Design Compiler Graphical to view user shapes in the floorplan. You control the visibility of user shapes in the active layout view by setting options on the View Settings panel. For more information about viewing user shapes in the floorplan, see the “Examining Preroutes” topic in Design Vision Help.

Table 10-7 summarizes the commands related to user shapes.

Table 10-7 Summary of User Shape Commands

Command	Description
`create_user_shape`	Creates a new user shape.
`remove_user_shape`	Removes user shapes.

Defining Physical Constraints for Pins

Design Compiler in topographical mode supports the following commands to control the placement of I/O cells and terminals:

- `set_pin_physical_constraints`
- `create_pin_guide`

Note:

Design Compiler in topographical mode supports only the placement of top-level design ports. It does not support the plan group or macro pin placement that is supported in IC Compiler design planning.

You can set physical constraints, such as the width and depth, on specified pins by using the `set_pin_physical_constraints` command. The following example shows the typical usage for the `set_pin_physical_constraints` command:

```
dc_shell-topo> set_pin_physical_constraints -pin_name {CCEN} -width 1.2 \
               -depth 1.0 -side 1 -offset 10 -order 2
```

In the example, the command sets constraints on the CCEN pin. The geometry width is 1.2 microns, the depth is 1.0 micron, and the pin abuts to the side 1 edge, which is the lower left-most vertical edge. The offset distance is 10 microns, and the relative placement order is 2. Therefore, the pin is placed at a location on the left-most edge.

You can also use the `create_pin_guide` command to create a pin guide to constrain the pin assignment to a specified bounding box. This allows you to specify which area the port should be placed. The following example creates a pin guide named abc for all ports whose name begins with the string xyz.

```
dc_shell-topo> create_pin_guide -bbox {{-20 50} {20 800}} \
               [get_ports xyz*] -name abc
```

The constraints that you specify are honored during the `compile_ultra` run.

By default, Design Compiler in topographical mode snaps ports to tracks during pin placement. However, you can prevent the snapping of ports to tracks by setting the `dct_port_dont_snap_onto_tracks` variable to `true`.

The pin constraints are saved to the design in the .ddc file. When you read the .ddc file in to a new Design Compiler topographical session, the pin constraints are restored.

The `write_floorplan` and `report_physical_constraints` commands do not report the pin constraint information. However, you can use the layout window in Design Compiler Graphical to verify that the pin constraints are placed correctly in the floorplan. To view the pin constraints in the layout view, select the Pin Guide visibility option (Vis) on the View Settings panel in the layout window. For more information, see the “Examining Physical Constraints” topic in Design Vision Help.

Design Compiler Graphical does not pass the pin constraints to floorplan exploration, and floorplan exploration does not pass the pin constraints to Design Compiler Graphical.

Creating Design Via Masters

Design via masters are created when

- A design via master is not in the technology file but it is in the DEF file being read by Design Compiler.
- You use the `create_via_master` command.

Via instances that are created from design via masters defined in a DEF file can be written out by using the `write_floorplan` command and then read in to Design Compiler by using the `read_floorplan` command. After the design via master is created, you can use it anywhere in the design, similar to a ContactCode definition in the technology file. The design via master is stored as an object together with the design cell in the binary design database, and therefore it can be used only in that design. To add an instance of a via defined as a design via master, use the `create_via` command. For more information, see the Creating Vias.

You can use the `report_physical_constraints` command to report design via masters. For a report example, see Reporting Design Via Masters.

Creating Vias

You can create vias by using the `create_via` command if the master via is defined in the technology file. Use the `-type` option to specify the type of via you want to create. The following example shows the creation of a via instance with its center specified as {213 215} and with an orientation of E. The via's size is determined by the via master, via1, which is defined in the technology file:

```
dc_shell-topo> create_via -no_snap -type via -net vdd -master via1 \
    -route_type pg_strap  -at {213 215} -orient E
```

The following example shows the creation of a via array with four columns and three rows. The via master is via4:

```
dc_shell-topo> create_via -no_snap -type via_array -net vdd -master via4 \
    -route_type signal_route -at {215 215} -orient W -col 4 -row 3 \
    -x_pitch 0.4 -y_pitch 0.5
```

Use the `-name` option to specify the via's name. If the via name is not specified, its name is automatically assigned internally in the Milkyway database, and the via name is passed to the .ddc file.

If a via is defined in the DEF file and it does not exist in the logic library, a FRAM view is created in the Milkyway design library to save that via master. When that via is instantiated as a preroute in the floorplan, its instance is saved as a via cell that references the via master in the FRAM view.

You can visually examine preroute vias in your floorplan by viewing them in the Design Vision layout window. You can display or hide vias in the active layout view by setting options on the View Settings panel. Vias are visible by default. For more information about controlling the visibility of vias in the active layout view, see Design Vision Help.

Creating Routing Tracks

You can create tracks for routing layers and polygon layers by using the `create_track` command. Tracks cover the entire die area. If the die area is a polygon, the tracks cover the die area in a rectangle, stretching outside the polygon die area.

The `create_track` command creates a group of tracks on the floorplan so the router can use them to perform detail routing. You must specify either a polygon layer or a routing layer for the tracks. Creating the track on a polygon layer is not intended to support routing on the polygon layer.

The following example creates routing tracks for a routing layer named m3 on the floorplan:

```
dc_shell-topo> create_track \
        -layer MET3 \
        -dir Y \
        -coord 0.000 \
        -space 0.290 \
        -count 4827 \
        -bounding_box {{0.000 0.000} {1460.000 1400.000}}

dc_shell-topo> create_track \
        -layer MET3 \
        -dir X \
        -coord 0.000 \
        -space 0.290 \
        -count 5034 \
        -bounding_box {{0.000 0.000} {1460.000 1400.000}}
```

To return a collection of tracks in the current design that meet your selection criteria, use the `get_tracks` command. For example, you can use the `-within` *`rectangle`* option to create a collection containing all tracks within the specified rectangle. The format of a rectangle specification is {{llx lly} {urx ury}}, which specifies the lower-left and upper-right corners of the rectangle. The coordinate unit is specified in the technology file.

The following example returns the tracks within the rectangle with the lower-left corner at {2 2} and the upper-right corner at {25 25}:

```
dc_shell-topo> get_tracks * -within {{2 2} {25 25}}
{"USER_TRACK_5389"}
```

The following example returns all tracks:

```
dc_shell-topo> get_tracks "*TRACK_*"
{"DEF_TRACK_4683 USER_TRACK_5389"}
```

Table 10-8 summarizes the commands related to tracks. The `create_track` and `remove_track` commands are written out by the `write_floorplan` command, while the other commands listed in the table are not.

Table 10-8 Summary of Routing Track Commands

Command	Description
`create_track`	Creates tracks for routing layers and polygon layers.
`remove_track`	Removes tracks from the current design.
`report_track`	Reports the routing tracks for a specified layer or for all layers.
`get_tracks`	Returns a collection of tracks in the current design that meet the selection criteria.

Creating Keepout Margins

You can create a keepout margin for the specified cell or library cell by using the `set_keepout_margin` command. Use the `-type hard` option or the `-type soft` option to specify the type of keepout, either hard or soft. The default is hard. Use the `-all_macros` option to apply the keepout margins to all macros, the `-macro_masters` *`object_list`* option for all instances of specified macro masters, or the `-macro_instances` *`object_list`* option for specified instances of macros.

You can specify explicit keepout margins by using the `-outer {`*`lx by rx ty`*`}` option, where the four numbers are the left, bottom, right, and top margins. A value of 0 results in no keepout margin for that side. The `-north` option sets the margins with respect to the north orientation of the cell.

Instead of specifying the keepout margins explicitly, you can have them derived automatically by using the `-tracks_per_macro_pin` *`value`* option, as shown in the following example. In this case, the keepout margin is calculated from the track width, the number of macro pins, and the specified track-to-pin ratio, which is typically set to a value near 0.5. A larger value results in larger keepout margins.

```
dc_shell-topo> set_keepout_margin -tracks_per_macro_pin .6 \
  -min_padding_per_macro .1 -max_padding_per_macro 0.2
```

The derived keepout margin is always hard; the `-type` option setting is ignored. The `-all_macros`, `-macro_masters`, and `-macro_instances` options are not allowed for derived margins. The derived margins are subject to minimum and maximum values that are specified by the `-min_padding_per_macro` and `-max_padding_per_macro` options.

Table 10-9 summarizes the commands related to keepout margins.

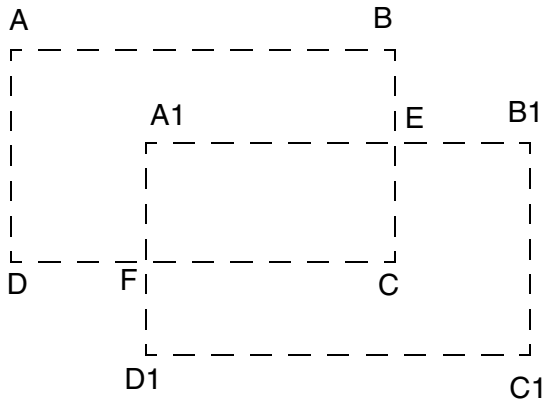
Table 10-9 Summary of Keepout Margin Commands

Command	Description
`set_keepout_margin`	Creates a keepout margin of the specified type for the specified cell or library cell
`remove_keepout_margin`	Removes keepout margins of a specified type for the specified cells or library cells in the design
`report_keepout_margin`	Reports keepout margins of a specified type for the specified cells in the design

Computing Polygons

The `compute_polygons` command returns a list or collection of polygons that exactly cover the region computed by performing a Boolean operation on the input polygons. The command performs an AND, OR, NOT, or XOR operation between two polygons or between two sets of polygons that are specified as lists of vertexes or as polygon collections. The typical scenario is to compute the geometry operations of physical objects, such as net shapes, user shapes, blockages, and so on. For example, assume you have two polygons, A-B-C-D-A and A1-B1-C1-D1-A1, as shown in Figure 10-9.

Figure 10-9 Example of an OR Operation on Two Polygons



The following example performs the Boolean OR operation on these polygons. The result is the B-E-B1-C1-D1-F-D-A-B polygon.

```
dc_shell-topo> compute_polygons -boolean or \
{{0 30} {30 30} {30 10} {0 10} {0 30}} \
{{10 20} {40 20} {40 0} {10 0} {10 20}}
{{30.000 30.000} {30.000 20.000} {40.000 20.000} {40.000 0.000} {10.000 0.000}
{10.000 10.000} {0.000 10.000} (0.000 30.000} {30.000 30.000}}
```

Figure 10-10 shows the resulting polygon.

Figure 10-10 Resulting Polygon

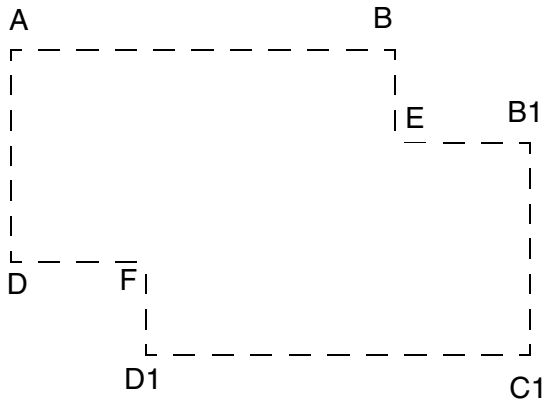


Table 10-8 summarizes the commands related to polygons.

Table 10-10 Summary of Polygon Commands

Command	Description
`compute_polygons`	Returns a list or collection of polygons that exactly cover the region computed by performing a Boolean operation on the input polygons.
`convert_to_polygon`	Returns a polygon list or collection for the specified objects. Bounds and placement blockage objects are supported.
`convert_from_polygon`	Converts each polygon into a list or collection of mutually exclusive rectangles. You can specify a single polygon or multiple polygons.
`resize_polygon`	Resizes the input polygon. You can specify a single polygon or multiple polygons.
`get_polygon_area`	Calculates the area of the input polygon.

Including Physical-Only Cells

You can include physical cells that do not have logic functions in your floorplan. These cells are referred to as physical-only cells. Examples of physical-only cells include filler cells, tap cells, flip-chip pad cells, endcap cells, and decap cells. Although physical-only cells have no logic function, they create placement blockages that the tool considers during optimization and congestion analysis.

To learn about physical-only cells, see

- Declaring Physical-Only Cells
- Identifying Physical-Only Cells
- Creating Collections With Physical-Only Cells
- Reporting Physical-Only Cells
- Saving Physical-Only Cells

Declaring Physical-Only Cells

You declare physical-only cells in the following ways:

- Specifying Physical-Only Cells Manually
- Extracting Physical-Only Cells From a DEF File

Specifying Physical-Only Cells Manually

To specify physical-only cells manually, perform the following steps:

1. Create the physical-only cell by using the `create_cell` command with the `-only_physical` option:

```
dc_shell-topo> create_cell -only_physical MY_U_PO_CELL MY_FILL_CELL
Information: create physical-only cell 'MY_U_PO_CELL'
Warning: The newly created cell does not have location.
         Timing will be inaccurate. (DCT-004)
```

   This creates the physical-only cell MY_U_PO_CELL, which references the MY_FILL_CELL physical library cell, and sets the `is_physical_only` attribute on the created cell.

2. Assign a location to the cell.

   Unlike physical cells with logic functions, Design Compiler topographical does not assign a location to a physical-only cell during synthesis.

To assign a location to a physical-only cell, use one of the following methods:

- Assign a location by using a DEF file.

  You can define the location of a physical-only cell in a DEF file, as shown in the following example:

  ```
  COMPONENTS 1 ;
  - MY_U_PO_CELL MY_FILL_CELL + FIXED ( 3100000 700000 ) N ;
  END COMPONENTS
  ```

  After you have defined the location in a DEF file, apply the location to the physical-only cell by using the `extract_physical_constraints` command.

- Assign a location using Tcl commands.

  You can define the location of physical-only cells in a Tcl floorplan file. For example, the following commands define the location of the MY_U_PO_CELL cell:

  ```
  set obj [get_cells {"MY_U_PO_CELL"} -all]
  set_attribute -quiet $obj orientation N
  set_attribute -quiet $obj origin {3100.000 700.000}
  set_attribute -quiet $obj is_fixed TRUE
  ```

  The `is_fixed` attribute must be assigned to the cell. Otherwise, the tool ignores the location specification. To assign the location to the physical-only cell, read the Tcl floorplan file into Design Compiler topographical mode using the `read_floorplan` command.

- Assign a location using the `set_cell_location` command.

  You can define the location of physical-only cells by using the `set_cell_location` command in Design Compiler topographical mode, as shown:

  ```
  set_cell_location -coordinates {3100.00 700.00} -orientation N \
     -fixed MY_U_PO_CELL
  ```

Extracting Physical-Only Cells From a DEF File

To extract physical-only cells from a DEF file and add them to the design, use the `-allow_physical_cells` option with the `extract_physical_constraints` command. The physical-only cell definition in the DEF file must contain the `+fixed` attribute. Otherwise, the tool ignores the location specification.

For example, if you want to extract the filler cells in the my_design.def file shown in Example 10-33, use the following command:

```
extract_physical_constraints -allow_physical_cells my_design.def
```

Example 10-33 DEF Definitions for Two Filler Cells

```
COMPONENTS 2 ;
- fill_1 FILL_CELL + FIXED ( 3100000 700000 ) N ;
```

```
- fill_2 FILL_CELL + FIXED ( 3000000 600000 ) N ;
END COMPONENTS
```

The following types of cells are considered physical-only cells. Design Compiler assigns the `is_physical_only` attribute on these cells when you run the `extract_physical_constraints -allow_physical_cells` command:

- Standard cell fillers
- Pad filler cells
- Corner cells
- Chip cells
- Cover cells
- Tap cells
- Cells containing only power and ground ports

In addition to extracting physical-only cells from the DEF file, the `extract_physical_constraints -allow_physical_cells` command also extracts some cells in the DEF file that are not identified as physical-only cells. These cells could be logic cells such as ROM or RAM cells. The tool updates the current design with these new logic cells and they are included in your Verilog netlist.

Design Compiler in topographical mode sets a `logical_cell_from_def` attribute on logic cells that are created from a DEF file. You can identify these logic cells by using the following command:

```
dc_shell-topo> get_cells -hierarchical -filter "logical_cell_from_def == true"
```

Note:

The physical-only cell definitions in the DEF file must include cell locations. Design Compiler does not assign locations to physical-only cells during synthesis. For more information about assigning a location to a physical-only cell, see Specifying Physical-Only Cells Manually.

Identifying Physical-Only Cells

Design Compiler topographical sets the `is_physical_only` attribute on all the physical-only cells. To check if a cell is a physical-only cell, run the following command:

```
get_attribute [get_cells -all $cell_name] is_physical_only
```

Creating Collections With Physical-Only Cells

You can create collections with physical-only cells. You can create a collection before specifying locations for the new physical-only cells or before compiling the design.

To learn about creating collections with physical-only cells, see

- Creating a Collection of All Cells Including Physical-Only Cells
- Creating a Collection of Only Physical-Only Cells

Creating a Collection of All Cells Including Physical-Only Cells

To return a collection of all cells in the design that includes physical-only cells, use the `get_cells` command with the `-all` option:

```
dc_shell-topo> get_cells -all
{fill_1 fill_2 U0 U1 U2 U3...}
```

Creating a Collection of Only Physical-Only Cells

To return a collection of all physical-only cells in the design, use the `all_physical_only_cells` command:

```
dc_shell-topo> all_physical_only_cells
{fill_1 fill_2}
```

Reporting Physical-Only Cells

You can report physical-only cells for the design and in the floorplan report.

To learn how to generate reports for physical-only cells, see

- Reporting Physical-Only Cell Information for the Design
- Reporting Physical-Only Cell Information in the Floorplan Report

Reporting Physical-Only Cell Information for the Design

To report physical-only cell information for the current design, use the `report_cell` command with the `-only_physical` option.

Example 10-34 shows the report for the fill_1 and fill_2 physical-only cells.

Example 10-34 Reporting Physical-Only Cell Information

```
dc_shell-topo> report_cell -only_physical
****************************************
Report : cell
         -only_physical
Design : test
Version: D-2010.03-SP3
Date : Mon Jul 12 02:56:25 2010
****************************************
Cell Reference Library Area Orient. Location
--------------------------------------------------------------------------------
fill_1 FILL_CELL xyz.mw 26.61 0 (3100.00, 700.00)
```

```
fill_2 FILL_CELL xyz.mw 26.61 0 (3000.00, 600.00)
--------------------------------------------------------------------------------
Total 2 cells 53.22
1
```

Reporting Physical-Only Cell Information in the Floorplan Report

To report the current design's floorplan information that includes physical-only cell information, use the `report_physical_constraints` command. Example 10-35 shows a sample `report_physical_constraints` report.

Example 10-35 report_physical_constraints Output for Physical-Only Filler Cells

```
Cell2   Location             Orientation   Fixed
--------------------------------------------------------------------------------
fill_1 {3100.000 700.000}     N            Yes
fill_2 {3000.00, 600.00}      N            Yes
--------------------------------------------------------------------------------
```

Saving Physical-Only Cells

The `write_floorplan` command writes out the physical-only cell information while writing out the floorplan. Example 10-36 shows the `write_floorplan` output for the two filler cells described in Example 10-33.

Example 10-36 write_floorplan Output for Physical-Only Filler Cells

```
#************************************************
# SECTION: Std Cells, with number: 2
#************************************************
set obj [get_cells {"fill_cell_1"} -all]
set_attribute -quiet $obj orientation N
set_attribute -quiet $obj origin {3100.000 700.000}
set_attribute -quiet $obj is_fixed TRUE
set obj [get_cells {"fill_cell_2"} -all]
set_attribute -quiet $obj orientation N
set_attribute -quiet $obj origin {3000 600.00}
set_attribute -quiet $obj is_fixed TRUE
The read_floorplan command extracts floorplan information
```

Similarly, the `read_floorplan` command reads the physical-only cell information while reading in the saved floorplan.

The physical-only cell information is saved in the .ddc file, along with other constraints, such as timing, and can be read back into Design Compiler.

Note:

Similar to all other physical information, the physical-only cell information is not written out in the ASCII netlist or the MilkyWay interface. It cannot be passed to IC Compiler through the .ddc file. The only way to pass physical information to IC Compiler is to use

physical guidance or floorplan exploration in Design Compiler Graphical. Design Compiler in topographical mode does not pass any physical information to IC Compiler.

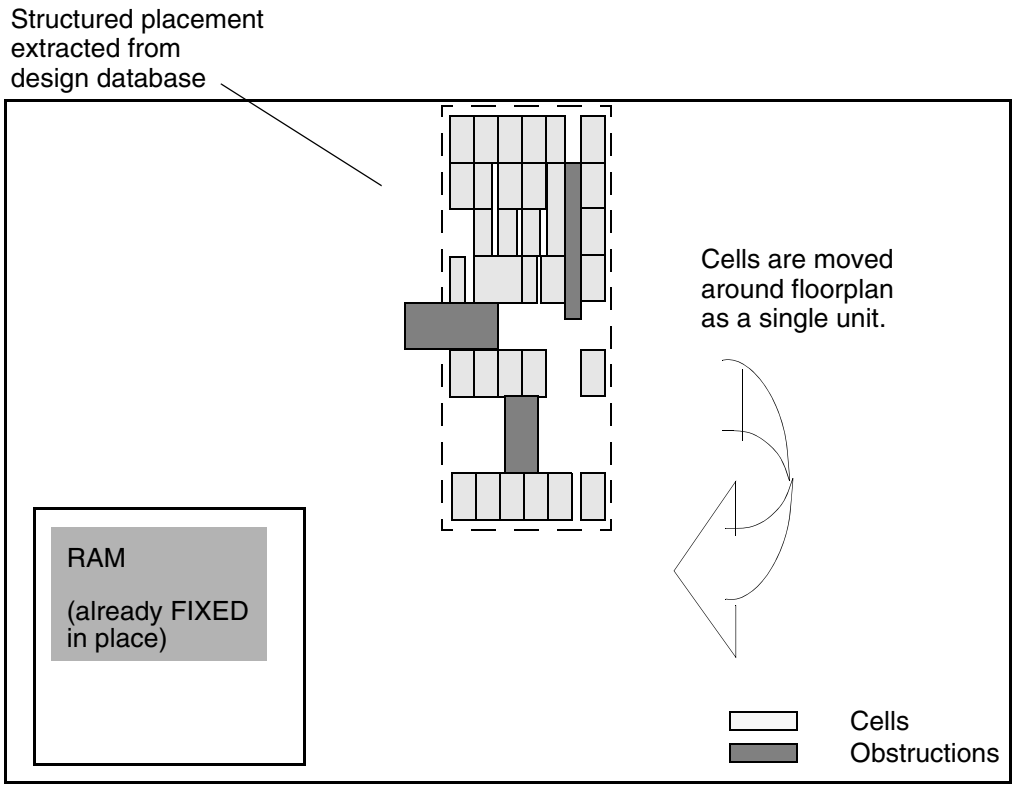
Specifying Relative Placement

The relative placement capability provides a way for you to create structures in which you specify the relative column and row positions of instances. During placement and optimization, these structures are preserved and the cells in each structure are placed as a single entity.

Relative placement is usually applied to datapaths and registers, but you can apply it to any cells in your design, controlling the exact relative placement topology of gate-level logic groups and defining the circuit layout. You can use relative placement to explore QoR benefits, such as shorter wire lengths, reduced congestion, better timing, skew control, fewer vias, better yield, and lower dynamic and leakage power.

The relative placement constraints implicitly generate a matrix structure of instances and control the placement of these instances. You use the resulting annotated netlist for optimization, during which the tool preserves the structure and places it as a single entity or group, as shown in Figure 10-11.

Figure 10-11 Relative Placement in a Floorplan



You can specify relative placement constraints in Design Compiler in topographical mode by using a dedicated set of Tcl commands, similar to the commands in IC Compiler. For information about the relative placement Tcl commands, see Summary of Relative Placement Tcl Commands.

Design Compiler in topographical mode also supports relative placement information embedded within the Verilog or VHDL description. You specify relative placement data within Verilog and VHDL by using HDL compiler directives. For more information, see Creating Relative Placement Using HDL Compiler Directives.

Relative placement is described in the following subsections:

- Benefits of Relative Placement
- Methodology for the Relative Placement Flow
- Creating Relative Placement Using HDL Compiler Directives
- Summary of Relative Placement Tcl Commands
- Creating Relative Placement Groups
- Anchoring Relative Placement Groups

Benefits of Relative Placement

Relative placement provides the following benefits:

- Provides a method for maintaining structured placement for legacy or intellectual property (IP) designs
- Reduces the placement search space in critical areas of the design, which means greater predictability of QoR (wire length, timing, power)
- Correlates better with IC Compiler
- Can minimize congestion and improve routability

Methodology for the Relative Placement Flow

The relative placement flow follows these major steps:

1. Read one or more of the following files in Design Compiler in topographical mode:
   - An RTL design with relative placement constraints specified by HDL compiler directives.
   - A GTECH netlist with relative placement constraints specified by HDL compiler directives.
   - A mapped gate-level netlist with relative placement constraints specified by HDL compiler directives.
   - A mapped gate-level netlist without relative placement constraints; you will specify the constraints with Tcl commands in step 2.

   For information about specifying relative placement using HDL compiler directives, including guidelines and restrictions, see the *HDL Compiler for VHDL User Guide* and the *HDL Compiler for Verilog User Guide*.

2. Define the relative placement constraints using Tcl commands. If the constraints have already been specified with HDL compiler directives, this step is optional. For information about specifying relative placement using HDL compiler directives, including guidelines and restrictions, see the *HDL Compiler for VHDL User Guide* and the *HDL Compiler for Verilog User Guide*.

   a. Create the relative placement groups by using the `create_rp_group` command. See Creating Relative Placement Groups.

   b. Add relative placement objects to the groups by using the `add_to_rp_group` command. See Adding Relative Placement Groups.

   Topographical mode annotates the netlist with the relative placement constraints and places an implicit `size_only` constraint on these cells.

3. Read floorplan information. For example, enter

   ```
   dc_shell> extract_physical_constraints floorplan.def
   ```

4. Check the relative placement by using the `check_rp_groups` command.

   The command reports relative placement failures, such as:

   ```
   RPGP-028: The height '%f' of the RP group '%s' is more than the height
   '%f' of the core area.
   RPGP-035 (warning) Relative placement leaf cell data may have been
   lost.
   RPGP-0348: All RP cells in cluster '%s' are not within same voltage
   region;
   ignoring voltage region placement.
   ```

   Note:

   The `check_rp_groups` command reports only the Tcl relative placement constraints at this stage in the flow. The command does not report constraints specified by HDL compiler directives until after the `compile_ultra` step.

5. Synthesize and optimize the design by using the `compile_ultra` command.

6. Visually verify the placement by using the layout view in the Design Vision layout window.

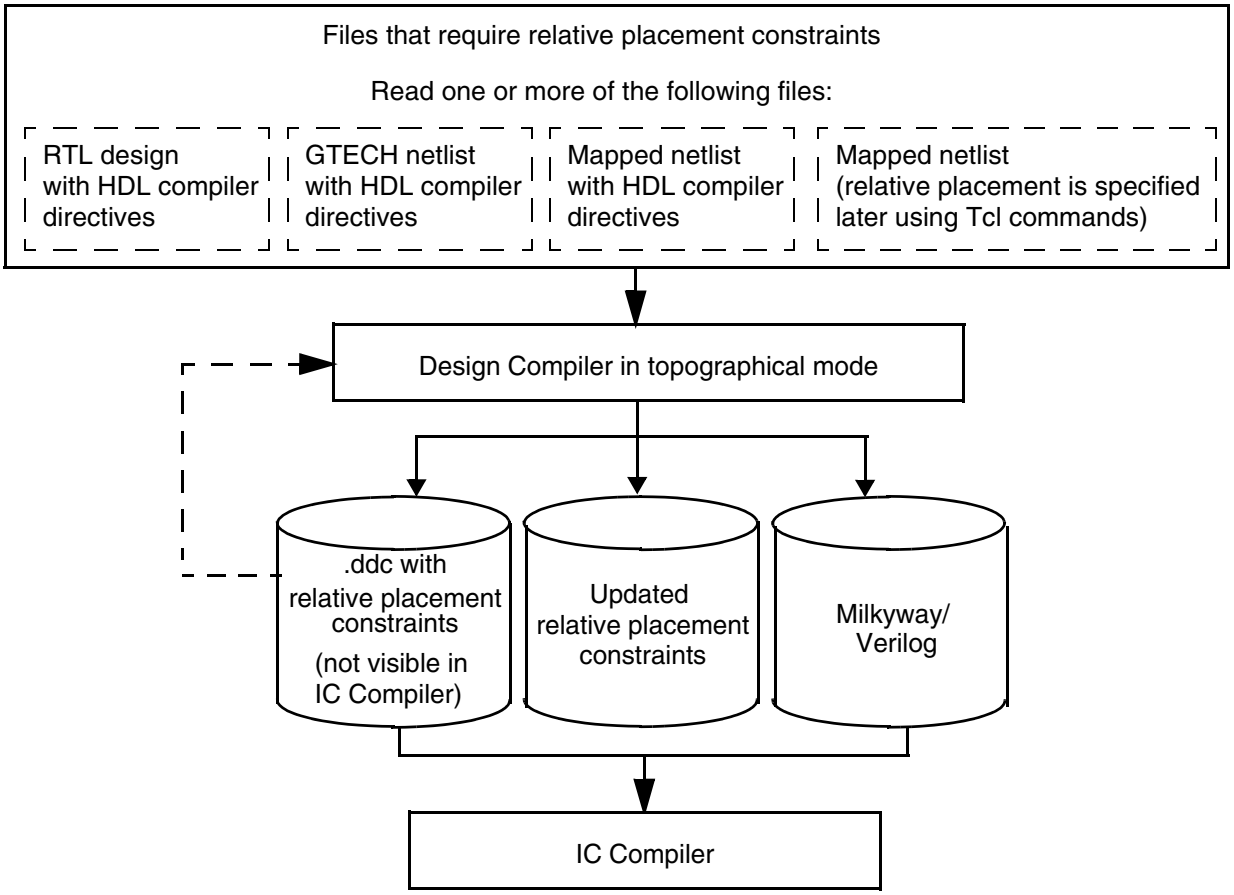
   The Design Vision layout window allows you to visually verify that your floorplan is laid out according to your expectations. The layout view automatically displays floorplan constraints read in with `extract_physical_constraints` or read in with Tcl commands. You need to link all applicable designs and libraries to obtain an accurate floorplan.

   For more information about using the GUI to view physical constraints, see the “Viewing the Floorplan” topic in Design Vision Help.

7. Write out the relative placement constraints to a Tcl file by using the `write_rp_groups` command. You can import this file into IC Compiler either by sourcing it or by using the `import_designs` command with the `-rp_constraint` option.

Figure 10-12 shows the relative placement flow in topographical mode.

Figure 10-12 Relative Placement Flow



Keep the following points in mind when you use relative placement:

- Topographical mode automatically places a `size_only` attribute on the relative placement cells to preserve the relative placement structure.
- Relative placement constraints are handled appropriately and preserved by using the `uniquify` and `ungroup` commands.
- Make sure that relative placement is applicable to your design. A design can contain both structured and unstructured elements. Some designs such as datapaths and pipelined designs are more appropriate for structured placement. Specifying relative placement constraints for cells that would be placed better by the tool can deliver poor results.
- Relative placement constraints are kept in the .ddc file for subsequent topographical mode sessions. These relative placement constraints are visible in Design Compiler in topographical mode only.

These relative placement constraints are not automatically transferred to IC Compiler. You can use the `write_rp_groups` command to write out the relative placement constraints to a Tcl script that can be read into IC Compiler. For information about using relative placement in IC Compiler, see the *IC Compiler Implementation User Guide*.

Creating Relative Placement Using HDL Compiler Directives

Design Compiler topographical mode supports relative placement information embedded within the Verilog or VHDL description. This capability is enabled by HDL compiler directives that can specify and modify relative placement information. Using these compiler directives to specify relative placement increases design flexibility and simplifies relative placement because you no longer need to update the location of many cells in the design.

Using the embedded HDL compiler directives, you can place relative placement constraints in an RTL design, a GTECH netlist, or a mapped netlist. For information about specifying relative placement in Verilog and VHDL using HDL compiler directives, including guidelines and restrictions, see the *HDL Compiler for VHDL User Guide* and the *HDL Compiler for Verilog User Guide*.

Summary of Relative Placement Tcl Commands

You can specify relative placement constraints by using a dedicated set of Tcl commands, similar to the commands in IC Compiler. Table 10-11 summarizes the Tcl commands available in Design Compiler topographical mode for relative placement. The sections that follow describe how to use these commands.

Table 10-11 Summary of Relative Placement Tcl Commands

Command	Description
`create_rp_group`	Creates new relative placement groups.
`add_to_rp_group`	Adds items to relative placement groups.
`set_rp_group_options`	Sets relative placement group attributes.
`report_rp_group_options`	Reports attributes for relative placement groups.
`get_rp_groups`	Creates a collection of relative placement groups that match certain criteria.
`write_rp_groups`	Writes out relative placement information for specified groups.
`all_rp_groups`	Returns a collection of specified relative placement groups and all subgroups in their hierarchy.

Table 10-11 Summary of Relative Placement Tcl Commands (Continued)

Command	Description
`all_rp_hierarchicals`	Returns a collection of hierarchical relative placement groups that are ancestors of specified groups.
`all_rp_inclusions`	Returns a collection of hierarchical relative placement groups that include specified groups.
`all_rp_instantiations`	Returns a collection of hierarchical relative placement groups that instantiate specified groups.
`all_rp_references`	Returns a collection of relative placement groups that contain specified cells (either leaf cells or hierarchical cells that contain instantiated relative placement groups).
`check_rp_groups`	Checks relative placement constraints and reports failures.
`remove_rp_groups`	Removes a list of relative placement groups.
`remove_rp_group_options`	Reports attributes for the specified relative placement groups.
`remove_from_rp_group`	Removes an item (cell, relative placement group, or keepout) from the specified relative placement groups.
`rp_group_inclusions`	Returns collections for directly embedded included groups (added to a group by using the `add_to_rp_group -hierarchy` command) in all or specified groups.
`rp_group_instantiations`	Returns collections for directly embedded instantiated groups (added to a group by using the `add_to_rp_group -hierarchy -instance` command) in all or specified groups.
`rp_group_references`	Returns collections for directly embedded leaf cells (added to a group by using the `add_to_rp_group -leaf` command), directly embedded included cells that contain hierarchically instantiated cells (added to the included group by using the `add_to_rp_group -hierarchy -instance` command), or both in all or specified relative placement groups.

Creating Relative Placement Groups

A relative placement group is an association of cells, other groups, and keepouts. A group is defined by the number of rows and columns it uses. To create a relative placement group in Design Compiler topographical mode, use the `create_rp_group` command. Topographical mode creates a relative placement group named *design_name*::*group_name*, where

design_name is the design specified by the `-design` option; if you do not use the `-design` option, the tool uses the current design. You must use this name or a collection of relative placement groups when referring to this group in other relative placement commands.

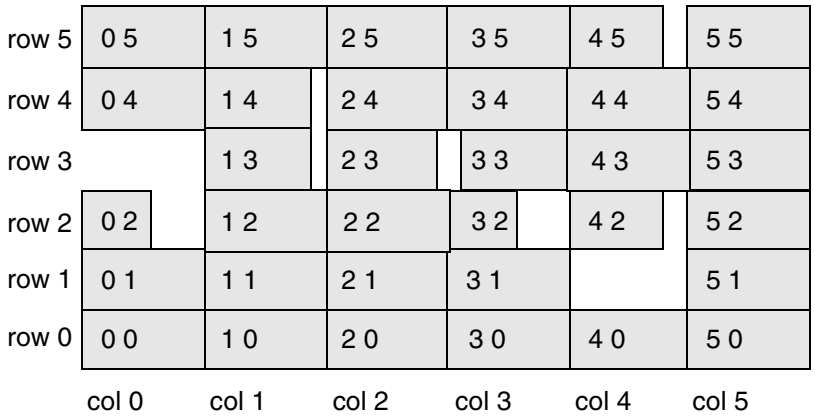
If you do not specify any options, the tool creates a relative placement group that has one column and one row. The group will not contain any objects. To add objects (leaf cells, relative placement groups, or keepouts) to a relative placement group, use the `add_to_rp_group` command, which is described in Adding Objects to a Group.

For example, to create a group named designA::rp1, having six columns and six rows, enter

```
create_rp_group rp1 -design designA -columns 6-rows 6
```

Figure 10-13 shows the positions of columns and rows in a relative placement group.

Figure 10-13 Relative Placement Column and Row Positions



In this figure,

- Columns count from column 0 (the leftmost column).
- Rows count from row 0 (the bottom row).
- The width of a column is the width of the widest cell in that column.
- The height of a row is determined by the height of the tallest cell in that row.
- It is not necessary to use all positions in the structure. For example, in this figure, positions 0 3 (column 0, row 3) and 4 1 (column 4, row 1) are not used.

Table 10-12 describes some options you can use with the `create_rp_group` command.

Table 10-12 Using the create_rp_group Command Options

To do this	Use this option
Specify the anchor location.	`-x_offset` `-y_offset`
Specify the type of alignment used by the group.	`-alignment`
Specify the group alignment pin.	`-pin_align_name`
Specify the utilization percentage (default is 100 percent).	`-utilization`
Ignore this relative placement group.	`-ignore`
Apply compression in the horizontal direction.	`-compress`

Anchoring Relative Placement Groups

By default, topographical mode can place a relative placement group anywhere within the core area. You can control the placement of a top-level relative placement group by anchoring it.

To anchor a relative placement group, use the `create_rp_group` or the `set_rp_group_options` command with the `-x_offset` and `-y_offset` options. The offset values are float values, in microns, relative to the chip's origin.

If you specify both the x- and y-coordinates, the group is anchored at that location. If you specify only one coordinate, IC Compiler can determine the placement by sliding the group along the unspecified coordinate.

For example, to specify a relative placement group anchored at (100, 100), enter the following command:

```
create_rp_group misc1 -design block1 \
           -columns 3 -rows 10 -x_offset 100 -y_offset 100
```

Applying Compression to Relative Placement Groups

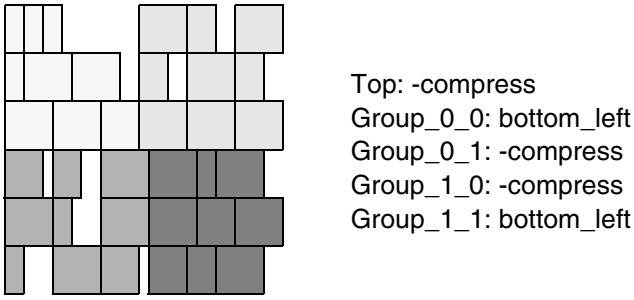
You can apply compression to a relative placement group in the horizontal direction during placement by using the `-compress` option with the `create_rp_group` or `set_rp_group_options` command. Setting this option enables bit-stack placement that places each row of a relative placement group without any gaps between leaf cells, lower-level hierarchical relative placement groups, or keepouts. Note that column alignment is not maintained when you use compression.

If you specify both the `-utilization` and `-compress` options, the utilization constraints are observed with gaps between leaf elements in a relative placement row. The `-compress` option does not propagate from a parent group to child groups. To disable relative placement with compression, use the `remove_rp_group_options -compress` command.

Supporting Compression with Mixed Alignment

Relative placement groups with alignment, such as bottom left, bottom right, or pin alignment, and the relative placement group that is created by using `-compress` can be placed on the same top-level groups as shown in Figure 10-14. The individual groups of the top level are aligned with compression only if you specify the `-compress` option. Note that the compression specified at the top level does not propagate to the child groups. The default alignment of the top-level groups is bottom left and the `-compress` option is disabled by default.

Figure 10-14 Compression of Relative Placement Groups with Mixed Alignment



Specifying Alignment

To specify the default alignment method to use when placing leaf cells and relative placement groups, use the `-alignment` option with the `create_rp_group` or `set_rp_group_options` command.

```
set_rp_group_options -alignment bottom-right [get_rp_groups *]
```

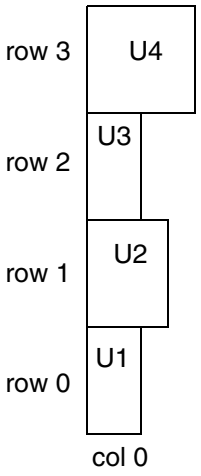
Controlling the cell alignment can improve the timing and routability of your design. You can specify a bottom-left, bottom-right or bottom-pin alignment. If you do not specify an option, the tool uses a bottom-left alignment.

The script in Example 10-37 defines a relative placement group that is bottom-left aligned. The resulting structure is shown in Figure 10-15.

Example 10-37 Definition for Bottom-Left-Aligned Relative Placement Group

```
create_rp_group rp1 -design pair_design -columns 1 -rows 4
   add_to_rp_group pair_design::rp1 -leaf U1 -column 0 -row 0
   add_to_rp_group pair_design::rp1 -leaf U2 -column 0 -row 1
   add_to_rp_group pair_design::rp1 -leaf U3 -column 0 -row 2
   add_to_rp_group pair_design::rp1 -leaf U4 -column 0 -row 3
```

Figure 10-15 Bottom-Left-Aligned Relative Placement Group



To align a group by pin location, use the `-alignment bottom-pin` and `-pin_align_name` options of the `create_rp_group` or `set_rp_group_options` command.

```
set_rp_group_options -alignment bottom-pin
   -pin_align_name align_pin
```

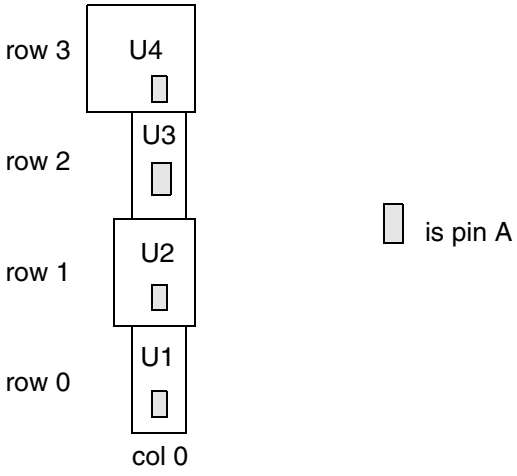
Topographical mode looks for the specified alignment pin in each cell in the column. If the alignment pin exists in a cell, the cell is aligned by use of the pin location. If the specified alignment pin does not exist in a cell, the cell is aligned at the bottom-left corner and the tool generates an information message. If the specified alignment pin does not exist in any cell in the column, the tool generates a warning message.

The script in Example 10-38 defines a relative placement group that is aligned by pin A. The resulting structure is shown in Figure 10-16.

Example 10-38 Definition for Relative Placement Group Aligned by Pins

```
create_rp_group rp1 -design pair_design -columns 1 -rows 4 -pin_align_name A
   add_to_rp_group pair_design::rp1 -leaf U1 -column 0 -row 0
   add_to_rp_group pair_design::rp1 -leaf U2 -column 0 -row 1
   add_to_rp_group pair_design::rp1 -leaf U3 -column 0 -row 2
   add_to_rp_group pair_design::rp1 -leaf U4 -column 0 -row 3
```

Figure 10-16 Relative Placement Group Aligned by Pins



When you specify an alignment pin for a group, the pin applies to all cells in the group. You can override the group alignment pin for specific cells in the group by specifying the `-pin_align_name` option when you use the `add_to_rp_group` command to add the cells to the group.

Adding Objects to a Group

You can add leaf cells, other relative placement groups, and keepouts to relative placement groups (created with the `create_rp_group` command). You use the `add_to_rp_group` command to add objects.

When you add an object to a relative placement group, keep the following points in mind:

- The relative placement group to which you are adding the object must exist.
- The object must be added to an empty location in the relative placement group.

Adding Leaf Cells

To add a leaf cell to a relative placement group, use the `add_to_rp_group` command. In a relative placement group, a leaf cell can occupy multiple column positions or multiple row positions, which is known as leaf cell straddling. You can create a more compact relative placement group by straddling leaf cells. To define straddling, you specify multiple column or row positions by using the `-num_columns` or `-num_rows` options respectively. If you do not specify these options, the default is 1. For example, to create a leaf cell of two columns and one row, enter

```
add_to_rp_group rp_group_name -leaf cell_name \
-column 0 -num_columns 2 -row 0 -num_rows 1
```

You should not place a relative placement keepout at the same location of a straddling leaf cell. In addition, straddling is for leaf cells only, but not for hierarchical groups or keepouts.

Note:

You should not apply compression to a straddling leaf cell that has either multiple column positions, multiple row positions, or both. You can apply right alignment or pin alignment to a straddling leaf cell with multiple row positions, but not to a cell with multiple column positions.

Include the `-orientation` option with a list of possible orientations when you add the cells to the group with the `add_to_rp_group` command.

Aligning Leaf Cells Within a Column

You can align the leaf cells in a column of a relative placement group by using the following alignment methods:

- Bottom left (default)
- Bottom right
- Pin alignment

Controlling the cell alignment can improve the timing and routability of your design.

Aligning by Bottom-Left Corners

To align the leaf cells by aligning the bottom-left corners, use the `-alignment bottom-left` option with the `create_rp_group` command or the `set_rp_group_options` command:

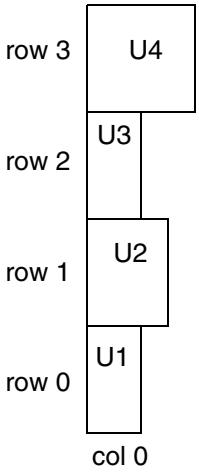
```
set_rp_group_options -alignment bottom-left [get_rp_groups *]
```

The script in Example 10-39 defines a relative placement group that is bottom-left aligned. The resulting structure is shown in Figure 10-17.

Example 10-39 Definition for Bottom-Left Aligned Relative Placement Group

```
create_rp_group rp1 -design pair_design -columns 1 -rows 4
   add_to_rp_group pair_design::rp1 -leaf U1 -column 0 -row 0
   add_to_rp_group pair_design::rp1 -leaf U2 -column 0 -row 1
   add_to_rp_group pair_design::rp1 -leaf U3 -column 0 -row 2
   add_to_rp_group pair_design::rp1 -leaf U4 -column 0 -row 3
```

Figure 10-17 Bottom-Left-Aligned Relative Placement Group



Aligning by Bottom-Right Corners

To align a group by aligning the bottom-right corners, use the `-alignment bottom-right` option with the `create_rp_group` command or the `set_rp_group_options` command:

```
set_rp_group_options -alignment bottom-right \
   [get_rp_groups *]
```

Note:

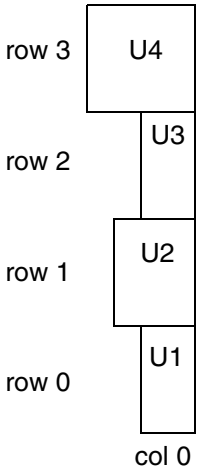
For hierarchical relative placement groups, the bottom-right alignment does not propagate through the hierarchy.

The script in Example 10-40 defines a relative placement group that is bottom-right aligned. The resulting structure is shown in Figure 10-18.

Example 10-40 Definition for Bottom-Right Aligned Relative Placement Group

```
create_rp_group rp1 -design pair_design -columns 1 -rows 4 \
   -alignment bottom-right
   add_to_rp_group pair_design::rp1 -leaf U1 -column 0 -row 0
   add_to_rp_group pair_design::rp1 -leaf U2 -column 0 -row 1
   add_to_rp_group pair_design::rp1 -leaf U3 -column 0 -row 2
   add_to_rp_group pair_design::rp1 -leaf U4 -column 0 -row 3
```

Figure 10-18 Bottom-Right Aligned Relative Placement Group



Aligning by Pin Location

To align a group by pin location, use the `-alignment bottom-pin` and `-pin_align_name` options of the `create_rp_group` or `set_rp_group_options` command.

```
set_rp_group_options -alignment bottom-pin \
   -pin_align_name align_pin
```

Design Compiler looks for the specified alignment pin in each cell in the column. If the alignment pin exists in a cell, the cell is aligned by use of the pin location. If the specified alignment pin does not exist in a cell, the cell is aligned at the bottom-left corner and Design Compiler generates an information message. If the specified alignment pin does not exist in any cell in the column, Design Compiler generates a warning message.

If you specify both pin alignment and cell orientation, Design Compiler resolves potential conflicts as follows:

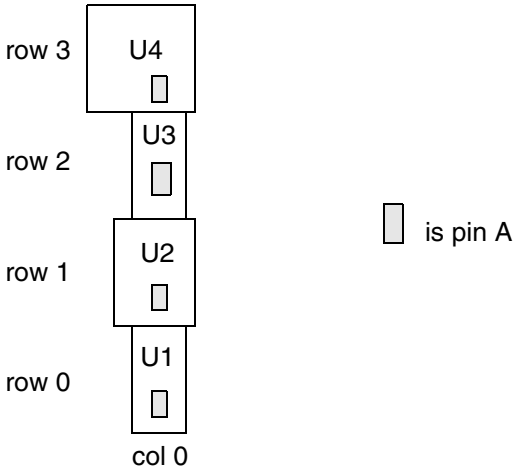
- User specifications for cell orientation take precedence over the pin alignment done by Design Compiler.
- Pin alignment done by Design Compiler takes precedence over the cell orientation optimization done by Design Compiler.

The script in Example 10-41 defines a relative placement group that is aligned by pin A. The resulting structure is shown in Figure 10-19.

Example 10-41 Definition for Relative Placement Group Aligned by Pins

```
create_rp_group rp1 -design pair_design -columns 1 -rows 4 -pin_align_name A
   add_to_rp_group pair_design::rp1 -leaf U1 -column 0 -row 0
   add_to_rp_group pair_design::rp1 -leaf U2 -column 0 -row 1
   add_to_rp_group pair_design::rp1 -leaf U3 -column 0 -row 2
   add_to_rp_group pair_design::rp1 -leaf U4 -column 0 -row 3
```

Figure 10-19 Relative Placement Group Aligned by Pins



When you specify an alignment pin for a group, the pin applies to all cells in the group. You can override the group alignment pin for specific cells in the group by specifying the `-pin_align_name` option when you use the `add_to_rp_group` command to add the cells to the group.

Note:

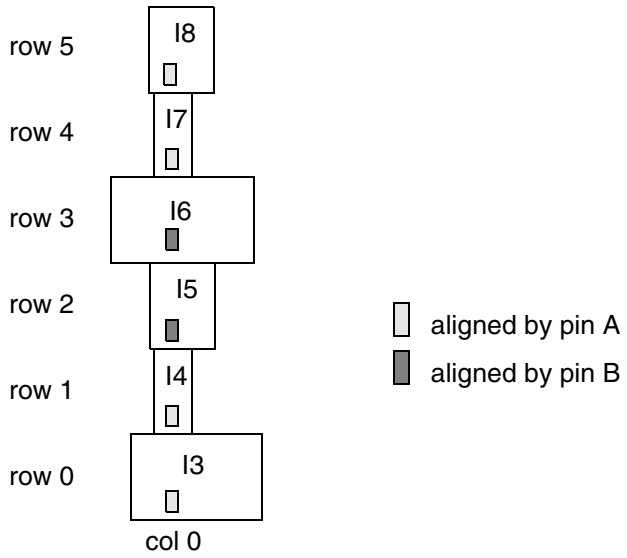
You cannot specify a cell-specific alignment pin when you add a leaf cell from the relative placement hierarchy browser.

The script in Example 10-42 defines relative placement group misc1, which uses pin A as the group alignment pin; however, instances I5 and I6 use pin B as their alignment pin, rather than the group alignment pin. The resulting structure is shown in Figure 10-20.

Example 10-42 Definition for Aligning a Group and Leaf Cells by Pins

```
create_rp_group misc1 -design block1 -columns 3 -rows 10 \
   -pin_align_name A
   add_to_rp_group block1::misc1 -leaf I3 -column 0 -row 0
   add_to_rp_group block1::misc1 -leaf I4 -column 0 -row 1
   add_to_rp_group block1::misc1 -leaf I5 -column 0 -row 2 \
      -pin_align_name B
   add_to_rp_group block1::misc1 -leaf I6 -column 0 -row 3 \
      -pin_align_name B
   add_to_rp_group block1::misc1 -leaf I7 -column 0 -row 4
   add_to_rp_group block1::misc1 -leaf I8 -column 0 -row 5
```

Figure 10-20 Relative Placement Group Aligned by Pins



Adding Relative Placement Groups

Hierarchical relative placement allows relative placement groups to be embedded within other relative placement groups. The embedded groups then are handled similarly to leaf cells. You can use hierarchical relative placement to simplify the expression of relative placement constraints. With hierarchical relative placement, you do not need to provide relative placement information multiple times for a recurring pattern.

There are two methods for adding a relative placement group to a hierarchical group. You can include the group or instantiate the group. You use the `add_to_rp_group` command for both methods:

- Include the group

  If the relative placement group to be added is in the same design as its parent group, it is an included group. You can include groups in either flat or hierarchical designs. When you include a relative placement group in a hierarchical group, it is as if the included group is directly embedded within its parent group. An included group can be used only in a group of the same design and only one time. However, a group that contains an included group can be further included in another group in the same design or can be instantiated in a group of a different design.

- Instantiate the group

  If the relative placement group to be added is in an instance of a subdesign of its parent group, it is an instantiated group. You can instantiate groups only in hierarchical designs.

The group specified in the `-hierarchy` option must be defined in the reference design of the instance specified in the `-instance` option. In addition, the specified instance must be in the

same design as the hierarchical group in which you are instantiating the specified group. Using an instantiated group is a useful way to replicate relative placement information across multiple instances of a design and to create relative placement relationships between those instances.

The script in Example 10-43 creates a hierarchical group (rp2) that contains three instances of group rp1. Group rp1 is in the design pair_design and includes leaf cells U1 and U2. Group rp2 is a hierarchical group in the design mid_design that instantiates group rp1 three times (mid_design must contain at least three instances of pair_design). Group rp2 is treated as a leaf cell. You can instantiate rp2 multiple times and in multiple places, up to the number of times mid_design is instantiated in your netlist.

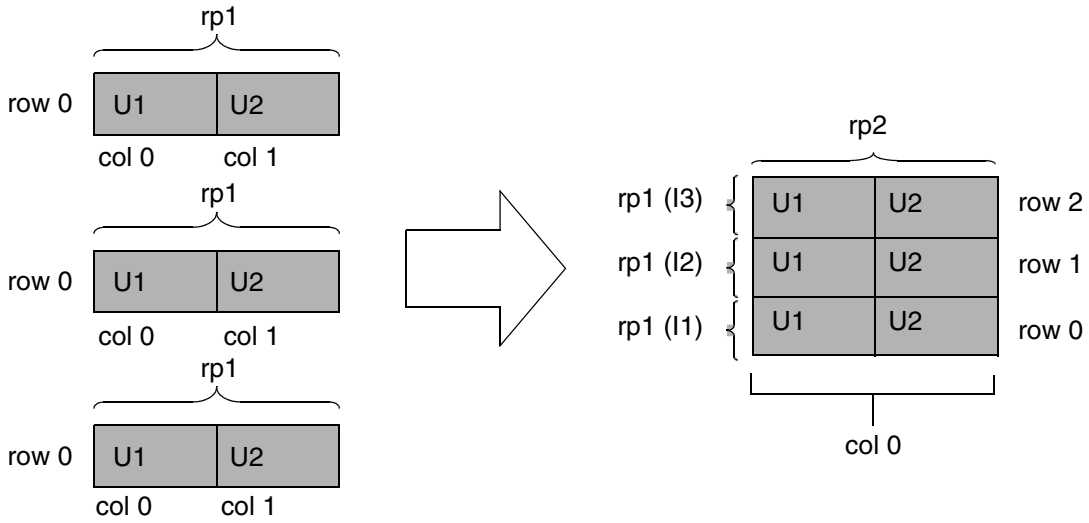
The resulting hierarchical relative placement group is shown in Figure 10-21.

Example 10-43 Instantiating Groups in a Hierarchical Group

```
create_rp_group rp1 -design pair_design -columns 2 -rows 1
   add_to_rp_group pair_design::rp1 -leaf U1 -column 0 -row 0
   add_to_rp_group pair_design::rp1 -leaf U2 -column 1 -row 0

create_rp_group rp2 -design mid_design -columns 1 -rows 3
   add_to_rp_group mid_design::rp2 \
      -hierarchy pair_design::rp1 -instance I1 -column 0 -row 0
   add_to_rp_group mid_design::rp2 \
      -hierarchy pair_design::rp1 -instance I2 -column 0 -row 1
   add_to_rp_group mid_design::rp2 \
      -hierarchy pair_design::rp1 -instance I3 -column 0 -row 2
```

Figure 10-21 Instantiating Groups in a Hierarchical Group



Adding Keepouts

To add a keepout to a relative placement group, use the `add_to_rp_group` command. For example, to create a keepout named gap1, enter

```
add_to_rp_group TOP::misc -keepout gap1 \
                -column 0 -row 2 -width 15 -height 1
```

where `TOP::misc` is the group list.

Querying Relative Placement Groups

Table 10-13 lists the commands available for querying particular types of relative placement groups to annotate, edit, and view.

Table 10-13 Commands for Querying Relative Placement Groups

To return collections for this	Use this command
Relative placement groups that match certain criteria	`get_rp_groups`
All or specified relative placement groups and the included and instantiated groups they contain in their hierarchies	`all_rp_groups`
All or specified relative placement groups that contain included or instantiated groups	`all_rp_hierarchicals`
All or specified relative placement groups that contain included groups	`all_rp_inclusions`
All or specified relative placement groups that contain instantiated groups	`all_rp_instantiations`
All or specified relative placement groups that directly embed leaf cells (added to a group by `add_to_rp_group -leaf`) or instantiated groups (added to a group by `add_to_rp_group -hierarchy -instance`) in all or specified relative placement groups in a specified design or the current design	`all_rp_references`

For example, to create a collection of relative placement groups that start with the letter *g* in a design that starts with the letter *r*, enter

```
get_rp_groups r*::g*
{ripple::grp_ripple}
```

To set the utilization to 95 percent for all relative placement groups, enter

```
set_rp_group_options [all_rp_groups] -utilization 0.95
```

Checking Relative Placement Constraints

To check whether the relative placement constraints have been met, run the `check_rp_groups` command. The command reports relative placement failures such as the following:

```
RPGP-028: The height ‘%f’ of the RP group ‘%s’ is more than the height
‘%f’ of the core area.
RPGP-035 (warning) Relative placement leaf cell data may have been lost.
RPGP-0348: All RP cells in cluster ‘%s’ are not within same voltage
region;
ignoring voltage region placement.
```

The `check_rp_groups` command checks for the following failures:

- The relative placement group cannot be placed as a whole.
- The height or width of the relative placement group is greater than the height or width of the core area.
- The user-specified orientation cannot be met.

If a failure prevents the group from being placed as a single entity, as defined by the relative placement constraints, the failure is considered critical. If the failure does not prevent placement but causes the relative placement constraints to be violated, the failure is considered noncritical. The generated report contains separate sections for critical and noncritical failures.

Note that the `check_rp_groups` command does not check for the failure: the keepouts are not created correctly in the relative placement group.

You can check all relative placement groups by specifying the `-all` option or you can specify which relative placement groups to check. By default, the report is output to the screen. To save the generated report to a file, specify the file name by using the `-output` option.

For example, to check the relative placement constraints for groups compare17::seg7 and compare17::rp_group3 and to save the output in a file called rp_failures.log, run the following command:

```
check_rp_groups "compare17::seg7 compare17::rp_group3" \
            -output rp_failures.log
```

The generated report is similar to this:

```
****************************************
Report : The RP groups, which could not be placed.
...
No. of RP groups:1
****************************************
RP GROUP: compare17::seg7
-------------------------------
ERROR: Could not get clean area for placing RP group compare17::seg7.


****************************************
Report : The RP groups, not meeting all constraints but placed.
...
No. of RP groups:1
****************************************
RP GROUP: compare17::rp_group_3
-------------------------------
WARNING: Could not set user specified orientation for cell u[8].
```

To display a detailed report, use the `-verbose` option, as shown:

```
check_rp_groups -all -verbose
```

The `-verbose` option reports all possible relative placement failures in the design.

Saving Relative Placement Information

You can use the `write_rp_groups` command to write out relative placement constraints.

For example, to save all the relative placement groups to disk, remove the information from the design, and then re-create the information about the design, enter

```
get_rp_groups
{mul::grp_mul ripple::grp_ripple example3::top_group}
write_rp_groups -all -output my_groups.tcl
1
remove_rp_groups -all -quiet
1
get_rp_groups
Error: Can't find objects matching '*'. (UID-109)
source my_groups.tcl
{example3::top_group}
get_rp_groups
{example3::top_group ripple::grp_ripple mul::grp_mul}
```

By default, the `write_rp_groups` command writes out commands for creating the specified relative placement groups and to add leaf cells, hierarchical groups, and keepouts to these groups. The commands for generating subgroups within hierarchical groups are not written. The `write_rp_groups` command writes out updated relative placement constraints and includes names changes from uniquification or ungrouping.

If you specified multiple column positions or multiple row positions for a cell using the `-num_columns` or `-num_rows` options with the `add_to_rp_group` command, the `write_rp_groups` command writes out the multiple-location cell as well.

You can modify the default behavior of the `write_rp_groups` command by using the options described in Table 10-14.

Table 10-14 Using the write_rp_groups Command Options

To do this	Use this option with the write_rp_groups command
Write all the relative placement groups within the hierarchy of the relative placement groups. If you omit this option, subgroups are not written.	`-hierarchy`
Write only `create_rp_group` commands to the script.	`-create`
Write only `add_to_rp_group -leaf` commands to the script.	`-leaf`
Write only `create_rp_group -keepout` commands to the script.	`-keepout`
Write only `create_rp_group -hierarchy -instance` commands to the script.	`-instance`
Write only `create_rp_group -hierarchy` commands to the script.	`-include`

Removing Relative Placement Group Attributes

To remove relative placement groups, run the `remove_rp_groups` command. You can remove all relative placement groups (by specifying the `-all` option), or you can specify which relative placement groups to remove. If you specify a list of relative placement groups, only the specified groups (and not groups included or instantiated within the specified group) are removed. To remove the included and instantiated groups of the specified groups, you must specify `-hierarchy`.

For example, to remove the relative placement group named grp_ripple and confirm its removal, enter

```
get_rp_groups
{mul::grp_mul ripple::grp_ripple example3::top_group}
remove_rp_groups ripple::grp_ripple
Removing rp group 'ripple::grp_ripple'
1
```

```
get_rp_groups *grp_ripple
Error: Can't find object 'grp_ripple'. (UID-109)
remove_rp_groups -all
Removing rp group 'mul::grp_mul'
Removing rp group 'example3::top_group'
1
```

To remove relative placement group attributes, run the `remove_rp_group_options` command. You must specify the group name and at least one option; otherwise, this command has no effect.

For example, to remove the `x_offset` attribute for the block1::misc1 group, enter

```
remove_rp_group_options block1::misc1 -x_offset
{block1::misc1}
```

The command returns a collection of relative placement groups for which attributes have been changed. If no attributes change for any object, an empty string is returned.

To remove objects from a relative placement group, use the `remove_from_rp_group` command. You can remove leaf cells (`-leaf`), included groups (`-hierarchy`), instantiated groups (`-hierarchy -instance`), and keepouts (`-keepout`).

For example, to remove leaf cell carry_in_1 from grp_ripple, enter

```
remove_from_rp_group ripple::grp_ripple -leaf carry_in_1
1
```

If you specified multiple column positions or multiple row positions for a cell using the `-num_columns` or `-num_rows` options with the `add_to_rp_group` command, the `remove_from_rp_group` command removes the cell from all its locations.

Sample Script for a Relative Placement Flow

Example 10-44 is a sample script for running a relative placement flow.

Example 10-44 Sample Script for the Relative Placement Flow

```
# Set library and design paths
source setup.tcl
read_ddc design_name
current_design top
link

# Create relative placement constraints
create_rp_group grp_ripple -design ripple -rows 8
...
add_to_rp_group ripple::grp_ripple -leaf carry_in_1
...

# Apply design constraints
source constraints.tcl
```

```
# Read physical constraints
extract_physical_constraints top.def

# Check relative placement
check_rp_groups

# Perform synthesis
compile_ultra -scan

# Write out the netlist and relative placement
change_names -rules verilog
write_file -format verilog -hierarchical -output ripple.v
write_rp_groups -all -output ripple.rp.tcl
```

Placement Options: Magnet Placement

Magnet placement improves timing and congestion correlation between Design Compiler and IC Compiler. Magnet placement moves standard cells closer to objects specified as magnets. You can use magnet placement with any netlist that is fully mapped. Use the `magnet_placement` command to enable magnet placement if `magnet_placement` was used in IC Compiler.

The tool will only perform magnet placement if the standard cells are placed. If you use the `magnet_placement` command before the standard cells are placed, the tool will not pull any objects to the specified magnet. The tool will issue a warning for this condition.

To perform magnet placement, use the `magnet_placement` command with a specification of the magnets and options for any special functions you need to perform. Table 10-15 lists options to use for various tasks. Specify magnet placement as follows:

`magnet_placement [`*`options`*`] `*`magnet_objects`*

Table 10-15 Using the magnet_placement Command Options

To do this	Use this option
Enable the movement of fixed cells and display a warning that lists the fixed cells to be moved	`-move_fixed`
Fix cells in their locations after magnet placement	`-mark_fixed`
Specify the number of logic levels from the magnet that should be checked for magnet placement	`-logical_level` *number*
Prevent movement of buffers and inverters	`-exclude_buffers`

Table 10-15 Using the magnet_placement Command Options (Continued)

To do this	Use this option
Skip specified cells when pulling cells toward the magnet object	`-exclude_cells` *`object_list`*
Prevent cells from being placed over soft blockages	`-avoid_soft_blockages`
Prevent placement beyond sequential cells	`-stop_by_sequential_cells`
Pull only those objects on the timing path between the magnet object and a specified end object. Note: This option is mutually exclusive with the `-logical_level` option.	`-stop_points` `object_list`

Use the `magnet_placement_fanout_limit` variable to specify the fanout limit. If the fanout of a net exceeds the specified limit, the magnet_placement command does not pull cells of the net toward the magnet objects. The default setting is 1000.

By default, the magnet placement operation is terminated before the sequential cell when the `-stop_by_sequential_cells` option is used with the magnet_placement command. If you want to terminate the magnet placement operation after the sequential cell, set the `magnet_placement_stop_after_seq_cell` variable to true. The default is `false`.

Magnet placement allows cells to be overlapped by default. To prevent overlapping of cells, you can set the `magnet_placement_disable_overlap` variable to `true`.

To return a collection of cells that can be moved with magnet placement, use the `get_magnet_cells` command with the options you need.

`get_magnet_cells [`*`options`*`] `*`magnet_list`*

Table 10-16 lists options to use for various tasks.

Resetting Physical Constraints

To reset all physical constraints, use the `reset_physical_constraints` command. Use this command to clear all existing physical constraints before reading in a new or modified floorplan.

Saving Physical Constraints Using the write_floorplan Command

You use the `write_floorplan` command to save the floorplan information. The write_floorplan command writes out individual floorplan commands relative to the top of the design, regardless of the current instance. The output is a command script file that contains floorplan information such as bounds, placement blockages, route guides, plan groups, and voltage areas.

When writing out the floorplan from Design Compiler, it is generally recommended that you use the `-all` option to write out the complete floorplan information. The `write_floorplan` command also provides options that allow you to write out partial floorplan information.

If you reuse this output to re-create the floorplan, you must read it in from the top level of the design with the `read_floorplan` command. You can also use the `source` command to import the floorplan information. However, the source command reports errors and warnings that are not applicable to Design Compiler in topographical mode. The `read_floorplan` command removes these unnecessary errors and warnings. In addition, you need to enable fuzzy name matching manually when you use the `source` command. The `read_floorplan` command automatically enables fuzzy name matching.

Reporting Physical Constraints

To report any physical constraints that are applied to the design, use the `report_physical_constraints` command. The command reports the following physical constraints:

- Die area
- Placement area
- Utilization
- Aspect ratio
- Rectilinear outline
- Port side
- Port location
- Cell location
- Placement blockage
- Wiring keepouts
- Vias
- Design via masters

- Voltage area
- Site rows
- Bounds
- Preroutes
- User shapes
- Routing tracks
- Keepout margins

To report the `create_net_shape` commands (preroutes), use the `-pre_route` option with the `report_physical_constraints` command. If you do not want the report to show site row information, use the `-no_site_row` option.

The following sections provide more information about reporting physical constraints.

Reporting Routing Tracks

You can use the `report_physical_constraints` command or the `report_track` command to report routing tracks. The following example shows the `report_physical_constraints` report format:

```
TRACK_NAME      LAYER      DIRECTION      COORDINATE      STEP      COUNT
USER_TRACK_1    METAL1       X            0.330           0.660     457
USER_TRACK_2    METAL1       Y            0.280           0.560     540
```

The `report_track` command reports the routing tracks for a specified layer or for all layers. The `-layer` *`layer`* option specifies which routing layer to report. You can specify a layer name, a layer number, or a collection containing one layer object. By default, Design Compiler reports the routing tracks on all layers.

You can use the `-dir` option to specify the routing direction. The valid values are either `X` or `Y`. By default, Design Compiler reports routing tracks in both directions.

To get a report similar to Example 10-45 that shows the metal layer, the metal direction, the starting point, the number of tracks, the metal pitch, and the origin of the attributes, run the `report_track` command without any options:

```
dc_shell-topo> report_track
```

In Example 10-45, all the attributes are defined from the DEF file, as indicated in the `Attr` column. However, the attributes could also be user defined (`usr`) or route defined (`rt`).

Example 10-45 Routing Track Report

```
****************************************
Report track
Design : frc_sys
```

```
Version: E-2010.12
Date   : Tue Oct 12 22:51:13 2010
****************************************
Layer          Direction     Start         Tracks    Pitch          Attr
-------------------------------------------------------------------------
Attributes :
         usr : User defined
         rt  : Route66 defined
         def : DEF defined
metal1             X         0.100          11577     0.200          def
metal1             Y         0.200          11575     0.200          def
metal2             Y         0.200          11575     0.200          def
```

Reporting Preroutes

You can report preroutes, including user shapes, net shapes, vias, via arrays, via cells, and preroute net types by using the `-pre_route` option with the `report_physical_constraints` command.

The following example shows the report output from the `report_physical_constraints -pre_route` command:

```
report_physical_constraints -pre_route
  ****************************************
  report_physical_constraints
  Version: F-2011.09-SP2
  Date: Fri Nov 18 09:56:44 2011
  ****************************************
  Design top
  Physical Nets Number: 3
  NAME           TYPE
  vdd            power
  vss            ground
  net1           signal
  1
```

Reporting Design Via Masters

You can use the `report_physical_constraints` command to report design via masters, as shown in the following example. The VIA MASTER section reports the via master definitions. The PREROUTES section includes the instantiation of vias using the via master definitions:

```
...

VIA MASTER total number: 35
INDEX NAME              RECT_NUM  LAYER RECTANGLES/VIA RULES
#0   via23_array1         6         MET2 { -0.28 -0.35 0.28 0.35 }
                                     MET3 { -0.28 -0.35 0.28 0.35 }
                                     VIA2 { -0.22 -0.22 -0.08 -0.08 }
                                     VIA2 { -0.22 0.08 -0.08 0.22 }
                                     VIA2 { 0.08 -0.22 0.22 -0.08 }
```

```
                                  VIA2 { 0.08 0.08 0.22 0.22 }

#1   via23_array2         6      MET2 { -0.28 -0.42 0.28 0.42 }
                                  MET3 { -0.28 -0.42 0.28 0.42 }
                                  VIA2 { -0.22 -0.22 -0.08 -0.08 }
                                  VIA2 { -0.22 0.08 -0.08 0.22 }
                                  VIA2 { 0.08 -0.22 0.22 -0.08 }
                                  VIA2 { 0.08 0.08 0.22 0.22 }
...

PREROUTES total number: 352486
TYPE    LAYER     NETS      ROUTE_TYPE           DATA_TYPE GEOMETRY ATTRIBUTE
via     via23_array1  vdd!  pg_strap                       at(375.5 438.5), N
via     via23_array2  vdd!  pg_strap                       at(487.5 438.5), N
...
```

Reporting Keepout Margins

You can use the `report_physical_constraints` command or the `report_keepout_margin` command to report keepout margins. The `report_physical_constraints` command reports keepout margins in the following format:

```
CELL_KEEPOUT_MARGIN   TYPE   MARGIN_VALUE
U542                  HARD      {10.000 10.000 50.000 50.000}
U543                  SOFT      {15.000 15.000 15.000 15.000}
```

The `report_keepout_margin` command reports keepout margins of a specified type for the specified cells in the design. Use the `-type` option to specify the type of keepout to be reported. The valid values are either `hard` or `soft`. By default, both soft and hard keepouts are reported for the cells or library cells.

You can use the `-parameters` option to report the following pin-count-based parameter values:

- `tracks_per_macro_pin`
- `min_padding_per_macro`
- `max_padding_per_macro`

The following example reports hard keepouts for cells in an object list named MY_CELL:

```
dc_shell-topo> report_keepout_margin -type hard MY_CELL
```

The following example reports all the pin-count-based parameters to be used during the calculation of pin-count-based keepout margins for macro cells:

```
dc_shell-topo> report_keepout_margin -parameters
```

Performing Automatic High-Fanout Synthesis

By default, Design Compiler topographical mode does automatic high-fanout synthesis. Design Compiler does not perform automatic high-fanout synthesis on any nets that are part of `dont_touch_network` and `ideal_network`. You can use the `set_ahfs_options` command to specify the constraints to be used when running automatic high-fanout synthesis (AHFS). After automatic high-fanout synthesis, Design Compiler sets a global design-based attribute that prevents IC Compiler from doing automatic high-fanout synthesis. If you need to enable automatic high-fanout synthesis in IC Compiler, for example, if you are changing any ideal or `dont_touch_network` settings, apply the `set_ahfs_options` command in the back-end script.

Test Synthesis in Topographical Mode

Topographical mode supports test synthesis. The flow is similar to the one in Design Compiler wire load mode except that the `insert_dft` command is used for stitch-only. Topographical mode supports basic scan and adaptive scan.

To perform scan insertion within topographical mode, use a script similar to the following:

```
dc_shell -topo
read_ddc top_elaborated.ddc
source top_constraint.sdc
source physical_constraints.tcl
compile_ultra -gate_clock -scan

## Provide DFT specifications
set_dft_signal ...

create_test_protocol
dft_drc
preview_dft
insert_dft
dft_drc

compile_ultra -incremental -scan
write_scan_def -output dft.scandef
```

For more information about scan insertion, see the DFT Compiler documentation.

Power Optimization in Topographical Mode

Design Compiler in topographical mode can perform power correlation. Power correlation is performed by estimating the clock tree power in Design Compiler.

Power prediction is performed if any gate-level power optimizations are enabled by the following commands:

- `set_dynamic_optimization`
- `set_leakage_optimization`
- `compile_ultra -gate_clock`

In addition, you can use the `set_power_prediction` command with the `-ct_references` option to specify clock tree references to improve correlation. The `set_power_prediction` command enables the tool to correlate post-synthesis power numbers with those after clock tree synthesis. To disable power prediction, set the `set_power_prediction` command to `false`.

The `report_power` command splits the power numbers into two categories: netlist power (based on cells already in the design) and estimated power (an estimate of the clock tree power).

To perform power correlation in topographical mode, run a script similar to the following:

```
read_ddc top_elaborated.ddc

source top_constraint.sdc
source physical_constraints.tcl

## Set power optimization constraints

set_leakage_optimization true
compile_ultra -gate_clock -scan
report_power
write_file -format ddc -output synthesized.ddc
```

For more information about power correlation, see the *Power Compiler User Guide*.

Multivoltage Designs

Design Compiler in topographical mode supports power optimization and power correlation for single voltage and multivoltage designs. For multivoltage designs, the subdesign instances (blocks) operate at different voltages. To reduce power consumption, multivoltage designs typically make use of power domains. The blocks of a power domain can be powered up and down, independent of the power state of other power domains (except where a relative always-on relationship exists between two power domains). In particular,

power domains can be defined and level shifter and isolation cells can be used as needed to adjust voltage differences between power domains and to isolate shut-down power domains.

A power domain is defined as a grouping of one or more hierarchical blocks in a design that share the following:

- Primary voltage states or voltage range (that is, the same operating voltage)
- Power net hookup requirements
- Power-down control and acknowledge signals (if any)
- Power switching style
- Same process, voltage, and temperature (PVT) operating condition values (all cells of the power domain except level shifters)
- Same set or subset of nonlinear delay model (NLDM) target libraries

Using IEEE™ 1801 Unified Power Format (UPF) Standard commands, you can specify the power domains of the design and the desired multivoltage implementation and verification strategies for each domain, including the domain's isolation cells, retention cells, and level shifters. During a compile operation, Design Compiler automatically inserts these power management cells according to the specified strategies. The same set of UPF commands can be used throughout the design, analysis, verification, and implementation flow.

Principal power domain commands are described in the *Power Compiler User Guide*.

Note:

Power domains are not voltage areas. A power domain is a grouping of logic hierarchies, whereas the corresponding voltage area is a physical placement area into which the cells of the power domain's hierarchies are placed. This correspondence is not automatic. You are responsible for correctly aligning the hierarchies to the voltage areas. You use the `create_voltage_area` command to set voltage areas.

Compile Flows in Topographical Mode

In addition to supporting a top-down `compile_ultra` flow, as described in Top-Down Compile in Optimizing the Design, Design Compiler topographical mode supports incremental and hierarchical flows:

- Performing an Incremental Compile

  The `-incremental` option of the `compile_ultra` command allows you to employ a second-pass, incremental compile strategy.

- Performing a Bottom-up (Hierarchical) Compile

Topographical mode supports a hierarchical flow if you need to address design and runtime challenges or use a divide and conquer synthesis approach. In topographical mode, the `-top` option of the `compile_ultra` command enables you to stitch compiled physical blocks into the top-level design.

Performing an Incremental Compile

The `-incremental` option of the `compile_ultra` command allows you to employ a second-pass, incremental compile strategy. The main goal for `compile_ultra -incremental` is to enable topographical-based optimization for post-topographical-based synthesis flows such as retiming, design-for-test (DFT), DFT MAX, and minor netlist edits. The primary focus in Design Compiler topographical mode is to maintain QoR correlation; therefore, only limited changes to the netlist can be made.

Use the incremental compile strategy to meet the following goals:

- Improve design QoR
- Fix the netlist after manual netlist edits or constraint changes
- Fix the netlist after various synthesis steps have been performed on the compiled design, for example, after `insert_dft` or register retiming
- Control design rule fixing by using the `-no_design_rule` or `-only_design_rule` option in combination with the `-incremental` option

Incremental compile supports adaptive retiming, that is, `compile_ultra -incremental -retime`.

Note that applying `compile_ultra -incremental` to a topographical netlist results in placement-based optimization only. This compile should not be thought of as an incremental mapping.

Note:

When you use the `insert_buffer` command and `remove_buffer` command described in Editing Designs, the `report_timing` command does not report placement-based timing for the edited cells. To update timing, run the `compile_ultra -incremental` command.

When using the `-incremental` option, keep the following in mind:

- Marking library cells with the `dont_use` attribute does not work for an incremental flow when it is applied to a topographical netlist. Make sure to apply any `set_dont_use` attributes before the first pass of a topographical-based synthesis.
- If you intend to use boundary optimization and scan insertion, apply them to the first pass of a topographical-based synthesis.

- Avoid significant constraint changes in the incremental pass.

  Note:
  Physical constraint changes are not supported.

Performing a Bottom-up (Hierarchical) Compile

In a hierarchical compile flow, you compile the subdesigns separately and then incorporate them in the top-level design. This is also known as a bottom-up flow.

The recommended strategy is a top-down compile flow. However, topographical mode supports a hierarchical flow or bottom-up flow if you need to address design and runtime challenges or use a divide-and-conquer synthesis approach. In the bottom-up strategy, individual subblocks are constrained and compiled separately. The compiled subblocks are then included in subsequent top-level synthesis. In topographical mode, the tool can read the following types of hierarchical blocks:

- Netlist generated in topographical mode
- Block abstractions generated in topographical mode or IC Compiler
- Interface logic models (ILMs) generated in topographical mode or IC Compiler

That is, you can compile the subblock in topographical mode and provide it to the top-level design as a full .ddc netlist, a block abstraction, or an ILM that was created in Design Compiler topographical mode. Alternatively, you can continue working on the subblock in IC Compiler to create a placed-and-routed block abstraction or ILM, which you can then provide to the top-level design in topographical mode. Timing and physical information of the subblock are propagated to the top-level for physical synthesis in topographical mode. In addition, you can provide the placement location for a subblock during top-level synthesis to maintain correlation with IC Compiler.

In the hierarchical or bottom-up flow, use the `compile_ultra -scan -gate_clock` command to perform top-level design integration. The tool automatically propagates block-level timing and placement to the top level and uses them to drive optimization. In addition, you can specify the placement location for the subblock. Top-level optimization is placement-aware and can be driven with the same physical constraints as your back-end tool.

The following sections describe the Design Compiler topographical mode hierarchical flow:

- Overview of Bottom-Up Compile
- Compiling the Subblock
- Compiling the Design at the Top Level

Overview of Bottom-Up Compile

In the bottom-up flow, first you compile the subblock in topographical mode and save the mapped subblock as a netlist in .ddc format. Then, you can continue working on the subblock (.ddc netlist) in IC Compiler to generate a block abstraction or interface logic model (ILM).

Alternatively, you can compile the subblock in topographical mode and save the mapped subblock as a block abstraction in .ddc format or an ILM in .ddc format. However, you cannot combine block abstractions and ILMs in the same flow.

Top-level synthesis can accept the following types of mapped subblocks:

- .ddc netlist synthesized in topographical mode
- Block abstraction created in topographical mode
- Block abstraction created in IC Compiler
- ILM created in topographical mode
- ILM created in IC Compiler

Note:
IC Compiler cannot accept Design Compiler ILMs or block abstractions.

Figure 10-22 provides an overview of the hierarchical flow for designs containing block abstractions. Figure 10-23 provides an overview of the hierarchical flow for designs containing ILMs.

Figure 10-22 Overview of the Hierarchical Flow for Designs Containing Block Abstractions

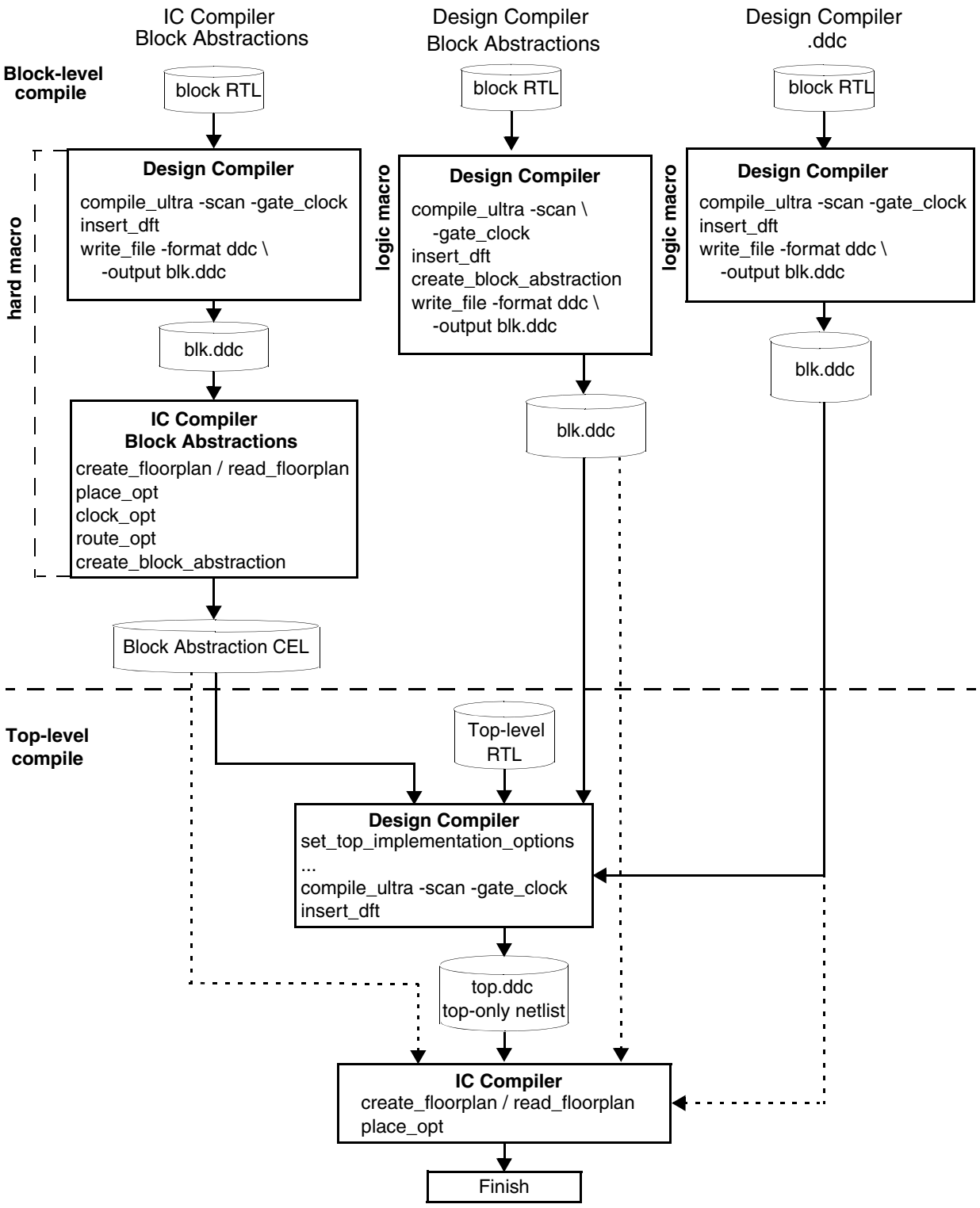
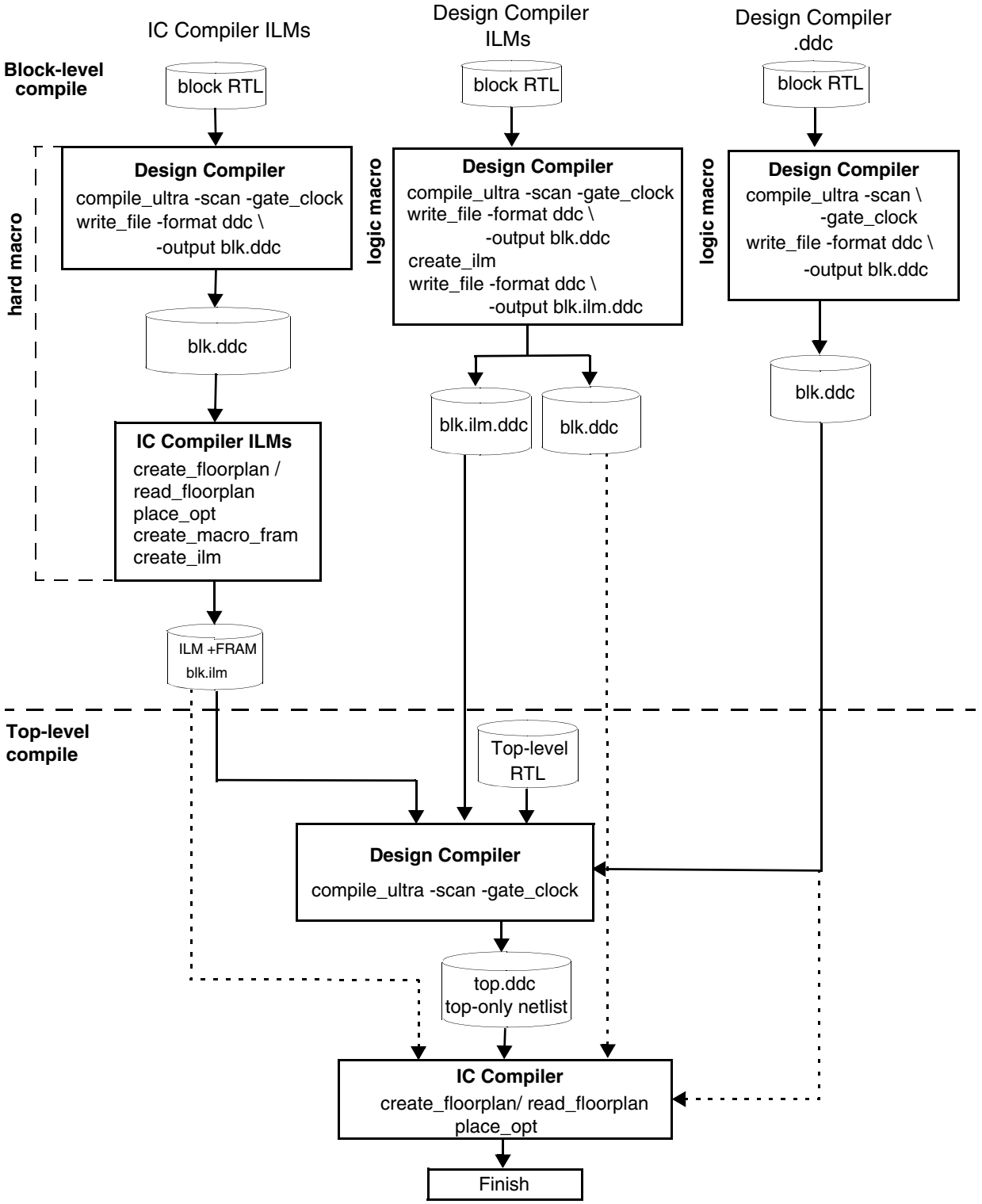


Figure 10-23 Overview of the Hierarchical Flow for Designs Containing ILMs



The bottom-up flow requires these main steps:

1. Compile the subdesigns independently. See Compiling the Subblock.
2. Read in the top-level design and any compiled subdesigns not already in memory; compile the top-level design. See Compiling the Design at the Top Level.

Compiling the Subblock

Compiling the subblock requires the following steps:

1. Specify the logic and physical libraries as described in Specifying Libraries.
2. Read in the subblock (Verilog, VHDL, netlist, or .ddc) and set the current design to the subblock.
3. Apply block-level timing constraints and power constraints.
4. (Optional) Provide physical constraints as described in Using Floorplan Physical Constraints.
5. (Optional) Visually verify the floorplan.

   Use the GUI layout window to visually verify that your pre-synthesis floorplan is laid out according to your expectations. The layout window automatically displays floorplan constraints read in with the `extract_physical_constraints` command or with Tcl floorplanning commands. You need to link all applicable designs and libraries to obtain an accurate floorplan.

   For more information about using the GUI to view physical constraints, see the “Viewing the Floorplan” topic in Design Vision Help.
6. Perform a test-ready and clock-gating compile of the subblock by using the `compile_ultra -scan -gate_clock` command.
7. Specify the design-for-test (DFT) configuration and run the `insert_dft` command as described in Test Synthesis in Topographical Mode.

   You use the `insert_dft` command to insert scan chains, if the subblock is to be included in the top-level scan chain.
8. Perform an incremental compile by using the `compile_ultra -scan -incremental` command.
9. Use the `uniquify_naming_style` variable to specify the unique naming convention to be used by the `uniquify` command, and use the `uniquify -force` command to uniquify the subblock. This prevents naming conflicts during top-level design integration.
10. Run the `change_names -rules verilog -hierarchy` command to apply the Verilog naming rules to all the design objects prior to writing out the design data files.

11. Write the SCANDEF and core test language (CTL) model files by running the following commands:

```
dc_shell-topo> write_scan_def -output design.scandef
dc_shell-topo> write_test_model -format ctl -output design.ctl
```

The SCANDEF file contains information about scan ordering requirements, and the .ctl file contains information about the DFT logic. The information in these files is also stored in the .ddc file, to be used by top-level DFT insertion and reordering. However, the ASCII files can be used for reference.

Note:

You must execute the `write_scan_def` command to annotate the scan ordering information onto the current design.

The steps you perform from this point forward depend on whether you are creating a full .ddc netlist for the subblock in topographical mode or generating a block abstraction or an ILM for the subblock in topographical mode, or you are continuing to implement the subblock in IC Compiler:

- If you are creating a full .ddc netlist for the subblock in topographical mode, save the mapped subblock in .ddc format by using the `write_file -format ddc` command. During top-level synthesis, read in the mapped subblock or add it to the `link_library` variable. See Compiling the Design at the Top Level.
- If you are generating a block abstraction for the subblock in topographical mode, continue with the steps in Generating a Block Abstraction for the Subblock in Topographical Mode.
- If you are generating an ILM model for the subblock in topographical mode, continue with the steps in Generating an ILM Model for the Subblock in Topographical Mode.
- If you are continuing to implement the subblock in IC Compiler, continue with the steps in Generating a Block Abstraction or ILM for the Subblock in IC Compiler.

Generating a Block Abstraction for the Subblock in Topographical Mode

To generate a block abstraction for the subblock in topographical mode, continue with the following steps after completing the steps in Compiling the Subblock:

1. Generate the block abstraction by running the `create_block_abstraction` command.

   The `create_block_abstraction` command identifies the interface logic of the current design and annotates the design in memory with the interface logic.

2. Save the block abstraction by using the `write_file` command. For example,

```
dc_shell-topo> write_file -hierarchy -format ddc -output my_file.mapped.ddc
```

It is important that you use the `write_file` command to save the block abstraction immediately after you create it. The `write_file` command writes the complete design, including the block abstraction information, into a single .ddc file.

To use this block as a block abstraction or as a physical hierarchy .ddc at the top level, continue with the steps in Compiling the Design at the Top Level.

Generating an ILM Model for the Subblock in Topographical Mode

To generate an ILM model for the subblock in topographical mode, continue with the following steps after completing the steps in Compiling the Subblock:

1. Save the full design in .ddc format by using the `write_file -format ddc` command.
2. Run the `write_interface_timing` command to create a timing report for the original netlist.
3. Run the `create_ilm` command to generate the ILM.

   By default, the `create_ilm` command writes out the ILM in .ddc format; therefore, you do not need to specify the `-format` option.
4. Run the `write_interface_timing` command again to create a timing report for the ILM model. This timing report is compared to the timing report for the netlist to help verify consistency between the ILM model and the netlist.
5. Compare the original netlist and ILM interface timing reports by using the `compare_interface_timing` command. This command checks for consistency between the ILM model and the original netlist. For example, the command

   `compare_interface_timing` *`original`*`.wit` *`ilm`*`.wit -output` *`check`*`.cit`

   compares the report for the original netlist in the original.wit file with the report for the ILM model in the ilm.wit file and outputs the comparison report to the check.cit file.
6. Run the `write_file -format ddc` command to save the ILM.

To use the ILM created in Design Compiler topographical mode at the top level, continue with the steps in Compiling the Design at the Top Level.

Generating a Block Abstraction or ILM for the Subblock in IC Compiler

You can continue to implement the subblock in IC Compiler. To create a block abstraction or ILM for top-level integration, perform the following steps. For more information about IC Compiler commands, see the IC Compiler documentation.

1. Write out the block-level design in Design Compiler using the `write_file -format ddc` command.
2. In IC Compiler, set up the design and Milkyway libraries.

3. Read in the .ddc format of the mapped subblock and block SCANDEF information generated in topographical mode.
4. Create the floorplan by using IC Compiler commands or read in floorplan information from a DEF file by using the `read_def` command.
5. Run the `place_opt -optimize_dft` command to perform placement and placement-aware scan reordering.

The steps you perform from this point forward depend on whether you are creating a block abstraction or an ILM:

- To create a block abstraction for the subblock,
  1. Run the `clock_opt` command to perform clock tree synthesis.
  2. Run the `route_opt` command to route the design.
  3. Run the `create_block_abstraction` command to create the block abstraction.
  4. Run the `save_mw_cel` command to create the CEL view.
- To create an ILM for the subblock,
  1. (Optional) Run the `clock_opt` command to perform clock tree synthesis.
  2. (Optional) Run the `route_opt` command to route the design.
  3. Run the `save_mw_cel` command to create the CEL view.
  4. Run the `write_interface_timing` command to create a timing report for the original netlist.
  5. Run the `create_ilm` command to generate the interface logic model for the subblock.
  6. Run the `write_interface_timing` command again to create a timing report for the ILM model. This timing report is compared to the timing report for the netlist to help verify consistency between the ILM model and the netlist.
  7. Compare the original netlist and ILM interface timing reports by using the `compare_interface_timing` command. This command checks for consistency between the ILM model and the original netlist. For example, the command

     `compare_interface_timing` *`original`*`.wit` *`ilm`*`.wit -output` *`check`*`.cit`

     compares the report for the original netlist in the original.wit file with the report for the ILM model in the ilm.wit file and outputs the comparison report to the check.cit file.
  8. Run the `create_macro_fram` command to create the FRAM view.
  9. Run the `save_mw_cel` command to create the CEL view.

During top-level synthesis, read in the ILM (.ddc netlist) of the mapped subblock to integrate it at the top level. See Compiling the Design at the Top Level.

Compiling the Design at the Top Level

To compile the design at the top level,

1. Specify the logic and physical libraries as described in Specifying Libraries.

   If you are using a block abstraction or ILM created by IC Compiler, you need to add the Milkyway design library that contains the ILM view or block abstraction CEL view to the Milkyway reference library list at the top level.

2. If you are using block abstractions created either in Design Compiler topographical mode or IC Compiler, use the `set_top_implementation_options` command to specify which blocks should be integrated with the top-level design as block abstractions.

   - Block abstraction created in topographical mode (.ddc)

     Use the `-block_references` option with the `set_top_implementation_options` command before opening the block. If you do not use this command, the full block netlist will be loaded.

     (Optional) Set the `-optimize_block_interface` option to `true` to enable transparent interface optimization, which optimizes the interface logic within the blocks to achieve faster timing closure.

     For example,

     ```
     dc_shell-topo> set_top_implementation_options \
        -block_references ${BLOCK_ABSTRACTION_DESIGNS} \
        -optimize_block_interface true
     ```

   - Block abstraction created in IC Compiler (CEL)

     Use the `set_top_implementation_options` command to set the top-level design options for linking. If you do not use this command, the tool cannot link to the IC Compiler block abstraction.

     The block abstraction is automatically loaded when you link the top-level design if the Milkyway design library that contains the block abstraction CEL view was added to the Milkyway reference library list at the top level.

3. Read in the top-level design (Verilog, VHDL, netlist, or .ddc) file.

4. Read in the mapped subblock.

   The subblocks can be any of the following:

   - Complete netlist in .ddc format created in Design Compiler topographical mode.

If `set_top_implementation_options` command options are not set for the block, the full netlist .ddc is read in for the block.

- Block abstraction in .ddc format created in Design Compiler.

  You must use the `set_top_implementation_options` command before the .ddc file is loaded in order to load the block as a block abstraction. Only the interface logic is loaded, not the full block.

- Block abstraction created in IC Compiler.

  You must use the `set_top_implementation_options` command before linking the top-level design. The block abstraction is automatically loaded when you link the top level if the Milkyway design library that contains the block abstraction CEL view was added to the Milkyway reference library list at the top level.

- Interface logic model in .ddc format created in Design Compiler topographical mode.

  If you read it in directly, the tool detects it as an ILM during linking. If you add it to the `link_library` variable, the tool automatically loads it when it is linked to the top level.

- Interface logic model created in IC Compiler.

  The ILM is automatically loaded when you link the top level if the Milkyway design library that contains the ILM view was added to the Milkyway reference library list at the top level.

Note:
You can add the .ddc file to the `link_library` variable instead of reading it in directly. If you do this, the tool automatically loads it when it is linked to the top level.

5. In topographical mode, set the current design to the top-level design.
6. Use the `set_physical_hierarchy` command for complete topographical netlists in .ddc format to specify that the subblock should be treated as a physical subblock. When you do this, top-level synthesis preserves both the logic structure and cell placement inside the block. If you do not want to treat a topographical netlist as a physical block, omit the `set_physical_hierarchy` command.

   Use the `set_dont_touch` command for complete topographical or wire load netlists in .ddc format to preserve only the logic structure of the block. When you do this, the block will not be treated as a physical block and the cells inside the block will undergo virtual placement during top-level synthesis.

   The `set_physical_hierarchy` and `set_dont_touch` commands do not need to be used on ILMs and block abstractions. ILMs and block abstractions will automatically have these settings to preserve the logical structure and placement.
7. Use the `link` command to link the subblocks specified in the previous step to the top level.

8. Apply top-level timing and power constraints.
9. (Optional) Provide physical constraints as described in Using Floorplan Physical Constraints.

   You can specify locations for the subblock by using the `set_cell_location` command or by using the `extract_physical_constraints` command to extract physical information from the Design Exchange Format (DEF) file.
10. (Optional) Visually verify the floorplan.

    Use the Design Vision layout window to visually verify that your pre-synthesis floorplan is laid out according to your expectations. The layout view automatically displays floorplan constraints read in with `extract_physical_constraints` or read in with Tcl commands. You need to link all applicable designs and libraries to obtain an accurate floorplan.

    For more information about using the GUI to view physical constraints, see the “Viewing the Floorplan” topic in Design Vision Help.
11. Run the `compile_ultra -scan -gate_clock` command.

    The `-scan` option enables the mapping of sequential cells to appropriate scan flip-flops. The `-gate_clock` option enables clock-gating optimization.
12. Apply the DFT configuration and run the `insert_dft` command to insert scan chains at the top level, followed by the `compile_ultra -scan -incremental` command.
13. Write out the SCANDEF information by using the `write_scan_def` command.

    Top-level scan chain stitching connects scan chains up to the interfaces of test pins of the subblock. In the top-level SCANDEF, you can control whether physical blocks are described using the “BITS” construct or as complete scan chains.

    If you are using Design Compiler ILMs, block abstractions, or physical hierarchy blocks, you use the `write_scan_def` command with the `-expand_elements` option to write out the SCANDEF information. You do not maintain the test hierarchy partition during top-level physical implementation in IC Compiler. The SCANDEF must directly reference the scan leaf objects of the physical block instead of the BITS construct to run the IC Compiler top-level flat flow. For example,

    ```
    write_scan_def -expand_elements block_instance
    ```

    If you are using IC Compiler ILMs or block abstractions, you maintain the test hierarchy partition during top-level physical implementation. Therefore, the top-level SCANDEF is written out with the block containing the BITS construct; you use the `write_scan_def` command without the `-expand_elements` option to write out the SCANDEF information.

    If your flow contains both Design Compiler and IC Compiler models, you should only specify the `-expand_elements` option for Design Compiler ILMs, block abstractions, or

physical hierarchies. Omit the blocks modeled by IC Compiler from the block list when you specify the `-expand_elements` option.

14. Write out the top-level netlist by using the `write_file -hierarchy -format ddc` command.

    When you do this, all physical blocks are removed automatically. If you enabled transparent interface optimization, you need to save the updated blocks. Use the `write_file` command to save each optimized block abstraction to a separate .ddc file.

    You can use the following script to save each optimized block abstraction to a .ddc file:

```
foreach design ${BLOCK_ABSTRACTION_DESIGNS}{
  write_file -hierarchy -format ddc \
             -output ${design}.mapped_tio.ddc \
             ${design}
}
```

    The `write_file` command merges the changes with the original .ddc design and writes the complete block as a new .ddc file.

Supported Commands, Command Options, and Variables

In topographical mode, if an unsupported command or variable is encountered in a script, an error message is issued; however, the script *continues*.

For example,

```
ERROR: Command set_wire_load_mode is not supported in

DC Topographical mode. (OPT-1406)
```

For unsupported command options (except wire load model options), an error message is issued, and the script stops executing. Wire load model options are ignored, and the script continues.

Also, variables that are read-only in topographical mode are indicated read-only variables in error messages.

Look for additional supported commands, command options, and variables in subsequent releases. You should check your scripts and update them as needed. See the appropriate man pages to determine the current status in topographical mode.

Using the Design Compiler Graphical Tool

The Design Compiler Graphical tool extends topographical technology and enables you to optimize multicorner-multimode designs, reduce routing congestion, and improve both area correlation with IC Compiler and runtime in IC Compiler by using Synopsys physical guidance. In addition, Design Compiler Graphical enables you to create and modify floorplans using floorplan exploration.

This section describes Design Compiler Graphical features in following subsections:

- Improving Area Correlation, Runtime, and Routability, and Reducing Congestion With Physical Guidance
- Creating and Modifying Floorplans Using Floorplan Exploration

For multicorner-multimode support details, see Optimizing Multicorner-Multimode Designs in Design Compiler Graphical.

Improving Area Correlation, Runtime, and Routability, and Reducing Congestion With Physical Guidance

Synopsys physical guidance technology enables Design Compiler Graphical to perform enhanced placement that is consistent with the IC Compiler `place_opt` command functionality and enhanced post-placement delay optimization in order to provide a better optimized starting point for physical implementation. The placement information is passed to the IC Compiler `place_opt` command and is used as seed placement to guide physical implementation of the design. As a result, placement is aligned between Design Compiler and IC Compiler, improving runtime, quality of results (QoR), correlation, and routability. In addition, physical guidance enables congestion optimization, which reduces routing-related congestion.

Using the physical guidance flow, IC Compiler no longer needs to create a coarse placement by running commands such as `create_placement`, `remove_buffer_tree`, or `psynopt`.

The following sections describe the physical guidance functionality:

- Physical Guidance Overview
- Reducing Routing Congestion
- Specifying Design Constraints and Power Settings
- Using Layer Optimization to Increase the Accuracy of Net Delay Estimation
- Using Physical Guidance in Design Compiler (compile_ultra -spg)
- Using the Incremental Flow (compile_ultra -incremental -spg)

Physical Guidance Overview

The physical guidance flow is jointly supported in Design Compiler and IC Compiler to improve runtime, correlation, and routability and to reduce routing-related congestion. Physical guidance significantly speeds up IC Compiler runtime because IC Compiler uses the physical guidance information from Design Compiler Graphical as its starting point and therefore goes through fewer placement steps.

To ensure good QoR and correlation between Design Compiler and IC Compiler, you need to use consistent design and tool setup between Design Compiler Graphical and IC Compiler. You should use the same sets of logic and physical libraries to drive both tools and use consistent library settings. Also, use consistent design goal specifications, such as timing constraints and design rule settings to drive logical and physical implementation.

You enable the physical guidance flow by using the `-spg` option with the `compile_ultra` command in Design Compiler Graphical and by using the `-spg` option with the `place_opt` command in IC Compiler. The `-spg` option works seamlessly with the existing `compile_ultra` options in Design Compiler and with the `place_opt` options in IC Compiler.

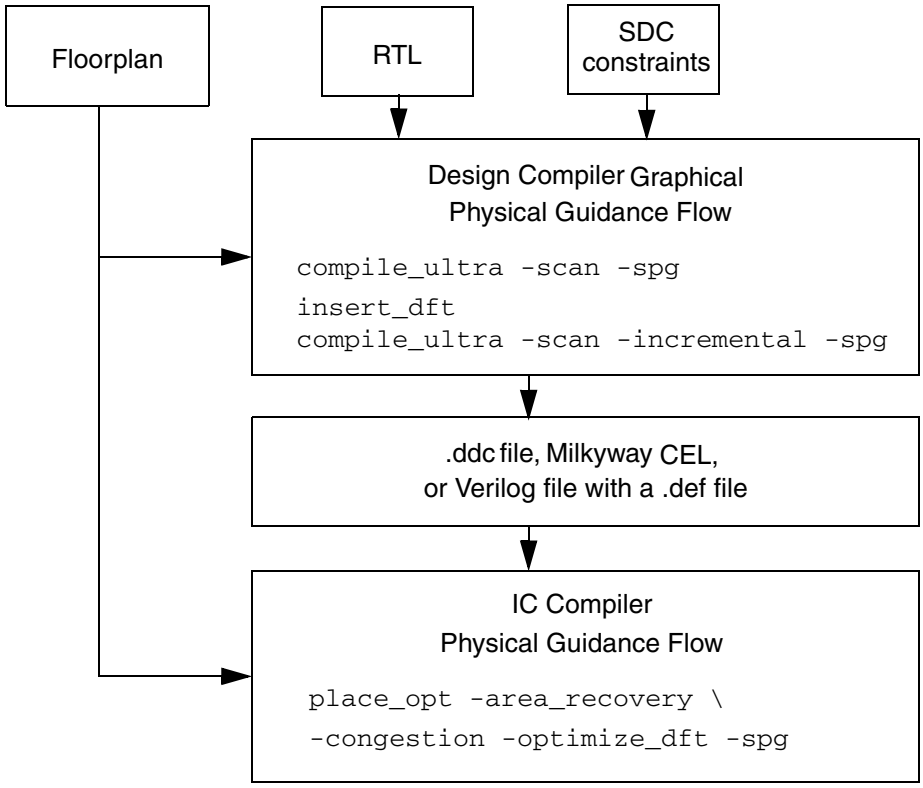
Physical guidance performs the following functions:

- In Design Compiler Graphical, the `-spg` option provides the following:
  - Further refined placement that is consistent with the `place_opt` command in IC Compiler
  - Enhanced post-placement delay optimization to provide a better optimized starting netlist for physical implementation
  - Optimizes the design for congestion to improve routability

  You can save the placement information derived by Design Compiler Graphical in binary format, in a .ddc file or a Milkyway CEL view, or you can save it in a .def file.
- In IC Compiler, the `-spg` option enables the `place_opt` command to use the Design Compiler Graphical placement as its seed placement. The `place_opt` command is enhanced to use topographical mode placement as the starting point in physical guidance mode.

Figure 10-24 shows an example of a typical physical guidance flow.

Figure 10-24 Physical Guidance Flow Example



The physical guidance flow supports all Design Compiler topographical mode compile options, such as `-scan`, `-clock_gate`, `-congestion`, and `-incremental`. In addition, physical guidance supports the multicorner-multimode based flow, the multivoltage-UPF flow, the power flow, the DFT flow, and the hierarchical flow.

Use the following guidelines to maintain good QoR and correlation between Design Compiler and IC Compiler when you use the physical guidance flow:

- Use consistent design and tool setup between Design Compiler topographical mode and IC Compiler. That is, you should use the same sets of logic and physical libraries to drive both tools and use consistent library settings, including the dont_use list. Also, use consistent design goal specifications, such as timing constraints and design rule settings to drive logical and physical implementation.

  You should load the same sets of the following files in both tools:

  - Logic library (.db)
  - Milkyway reference library
  - Technology file (.tf)

- Mapping file (maps the technology file layer names to TLUPlus layer names)
- Min and max TLUPlus files for net parasitics

Make sure that each standard cell and macro in the logic libraries has a corresponding abstract physical view in the Milkyway reference library. Otherwise, the cells are treated as dont_use cells for mapping and optimization.

In addition, you should use consistent library-specific attribute settings with IC Compiler. For instance, your library cell dont_use list, dont_touch settings, and design rule settings (maximum allowable fanout load and transition time) should be aligned with the settings used in IC Compiler.

- Apply a complete set of physical constraints by using a floorplan DEF file or Tcl commands before the first `compile_ultra` command step in Design Compiler. Also, reapply the same sets of physical constraints in IC Compiler to preserve the placement consistency for physical guidance.

  The same DEF file must be used by both tools. The DEF file contains floorplan information that must be consistent between tools. The tools issue various errors and warnings if your constraints, floorplan information, and libraries are not consistent.

- Since the physical guidance information is restored when you run the IC Compiler `place_opt` command, it is recommended that you run the `place_opt -spg` command as the first placement-related command in IC Compiler so that Design Compiler placement is restored properly. Using other placement-related commands (such as `create_placement`, `remove_buffer_tree`, and so on) might degrade correlation and can lead to the loss of some physical guidance information.

Reducing Routing Congestion

The following sections describe how physical guidance reduces routing congestion:

- Routing Congestion Overview
- Viewing Congestion With the Design Vision Layout Window
- Predicting, Analyzing, and Minimizing Routing Congestion
- Specifying Congestion Optimization Options
- Specifying Block-Level Congestion Optimization
- Reporting Congestion

Routing Congestion Overview

Designs are considered congested when the wires needed to connect design components become greater than the space available to put the wires. If Design Compiler passes a congested design to IC Compiler, it might be difficult and time consuming for IC Compiler to

reduce the congestion. Design Compiler can reduce congestion easier than IC Compiler because Design Compiler can optimize RTL structures. Traditionally, to minimize congestion related to the netlist topology, you would use scripting solutions to restrict technology mapping tools, such as IC Compiler, from choosing library cells that you perceive as poor for congestion. But these solutions do not significantly improve congestion; instead, they often create poor timing or area results.

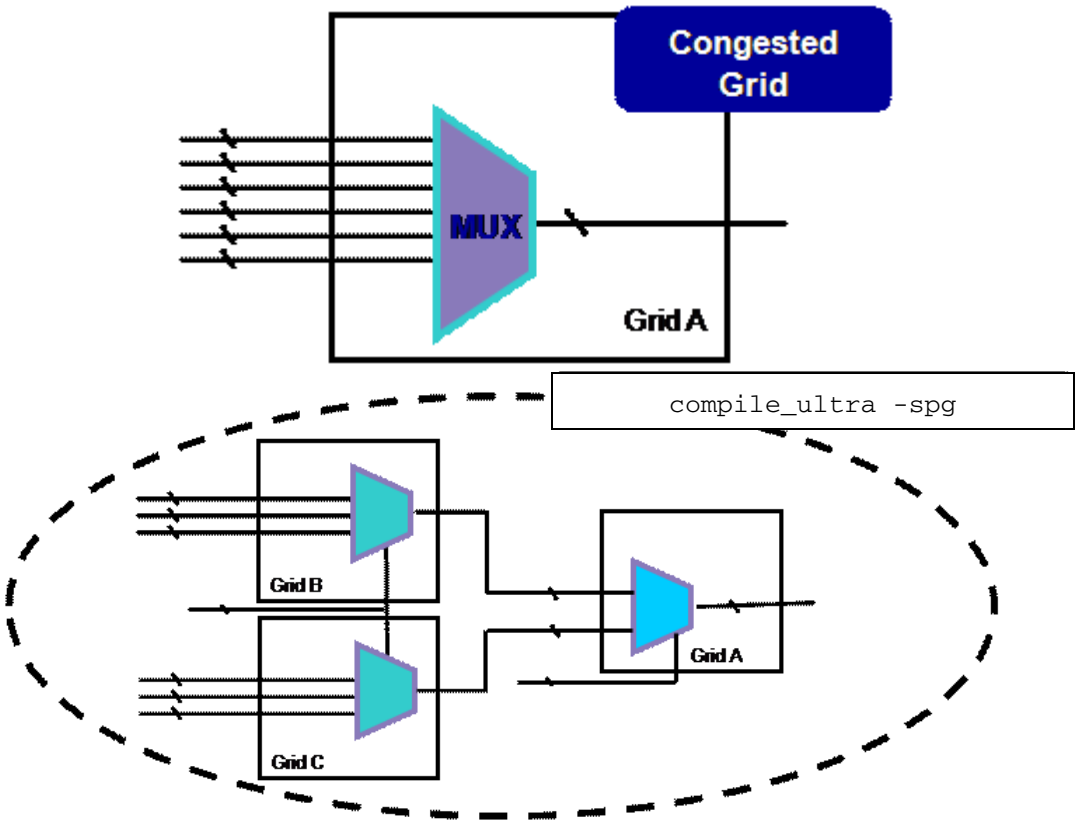
Design Compiler Graphical reduces routing congestion when you use the `-spg` option with the `compile_ultra` command. The tool produces logic structures that are better suited for a placement that minimizes congestion and improves routability. In addition, you can use the `set_congestion_options` command to specify options, such as the amount of resources available for a given layer. By default, Design Compiler optimizes the complete design for congestion. However, you can also use the `set_congestion_optimization` command to disable or enable congestion optimization for a specific design or instance. For more information about specifying congestion optimization for a specific design or instance, see Specifying Block-Level Congestion Optimization.

To report details about congestion, use the `report_congestion` and the `report_congestion_options` commands. For graphical representation of the congested areas, use the Design Vision layout window. From this window you can view and analyze congestion.

Congestion optimization is primarily beneficial for designs in which RTL logic structures result in congestion. This standard cell congestion is often caused by the topology of the netlist. Minimizing the congestion for such designs often requires significant changes in the netlist topology, which cannot be done during place and route. Such large changes are best done early in the flow during synthesis.

Figure 10-25 shows an example of how the tool optimizes an RTL structure to minimize congestion. In this example, the tool optimizes the single large multiplexer into three smaller multiplexers which reduces routing congestion by restructuring the RTL.

Figure 10-25 Optimizing RTL Structures to Minimize Congestion



Viewing Congestion With the Design Vision Layout Window

You can identify areas of high congestion in your design by viewing the congestion map in the Design Vision layout window. By visually examining congested areas in your design, you can determine whether the design is routable and identify the causes of the congestion if the design is not routable. You can display or hide the congestion map at any time when a layout view is open in the layout window.

In some cases, it may be useful to open the layout window and display the congestion map or visual mode images with your script. You can also save an image of the layout in a file.

For information about displaying the congestion map and viewing cells in congested areas, see the *Design Vision User Guide* and Design Vision Help.

Predicting, Analyzing, and Minimizing Routing Congestion

Design Compiler predicts wire-routing congestion “hot spots” during RTL synthesis and uses its interactive capability to visualize the design’s congestion. When the tool determines that the design has congestion problems, it minimizes congestion through congestion optimization, resulting in a better starting point for layout.

Wire-routing congestion in a design occurs when the number of wires traversing a particular region is greater than the capacity of the region. If the ratio of usage-to-capacity is greater than 1, the region is considered to have congestion problems. Congestion can be associated with standard cells or the floorplan. Typically, standard cell congestion is caused by the netlist topology, for example, large multiplexer trees, large sums of products (ROMs), test decompression logic, or test compression logic. Floorplan congestion can be caused by macro placement or port location.

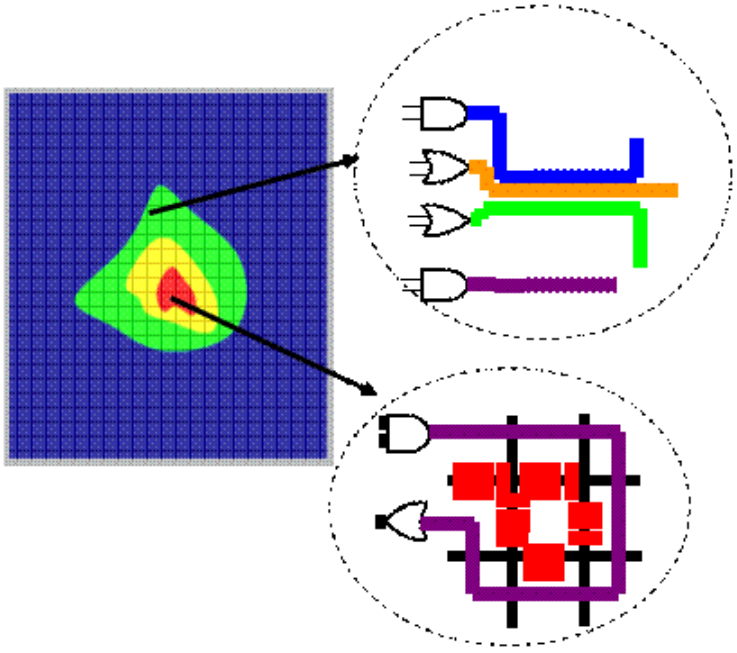
Design Compiler uses the Synopsys virtual global routing technology to predict wire-routing congestion. In this technology, congestion is estimated by dividing the design into a virtual grid of global routing cells followed by a global route to count the number of wires crossing each grid edge. The capacity and size of each global routing cell is calculated by using the technology physical library information. A cell is considered congested if the number of wires passing through it is greater than the number of available tracks.

Design Compiler predicts the congestion number by subtracting the maximum wires allowed for a particular edge direction from the total number of wires crossing per edge. Any number over zero is a congestion violation. This calculation is represented by the following equation:

```
Congestion = (number of wires) - (maximum wires allowed)
```

Figure 10-26 shows how Design Compiler Graphical predicts congestion. The red areas indicate congestion.

Figure 10-26 Predicting Congestion Using Virtual Grid Route



The tool considers floorplan-related physical constraints when it estimates routing congestion. In particular, the wiring keepout physical constraint ensures that the tool is aware of areas that must be avoided during routing. This physical constraint helps to achieve consistent congestion correlation with layout.

Specifying Congestion Optimization Options

To specify options for congestion optimization, use the `set_congestion_options` command. This command helps achieve consistent congestion correlation between Design Compiler in topographical mode and IC Compiler by ensuring that both tools see the same congestion setup.

To remove congestion options, use the `remove_congestion_options` command.

To report congestion options, use the `report_congestion_options` command.

Use the `set_congestion_options` command options as described in Table 10-16.

Table 10-16 Using the set_congestion_options Command

To do this	Use this option
Minimize congestion by moving cells from congested regions to uncongested regions. Note: Use the `-max_util` option to specify the maximum utilization the tool allows as cells migrate into uncongested areas. For example, setting a value of 0.9 allows the tool to place cells in an area with up to 90 percent utilization.	`-max_util`
Specify the layer whose availability you want to reduce.	`-layer`
Specify how much of the routing resource for the given layer is available. A value of 0.30 for availability means the tool considers 70% as being used.	`-availability`
Specify the lower left and upper right coordinates for which the congestion options apply.	`-coordinate`

Specifying Block-Level Congestion Optimization

By default, Design Compiler Graphical optimizes the complete design for congestion. However, you can use the `set_congestion_optimization` command to disable or enable congestion optimization for a specific design or instance. The design must be loaded in memory. When the command is specified on an instance, it is hierarchical and applies to all subinstances.

To specify block-level congestion optimization, use the `set_congestion_optimization` command:

```
set_congestion_optimization object_list true | false
```

where *`object_list`* represents cells, designs, or instances and can be a list or collection. The `set_congestion_optimization` command is set to `true` by default.

In the following example, congestion optimization is disabled for cell sub_a:

```
dc_shell-topo> set_congestion_optimization [get_cells top/sub_a] false
```

To specify congestion optimization on only one instance, first disable the optimization on all designs by setting the `set_congestion_optimization` command to `false` on the top-level design. Then, enable optimization on the instance, as shown:

```
dc_shell-topo> set_congestion_optimization [current_design] false
dc_shell-topo> set_congestion_optimization [get_cells top_level/i_sub] true
```

After you have enabled or disabled the optimization as needed, you must run the `compile_ultra -spg` command. The `-spg` option is required for the `set_congestion_optimization` settings to be applied.

Reporting Congestion

The text based `report_congestion` command provides a quick estimate of the congestion status of the design. In cases where the design is reported as significantly congested, you can further analyze congestion by generating a congestion map. By viewing the congestion map, you can identify areas of high congestion and determine whether the design can be routed.

For this detailed analysis, you use the Design Vision layout window to analyze the congestion map. You load the .ddc netlist synthesized in topographical mode to analyze the congestion map and identify the likely cause of congestion in the design. For more information, see the *Design Vision User Guide* or Design Vision Help.

The `report_congestion` command generates a report that shows the design's congestion status. You can use this report to assess the severity of congestion in the design. Example 10-46 shows a sample output generated by the `report_congestion` command.

Example 10-46 The report_congestion Command Output

```
****************************************
Report : congestion
Design : my_design
...
****************************************
Both Dirs: Overflow = 4651 Max = 5 (10 GRCs) GRCs = 3401 (0.59%)
H routing: Overflow = 2386 Max = 3 (50 GRCs) GRCs = 1788  (0.31%)
V routing: Overflow = 2265 Max = 3 (18 GRCs) GRCs = 1961  (0.34%)
```

You interpret this report as follows:

- *Overflow* is the total number of wires in the design global routing cells that do not have a corresponding track available. The following equation shows how overflow is measured:

$$\text{Overflow} = \sum_{i=1}^{\text{Max}} \text{violation(i)} \times \text{GRCs with violation(i)}$$

- *Max* corresponds to the highest number of over-utilized wires in a single global routing cell. It is the worst-case violation and the number of global routing cells that have the violation. The report shows that the worst-case violation is five and it occurs for ten global routing cells in the design.
- *GRC* is the total number of over-congested global routing cells in the design. The report indicates that there are 3401 violating GRCs in the design, which account for 0.59% of the design.

Specifying Design Constraints and Power Settings

Before performing design implementation, you must first specify the design QoR objectives for synthesis. You need to apply the design constraints and power settings that will be used to perform mapping and placement-driven optimization in Design Compiler topographical mode.

In the physical guidance synthesis flow, it is especially important that you fully specify your design goals before you run the first `compile_ultra` command step in order to achieve good QoR results that correlate to IC Compiler.

Keeping in mind that Design Compiler physical guidance information is used as seed placement for the `place_opt` command in IC Compiler, the following design objectives should be consistent so the placement is aligned and the timing context is the same between both tools:

- Design-Specific Settings
- Physical Constraints
- Power Optimization Settings

Design-Specific Settings

Align your design settings in Design Compiler with the settings used in IC Compiler. For example, if your back-end flow disables the use of certain metal layers for routing or uses any commands that affect net parasitic (resistance and capacitance) estimation, you need to specify the same design settings in Design Compiler.

The following list shows the type of design settings that should be consistent between Design Compiler and IC Compiler:

- Use the `set_ignored_layers` command to disable the routing resources on certain metal layers. Ignoring metal layers affects Design Compiler placement-driven net capacitance and resistance computation. The ignored metal layer is not considered part of the routing resource; its effect on the resistance and capacitance is not considered during optimization and congestion estimation in topographical mode.
- Use the `set_delay_estimation_options` command to specify scaling factors for derived resistance (R) and capacitance (C) from the library or override the per-unit R and C values of any metal layers specified in the physical library. A scaling factor within 5% to 10% is usually sufficient.
- Use the `set_ahfs_options` command to fine-tune automatic high-fanout synthesis options. By default, Design Compiler performs placement-based automatic high-fanout synthesis for high-fanout nets (with fanout above 100) such as reset and scan-enable, with the exception of dont_touch nets, ideal nets, and DRC-disabled clock and constant nets. Placement of fanout influences buffering. As a result, you need to allow Design Compiler to perform the default automatic high-fanout synthesis and only configure automatic high-fanout synthesis options to the same options as IC Compiler for consistent buffering behavior. IC Compiler automatically performs incremental automatic high-fanout synthesis as needed when you run the `place_opt` command.
- Use the `dont_touch` and `size_only` attributes to control cell mapping and optimization. The usage of these attributes limits optimization flexibility and affects QoR and correlation. Thus, you should use the same sets of `dont_touch` and `size_only` attributes in both tools.
- Use the same set of `set_ideal_network` constraints that are defined during `place_opt` in IC Compiler to ensure correlation.

You should only use these commands when they are being used in your back-end flows because they affect placement, parasitic estimation, mapping, and placement-based optimization in topographical mode the same way as in IC Compiler. Using consistent commands in both tools results in improved correlation for physical implementation.

Physical Constraints

In the physical guidance flow, you should drive placement-based optimization in synthesis with the same set of physical constraints used in IC Compiler to preserve placement consistency and improve correlation with IC Compiler.

Design Compiler supports a number of physical constraints. The physical constraints are extracted from your design floorplan DEF file with the `extract_physical_constraints` command, or they are written out from the Tcl output when you use the `write_floorplan` command in IC Compiler, and they are read into Design Compiler topographical mode when

you use the `read_floorplan` command. For more information about the physical constraints supported in Design Compiler topographical mode, see Using Floorplan Physical Constraints.

In addition to supporting physical constraints, Design Compiler also supports constraints for specifying the following:

- Net shapes or preroutes
- Relative placement
- Voltage area
- Tracks

You should consider the following physical constraint specification and usability guidelines for physical guidance flows and comply with the minimum set of floorplan requirements before you run the `compile_ultra` command:

- Use the same floorplan and set of physical constraints in Design Compiler as in IC Compiler. It is recommended that you drive synthesis with the same floorplan used for physical implementation in IC Compiler. However, as a minimum, you must specify the die area before proceeding with mapping and placement-based optimization using the `compile_ultra -spg` command.

  In addition, it is recommended that you specify the macro and ILM locations, the site arrays, and the port locations for all primary I/Os.

  To ensure that you have provided the required floorplan information, explicit physical guidance usability checks are automatically performed at the onset of the `compile_ultra -spg` command run in topographical mode. You can also run the `compile_ultra -check_only -spg` command to get information about any missing physical constraints.

- Use the same floorplan information between the initial `compile_ultra` step and subsequent incremental compilation steps with the exception of creating new test ports and specifying locations for the new test ports. If you do not provide the floorplan information before the initial `compile_ultra` step, you might see warning messages.

  If you need to modify the floorplan, you can use the Design Compiler Graphical floorplan exploration capability. For more information about using Design Compiler Graphical floorplan exploration, see Creating and Modifying Floorplans Using Floorplan Exploration.

Power Optimization Settings

Design Compiler supports power prediction and optimization in the physical guidance flow, which provides accurate power estimation and correlates post-synthesis power numbers with those after place and route.

Use the following commands to enable leakage power and dynamic power optimization, respectively:

- `set_leakage_optimization`
- `set_dynamic_optimization`

When you enable leakage power or dynamic power optimization, you must use multiple threshold-voltage libraries. For the best power results, set the leakage power before running the `compile_ultra` command.

To enable clock-gating optimization, use the `-gate_clock` option with the `-spg` option of the `compile_ultra` command. The tool automatically inserts, modifies, or deletes a clock-gating cell unless you have marked the cell or its parent hierarchical cell with the `dont_touch` attribute.

If you enable power correlation by using the `set_power_prediction` command, the tool performs clock tree estimation during the last phase of the `compile_ultra` or `compile_ultra -incremental` command run. Any subsequent incremental compilations cause clock tree estimation to run again.

To provide a list of library cells to be used for clock tree estimation, use the `-cts_references` option with the `set_power_prediction` command. Using the same clock buffers that are used in IC Compiler clock tree synthesis helps ensure that the predicted power is further correlated with IC Compiler.

If you enable power prediction, the `report_power` command reports the correlated power if the design is mapped to technology-specific cells. If power prediction is disabled, the `report_power` command reports only the total power, static power, and dynamic power used by the design without accounting for the estimated clock-tree power.

For information about configuring power optimization and prediction, see Power Optimization in Topographical Mode and the *Power Compiler User Guide*.

Using Layer Optimization to Increase the Accuracy of Net Delay Estimation

When Design Compiler computes the resistance and capacitance of a net, it uses an average based on the resistance and capacitance of all available layers from the logic library. That average works well when all layers have the same or close enough unit resistance and capacitance value. However, with submicron technologies, the layer characteristics can vary greatly. Such variation can cause correlation or timing issues for nets that are known to be routed with only some specified layers. In this case, averaging is not an appropriate solution.

Layer optimization in Design Compiler Graphical allows you to define specific layer assignment constraints and apply them to nets. This increases the accuracy of net delay estimation during optimization. To enable layer optimization, use the `compile_ultra`

command with the `-layer_optimization` option. Layer optimization is supported only in Design Compiler Graphical. Therefore, you must use the `-spg` option with the `compile_ultra -layer_optimization` command.

This section provides an overview of the layer optimization commands. For more information about these commands, related layer optimization commands, and the layer optimization flow, see SolvNet article 037946, "How to Implement Net Layer Optimization With Design Compiler Graphical."

You can perform layer optimization with the following methods:

- A net pattern
- User constraints
- Automatic recognition

In the net pattern methodology, you use the `create_net_search_pattern` command to define a pattern to identify nets, and you use the `set_net_search_pattern_delay_estimation_options` command to define which layers to use for a specific pattern.

In the user constraints methodology, you use the `set_net_routing_layer_constraints` command to specify which nets will have specific layer assignment constraints applied to them.

In the automatic recognition methodology, you allow Design Compiler Graphical to automatically identify nets and assign them to specific layers to reduce the resistance and capacitance on those nets, instead of using the averaging technique.

Regardless of the methodology you use, layer optimization can have an effect on runtime.

Using Physical Guidance in Design Compiler (compile_ultra -spg)

To enable physical guidance in Design Compiler, use the `compile_ultra` command with the `-spg` option. You must specify the die area at a minimum to ensure floorplan consistency between Design Compiler and IC Compiler. It is recommended that you also specify the macro and ILM locations, the site arrays, and the port locations for all primary I/Os.

Design Compiler performs explicit physical guidance usability checks the first time you run the `compile_ultra -spg` command in topographical mode to ensure that you have provided the required floorplan information, and error messages and warning messages are issued for any missing floorplan information. For example, Design Compiler issues a warning message if a macro location is missing during a physical guidance flow; however, the `compile_ultra -spg` command assigns the macro location automatically.

If a port, macro, or ILM location is missing, you can proceed with synthesis. However, you must provide the location information in IC Compiler. Otherwise, correlation could be affected. Alternatively, you can use Design Compiler Graphical floorplan exploration to place the ports, macros, or ILMs. After you specify the physical constraints, update the floorplan in floorplan exploration. The `compile_ultra -spg` command can move ports incrementally to enhance timing if the port locations have not been defined.

The IC Compiler `restore_spg_placement` command restores the port locations created by the Design Compiler `compile_ultra -spg` command for ports that were not constrained by the `set_port_location` command or defined in the DEF file. The `restore_spg_placement` command also restores the standard cell placement created by the `compile_ultra -spg` command. For more information, see the IC Compiler documentation.

In the physical guidance flow, RTL congestion optimization is enabled as well as congestion-aware placement to consider cell density and provide a more accurate net model for optimization in topographical mode. This helps to ensure that timing is consistent between the endpoint in Design Compiler and the startpoint in IC Compiler. In addition, placement and placement-based optimization in physical guidance enhances the delay optimization effort in Design Compiler so it is consistent with IC Compiler behavior.

At the end of the `compile_ultra -spg` command run, Design Compiler creates signatures of the physical constraints that were used. The signatures are saved in binary format, in either the .ddc file or the Milkyway CEL, to be later matched in IC Compiler against the physical constraints used for the `place_opt -spg` command run. This allows you to ensure that the physical constraints used in Design Compiler match the physical constraints used in IC Compiler.

At this point in the flow, the mapped top-level design is marked with a read-only `dct_spg_flow_done` attribute to signify that the design was implemented with physical guidance. You can query the attribute to help determine if your design .ddc file or Milkyway CEL was generated using physical guidance.

The derived placement for standard cells and the design floorplan information used to drive synthesis are retained on the in-memory implemented design. You can save the physical guidance information in a .ddc file by using the `write_file` command or to a Milkyway CEL view by using the `write_milkyway` command. In the ASCII flow, you can save the physical guidance information in a .def file by using floorplan exploration.

Using the Incremental Flow (compile_ultra -incremental -spg)

Design Compiler Graphical also provides an incremental physical guidance flow. The information in the full `compile_ultra -spg` command section, Using Physical Guidance in Design Compiler (compile_ultra -spg), also applies to the `compile_ultra -incremental -spg` command. In addition, the following applies to the incremental flow:

- When you run the `compile_ultra -incremental -spg` command on an input .ddc file, Design Compiler Graphical performs an explicit check for the presence of the read-only `dct_spg_flow_done` attribute to ensure that the design was initially implemented in physical guidance mode.

  The tool issues an error message when you run subsequent `compile_ultra -incremental -spg` commands on a .ddc file that was not generated using physical guidance.

- When Design Compiler Graphical confirms that the .ddc file that is being read was generated using physical guidance, any subsequent placement-based optimization using the `compile_ultra -incremental` command uses physical guidance regardless of whether you include the `-spg` option. That is, the `compile_ultra -incremental` and `compile_ultra -incremental -spg` commands have the same effect. At this point, the tool issues a warning message informing you that you are in physical guidance mode and that the design was compiled with physical guidance optimizations.

- When you run the `compile_ultra -incremental -spg` command, Design Compiler Graphical automatically enables scan chain reordering during incremental compilation to further reduce scan chain wire length. To disable it, set the `test_enable_scan_reordering_in_compile_incremental` variable to `false`.

  Scan chain reordering is enabled in binary and ASCII flows. For more information about performing scan chain reordering using an ASCII flow, see the “Incremental ASCII Flow With a Third-Party DFT Flow Example” chapter in the *Design Compiler Optimization Reference Manual*.

The physical guidance flow accepts any of the input source formats that are supported by Design Compiler. The following describes the expected behavior for the `compile_ultra -incremental -spg` command on various input source types:

- .ddc file generated with the `-spg` option

  The `compile_ultra -incremental -spg` command proceeds with placement-based optimization in physical guidance mode.

- .ddc file generated without the `-spg` option

  The `compile_ultra -incremental -spg` command issues a DCT-054 error message. You must first use `compile_ultra -spg` to map the design.

- ASCII mapped netlist

  If you provide standard cell placement information by using the `-standard_cell spg` option with the `extract_physical_constraints` command, the `compile_ultra -incremental -spg` command proceeds with placement-based optimization in physical guidance mode. The tool does not support a netlist that contains both mapped and unmapped cells in this flow.

  If you provide standard cell placement information by using the `-standard_cell topo` option with the `extract_physical_constraints` command, the `compile_ultra -incremental -spg` command issues a DCT-054 error message because the design was not compiled with Design Compiler topographical physical guidance optimizations.

  If you do not provide standard cell placement information, the `compile_ultra -incremental -spg` command runs limited mapping optimizations and then performs placement-based optimization in physical guidance mode.

If you use other mapping commands, such as `optimize_registers` or netlist editing commands that perform various degrees of design modification, it is recommended that you run an explicit incremental compilation flow afterward with the `-spg` option. However, it is not required. The `place_opt -spg` command in IC Compiler can handle cells that do not have physical guidance defined.

It is recommended, but not required, that you run an incremental compilation flow after running the following commands:

- `optimize_registers`
- `insert_dft`
- `insert_buffer`
- `remove_buffer`
- `balance_buffer`
- `change_link`

The `compile_ultra -incremental -spg` command can move ports incrementally to enhance timing if the port locations have not been defined.

Exporting the Design

You can save standard cell placement in Design Compiler in binary or ASCII format, as described in the following sections.

Saving in Binary Format

You can save standard cell placement in .ddc format by using the `write_file -format ddc` command, or save it in Milkyway CEL format by using the `write_milkyway` command.

In regular topographical mode, the standard cell placement can only be used by Design Compiler for further optimization. IC Compiler does not use it. However, in the physical guidance flow, the standard cell placement information is propagated to IC Compiler through the .ddc file or the Milkyway CEL.

The physical constraints and floorplan information used for synthesis are not passed from Design Compiler to IC Compiler. However, physical constraint signatures are created from the physical constraints at the end of the `compile_ultra -spg` and the `compile_ultra -incremental -spg` command steps in the physical guidance flow. These signatures are saved in the .ddc file or Milkyway CEL view. IC Compiler reads the signatures when the physical guidance information is restored and matches them to the physical guidance information in IC Compiler to ensure consistency between the floorplan used in synthesis and the floorplan used for physical implementation.

All binary netlists that are generated in the physical guidance synthesis flow, whether a .ddc file or a Milkyway CEL, also contain a read-only `dct_spg_flow_done` attribute to signify that the design was implemented with the physical guidance flow. You can query the attribute to determine if your design .ddc file or Milkyway CEL was generated using the physical guidance flow.

Saving in ASCII Format

When you save the design in ASCII format, either in a Verilog or a VHDL netlist, the file that is created does not contain standard cell placement information or physical constraint signatures. You can save the standard cell placement information in a DEF file by using the `write_def` command in floorplan exploration. You can then read the DEF file, along with the ASCII netlist, back into Design Compiler to be optimized with the `compile_ultra -incremental -spg` command, or you can read it into IC Compiler before the `place_opt -spg` step.

Using Physical Guidance in IC Compiler (place_opt -spg)

To enable physical guidance in IC Compiler, use the `-spg` option with the `place_opt` command. When you run the `place_opt -spg` command, the stored physical guidance information from Design Compiler is used as the initial seed placement.

For more information about using physical guidance in IC Compiler, see SolvNet article 031198, “Design Compiler and IC Compiler Physical Guidance Technology Application Note.”

Using the Design Compiler Graphical and IC Compiler Hierarchical Flow

You can use physical guidance with a hierarchical flow in Design Compiler Graphical and IC Compiler. If the hierarchical blocks in Design Compiler correspond to physical blocks, use IC Compiler block abstractions or IC Compiler ILMs to model the physical blocks at the top level in both tools. This ensures optimal correlation and alignment between the tools.

If IC Compiler hierarchical models are not available, Design Compiler physical blocks (full netlist .ddc files, Design Compiler block abstractions, or Design Compiler ILMs) can be used at the top level in Design Compiler Graphical. However, IC Compiler does not support Design Compiler physical blocks, so when you run the top-level design in IC Compiler, you must generate IC Compiler ILMs or IC Compiler block abstractions.

The design that is written out from Design Compiler Graphical after completing top-level synthesis will contain physical guidance data only for the top-level logic. Physical guidance information will not be included for the physical blocks. Each physical block should be passed to IC Compiler separately to complete the physical guidance flow for each block.

If the hierarchical blocks in Design Compiler do not correspond to physical blocks, do not use block abstractions or ILMs for those blocks when performing top-level synthesis. You should read in the block-level .ddc file at the top level without using the `set_physical_hierarchy` command on those blocks. The logic in blocks that are not physical blocks will be placed during top-level synthesis and will be included in the physical guidance information at the top level.

In the hierarchical Design Compiler Graphical and IC Compiler flow using physical guidance, specify the `compile_ultra -spg` command in Design Compiler Graphical, and specify the `place_opt -spg` command in IC Compiler. For more information about running a hierarchical flow, see Performing a Bottom-up (Hierarchical) Compile.

Incremental ASCII Flow With a Third-Party DFT Flow Example

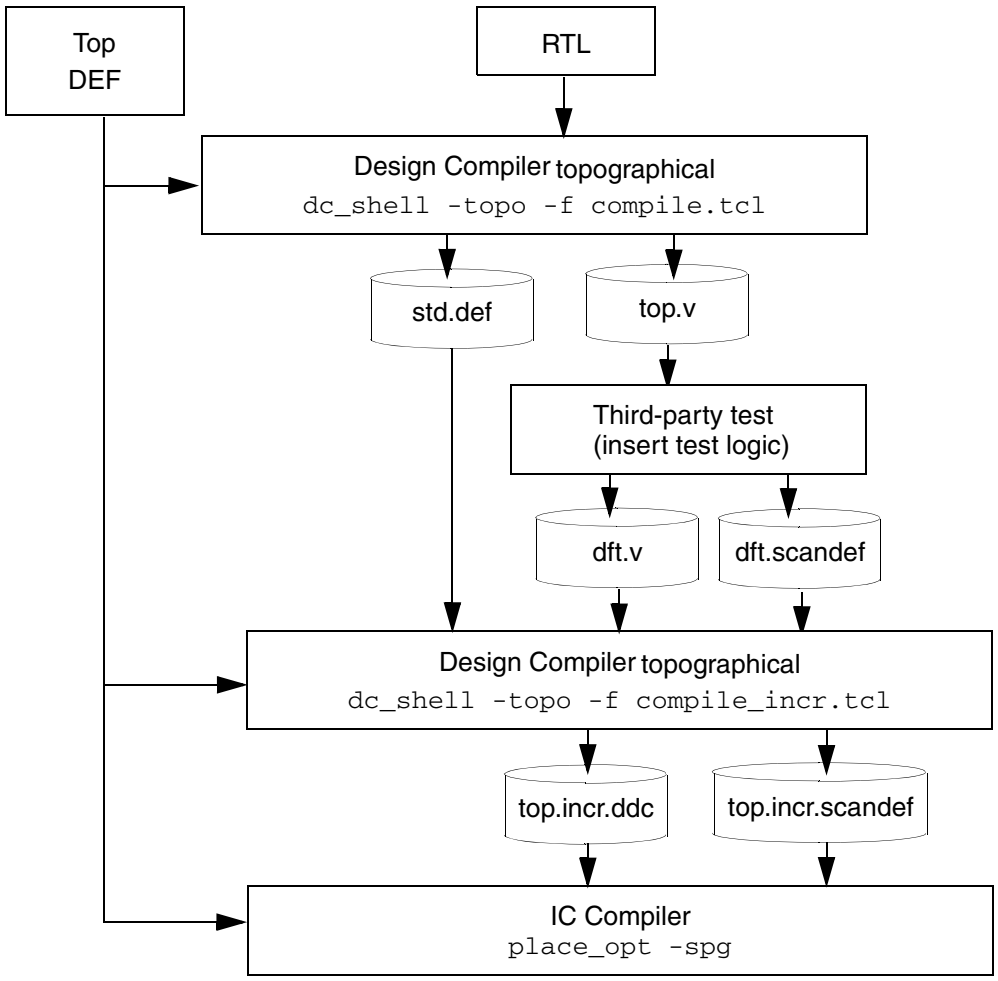
Physical guidance information is automatically saved with your design when you use the .ddc or Milkyway file format. However, some flows require you to use a Verilog or VHDL (ASCII) netlist output format. Verilog and VHDL formats are not suitable for physical guidance storage. Therefore, you need to save the physical guidance information in a separate DEF file along with the gate-level Verilog or VHDL netlist. This section provides an overview of the incremental ASCII flow using a third-party DFT flow as an example.

Saving the physical guidance information in DEF format allows you to retain the placement of the standard cells after the initial `compile_ultra -spg` command run. The standard cell placement is restored for the netlist coming back from test insertion and is used in the incremental compilation when you run the `compile_ultra -incremental -spg` command. During the incremental step, Design Compiler uses the placement information that was saved in the DEF file as the seed placement, allowing better correlation between the

`compile_ultra -spg` and `compile_ultra -incremental -spg` steps. You must use a fully mapped netlist in this flow.

Figure 10-27 shows the physical guidance flow for third-party DFT flows.

Figure 10-27 Physical Guidance Third-Party Test Flow



For more information about using the incremental ASCII flow using a third-party flow as an example, see chapter 12, “Incremental ASCII Flow With a Third-Party DFT Flow Example” in the *Design Compiler Optimization Reference Manual*.

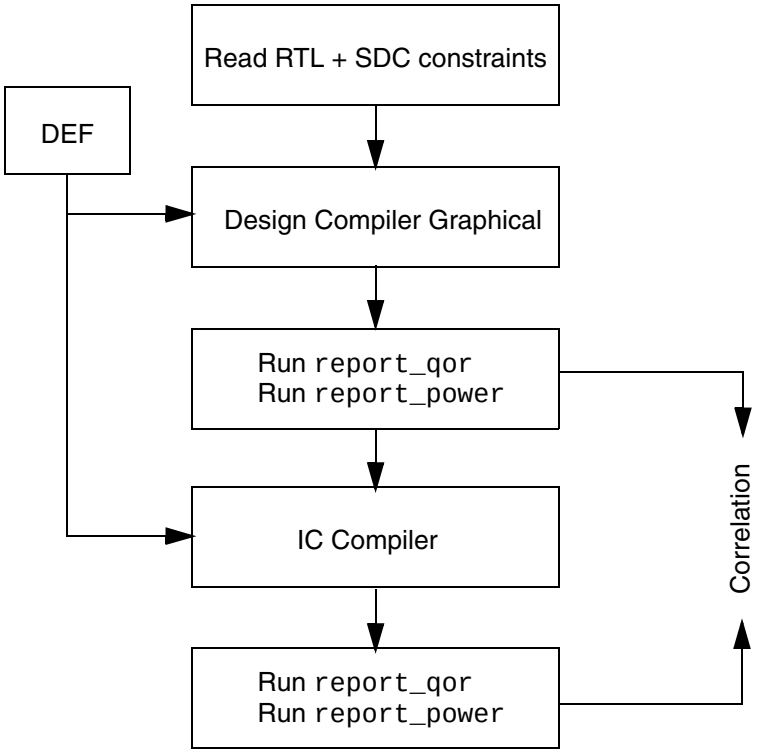
Reporting Physical Guidance Information

The physical guidance information is stored with the design in binary format. It is not available as a standalone file. To view the physical guidance information, you must run IC Compiler and execute the `place_opt -spg` command.

For correlation result comparison between non-physical guidance and physical guidance flows, you should compare post-synthesis results against respective post-placement results for each flow. However, you should measure and compare post-placement results instead of post-synthesis results when looking at overall QoR comparison between non-physical guidance and physical guidance flows.

Figure 10-28 shows how correlation results are measured between Design Compiler and IC Compiler.

Figure 10-28 Correlation Measurement Between Design Compiler and IC Compiler



Physical Guidance Limitations

Physical guidance has the following limitations:

- You cannot combine ILMs and block abstractions in the same flow.
- IC Compiler does not support ILMs and block abstractions created by Design Compiler. Therefore, in the hierarchical flow between Design Compiler and IC Compiler, you must use ILMs or block abstractions created by IC Compiler.
- Design Compiler does not support the `-top` option with the `compile_ultra` command in the physical guidance flow.

Creating and Modifying Floorplans Using Floorplan Exploration

This section describes how to use floorplan exploration, which allows you to create and modify floorplans using Design Compiler Graphical.

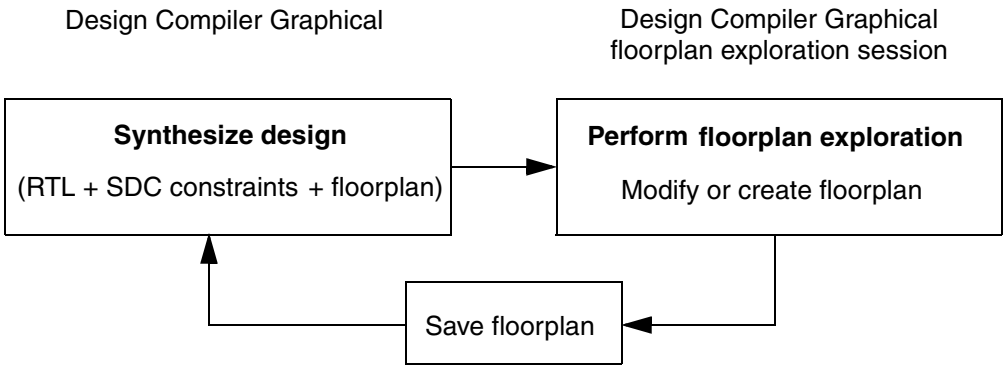
This section contains the following subsections:

Floorplan Exploration Overview

Floorplan exploration allows you to perform floorplanning tasks, including floorplan analysis, floorplan creation, and floorplan modification from within the synthesis environment. Design Compiler Graphical floorplan exploration uses the IC Compiler floorplanning tools in the IC Compiler layout window. Although you use the IC Compiler layout window, the interface between floorplan exploration in Design Compiler Graphical and the IC Compiler layout window is transparent, allowing you to move seamlessly between the Design Vision and IC Compiler layout windows.

Figure 10-29 shows a typical floorplan exploration flow. You read an RTL file, specify the physical and logic libraries, define the Synopsys design constraints, specify a floorplan (if you have one), and synthesize the design in Design Compiler Graphical. After synthesis, you evaluate the QoR results and use floorplan exploration to create a new floorplan or improve an existing floorplan, as needed.

Figure 10-29 Floorplan Exploration Flow



Enabling Floorplan Exploration

Before you begin floorplan exploration, you must have the following:

- A synthesized design

  Design Compiler Graphical issues an error message if it finds any unmapped logic.

- Both the physical and logic libraries

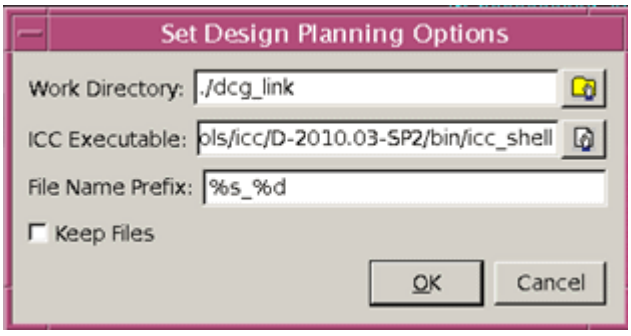
  All cells in the design must link to the physical and logic libraries.

After you have a synthesized design that links to the physical and logic libraries, you must set the design planning options, as described in the following steps:

1. In Design Compiler Graphical, open the GUI by entering the `gui_start` command.
2. In the GUI, open the layout window.
3. Choose Floorplan > Set Design Planning Options.

   The Set Design Planning Options dialog box appears, as shown in Figure 10-30.

Figure 10-30 Set Design Planning Options Dialog Box



4. In the Work Directory box, enter the name of your working directory with the relative path from the current directory.

   This is where the floorplanning scripts, the floorplan, and any other necessary files are stored.

5. In the ICC Executable box, enter the name and location of the IC Compiler executable.

   By default, Design Compiler uses the icc_shell executable specified by the $path variable.

6. In the File Name Prefix box, enter the prefix you want to use in the file name for any generated files, such as the floorplanning scripts and floorplan files.

   The default file prefix naming style is *%s_%d* where *%s* is the design name and *%d* is the process ID.

7. (Optional) Select the Keep Files option if you want to retain the files that are created during and after the floorplan exploration session, such as the floorplanning scripts, floorplan files, and so on.

   This option is deselected by default, which means that the generated files are removed when the floorplan exploration session is closed.

8. Click OK.

You can set the design planning options at the command line by using the `set_icc_dp_options` and `start_icc_dp` commands. For more information, see Using Floorplan Exploration With a dc_shell Script.

Running Floorplan Exploration

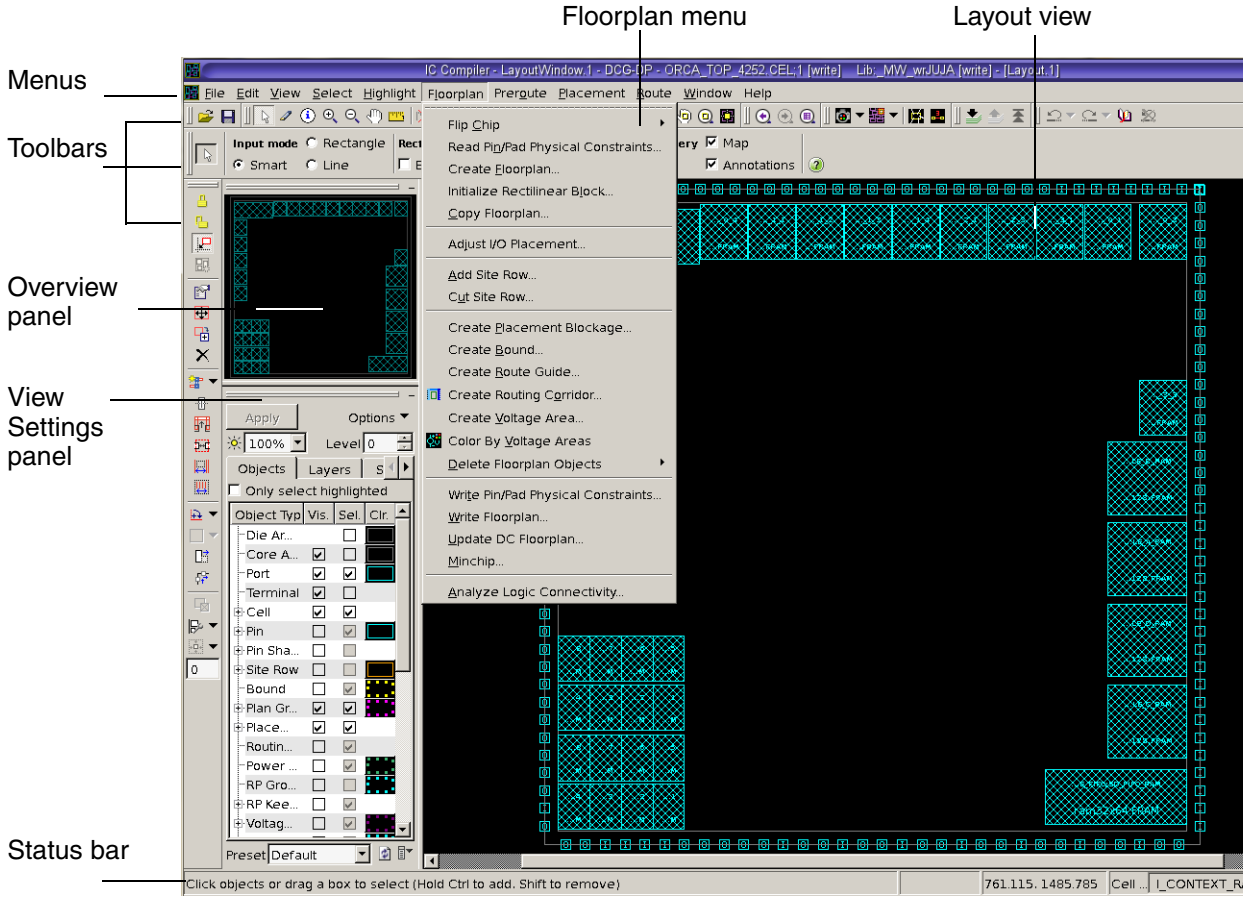
After you set the design planning options as described in Enabling Floorplan Exploration, you can start a floorplan exploration session by choosing Floorplan > Start Design Planning in the GUI. When you do this, the following actions are performed automatically:

- Design Compiler Graphical invokes floorplan exploration with simplified floorplanning menus in the IC Compiler layout window.
- Design Compiler Graphical transfers the design and its floorplan to the IC Compiler layout window.
- Design Compiler Graphical transfers all SDC constraints, attributes, and library setup files to IC Compiler for floorplanning.
- All Design Compiler Graphical windows and Design Compiler Graphical shells are disabled until you exit the IC Compiler floorplanning session.

Using the Floorplan Exploration GUI

Floorplan exploration in Design Compiler Graphical uses the IC Compiler layout window. By default, when you enable floorplan exploration, Design Compiler Graphical configures the IC Compiler layout window so that it includes only the menus and menu commands you need to perform Design Compiler Graphical floorplanning. Figure 10-31 shows the simplified floorplanning menu structure.

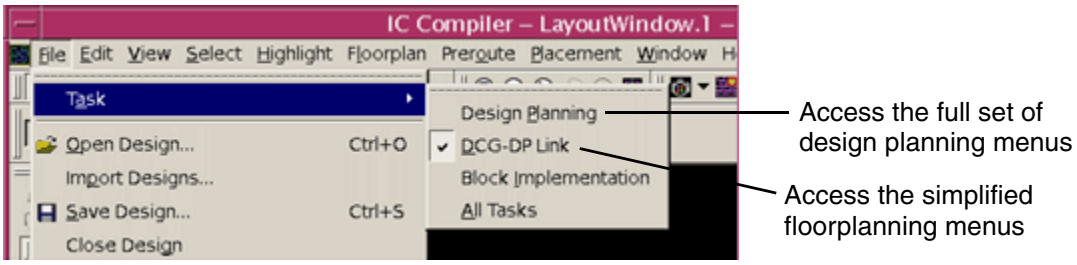
Figure 10-31 IC Compiler Simplified Layout Window Features



If you are an experienced IC Compiler user, you can use the full set of design planning menus.

You can access the full set of IC Compiler design planning menus by choosing File > Task > Design Planning, as shown in Figure 10-32. To switch from the full design planning menu structure back to the simplified menu structure, choose File > Task > DCG-DP Link.

Figure 10-32 Switching Between Simplified and Full Design Planning Menu Structures



Any work you complete using the full set of IC Compiler design planning menus is available when you switch back to the simplified floorplanning menus.

For general information about working in the IC Compiler layout window, see the IC Compiler Help system and the *IC Compiler Design Planning User Guide*.

Creating and Editing Floorplans

Whether you use the simplified Design Compiler Graphical floorplanning menus or the full set of IC Compiler design planning menus, the IC Compiler layout window provides interactive tools and commands that you can use to create or modify your floorplan. You can

- Use editing tools to move, resize, copy, split, reshape, align, distribute, spread, or remove objects.
- Use editing commands to rotate or expand objects or to change cell orientations.

Shortcut keys are available for the most frequently performed editing operations. Online Help pages are provided for each of the editing tools.

In addition, you can use commands on the Floorplan menu to create physical objects such as placement blockages, bounds, and routing keepouts. When you create an object, you can specify its coordinates or draw the object in the layout view.

For more information about working with editing tools and commands in the IC Compiler layout window, see the IC Compiler Help system. For sample flows that show how to create a floorplan using Design Compiler Graphical floorplan exploration and how to modify a floorplan to improve timing or congestion, see SolvNet article 030613, “Design Compiler Graphical Floorplan Exploration Application Note.”

Analyzing the Data Flow for Macro Placement

Floorplan exploration allows you to analyze and improve the data flow through the physical design using the IC Compiler design planning data flow analysis tools. Information about the data flow can help you to decide what to do next to improve the placement of hard macros in your design.

After performing your initial macro placement in IC Compiler, you should analyze your floorplan to find areas of high congestion. If your macro placement appears to be causing congested hot spots, you can use the data flow analysis tools to examine hard macro placement and connectivity and to edit the macro placement.

To open the data flow analyzer window, choose Floorplan > Analyze Logic Connectivity. Alternatively, you can use the `analyze_logic_connectivity` command.

For information about using the data flow analysis tools, see the IC Compiler Help system and the *IC Compiler Implementation User Guide*.

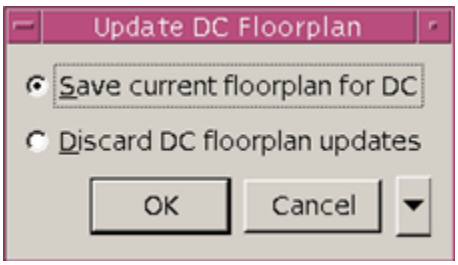
Saving the Floorplan or Discarding Updates

You can save or discard floorplan updates in Design Compiler Graphical from the IC Compiler floorplanning session at any time using the “Update DC Floorplan” command. To save or discard the floorplan, perform the following steps:

1. Choose Floorplan > Update DC Floorplan.

   The Update DC Floorplan dialog box appears, as shown in Figure 10-33.

Figure 10-33 Update DC Floorplan Dialog Box



   The “Update DC Floorplan” command is available only in the simplified floorplanning menu structure for Design Compiler Graphical. For information about switching from the full set of design planning menus to the simplified floorplanning menu structure, see Using the Floorplan Exploration GUI.

2. Choose one of the following options:

   - To save the current floorplan in Design Compiler Graphical, select the “Save current floorplan for DC” option.

     If you have previously saved a floorplan, the tool overwrites it.

   - To remove a previously saved floorplan, select the “Discard DC floorplan updates” option.

3. Click OK.

Saving the Floorplan into a Tcl Script File or DEF File

You can also save the floorplan into a Tcl script file or a DEF file during the floorplan exploration session. However, the Tcl or DEF file is not automatically read in Design Compiler Graphical.

To save the floorplan into a Tcl script file, choose Floorplan > Write Floorplan, and specify the required options in the Write Floorplan dialog box.

Alternatively, you can use the `write_floorplan` command with its required options at the tool prompt.

To save the floorplan in a DEF file, choose File > Export > Write DEF, and specify the required options in the Write DEF dialog box.

Alternatively, you can use the `write_def` command with its required options at the tool prompt.

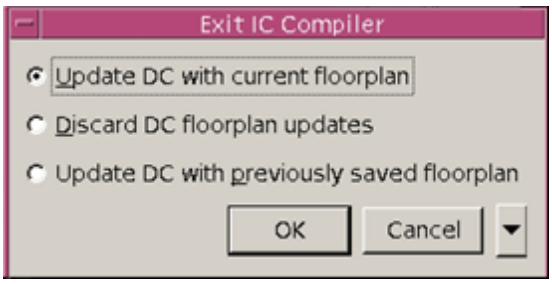
Exiting the Session

To exit the floorplan exploration session and save your floorplan for synthesis or discard the floorplan, perform the following steps:

1. Choose File > Exit in the menu.

   The Exit IC Compiler dialog box appears, as shown in Figure 10-34.

Figure 10-34 Exit IC Compiler Dialog Box



2. Choose one of the following options:

   - Select the “Update DC with current floorplan” option to save the current floorplan in Design Compiler Graphical and exit the floorplan exploration session.
   - To exit the floorplan exploration session and discard the floorplan, including any previously saved floorplans, select the “Discard DC floorplan updates” option.
   - To keep the floorplan that you saved previously in Design Compiler Graphical, discard any subsequent changes that you have not saved, and exit the floorplan exploration session, select the “Update DC with previously saved floorplan” option.

3. Click OK.

When you exit the floorplan exploration session, the Design Compiler Graphical windows and shell interface become active.

If you update the floorplan, the tool closes the original Design Vision layout window. To see the floorplan changes, open a new Design Vision layout window.

Incremental or Full Synthesis after Floorplan Changes

The floorplan changes affect the design during synthesis. After you create a floorplan or modify an existing floorplan, you can perform incremental synthesis by using the `compile_ultra -incremental` command for a quick assessment of the floorplan's QoR benefits. However, incremental synthesis might not meet the requirements for achieving good QoR results or the requirements for a final implementation flow. In these cases, perform full synthesis using the `compile_ultra` command.

To prepare for full synthesis, you must save the floorplan into a Tcl script file by running the `write_floorplan` command at the tool prompt, as shown:

```
write_floorplan -all DESIGN.fp
```

The `-all` option is recommended for resynthesizing the design with the new or modified floorplan.

Use the `read_floorplan` command to read the floorplan in Design Compiler for synthesis. After you read the RTL file, the SDC constraints, and the new floorplan file, you perform full synthesis by using the `compile_ultra` command.

For information about the `read_floorplan` and `write_floorplan` commands, see Using Floorplan Physical Constraints. For information about synthesizing the design, see Overview of Topographical Technology.

Using Floorplan Exploration With a dc_shell Script

You can perform floorplan exploration from a script. Scripts allow you to create and modify floorplans on the command line.

Use the following commands in Design Compiler Graphical to perform floorplan exploration with a script:

1. Use the `set_icc_dp_options` command to specify the IC Compiler executable and other setup requirements.

   The following options are available with the `set_icc_dp_options` command:

   - `-work_dir` *directory*

     Specifies the directory where the temporary files are created and stored.

- `-icc_executable` *executable*

  Specifies the path to icc_shell.

- `-check`

  Performs permission checks and reports any errors that would prevent the `start_icc_dp` command from successfully launching the floorplan exploration session.

- `-file_name_prefix` *file_name*

  Specifies the prefix for the name of any generated files.

- `-keep_files`

  Specifies to keep the temporary files.

2. (Optional) Use the `report_icc_dp_options` command to report the options specified by the `set_icc_dp_options` command.

   If you do not specify any options using the `set_icc_dp_options` command, `report_icc_dp_options` reports the default settings.

   The following options are available with the `report_icc_dp_options` command:

   - `-check`

     Performs permission checks and reports any errors that would prevent the `start_icc_dp` command from successfully launching the floorplan exploration session.

   - `-verbose`

     Prints verbose messages.

3. Use the `start_icc_dp` command to launch the floorplan exploration session.

   The `start_icc_dp` command uses the IC Compiler executable specified by the `set_icc_dp_options` command. To source a script, use the `-f` option with the `start_icc_dp` command:

   ```
   dc_shell-topo> start_icc_dp -f ../../scripts/script_file_name.tcl
   ```

   The `start_icc_dp` command sources the file specified with the `-f` option and runs the script in floorplan exploration.

   The following options are available with the `start_icc_dp` command:

   - `-check_only`

     Performs permission checks and reports any errors that would prevent the `start_icc_dp` command from successfully launching the floorplan exploration session.

- `-verbose`

  Prints verbose messages.

- `-f` *`file_name`*

  Specifies the path to the script file to be sourced.

Note:

If you run floorplan exploration from a batch script, by using the `-f` option on the command line when you start the tool, the `start_icc_dp` command closes the GUI automatically before starting the floorplan exploration session.

To ensure that IC Compiler saves the floorplan to Design Compiler Graphical when exiting the floorplan exploration session, you must include the `update_dc_floorplan` command in the IC Compiler floorplanning script after the floorplan creation or modification steps. After the `update_dc_floorplan` command, you must include the `exit` command in the IC Compiler floorplanning script to return to the Design Compiler Graphical shell interface:

```
# After many IC Compiler floorplanning script commands:
update_dc_floorplan
exit
```

The following example shows a Design Compiler Graphical script used in a floorplan exploration session. It sources an IC Compiler floorplanning script called iccdp_script.tcl.

```
compile_ultra -scan
set_icc_dp_options -icc_executable
   "/global/apps5/icc_2010.03-SP1-1/bin/icc_shell"
start_icc_dp -f iccdp_script.tcl
compile_ultra -scan -incremental
```

Handling Black Boxes

Design Compiler Graphical supports floorplan exploration with netlists that contain black box cells. If a design has black box cells, you do not need to perform any additional steps or modify the floorplan exploration flow. The black boxes that are defined or created in Design Compiler Graphical are transferred to IC Compiler design planning automatically. Using IC Compiler, you can edit the floorplan with black boxes during the floorplanning session and transfer the modifications back to Design Compiler Graphical.

Handling Physical Hierarchies and Block Abstractions

Design Compiler Graphical transfers Design Compiler topographical physical hierarchies and Design Compiler topographical block abstractions to IC Compiler design planning automatically during floorplan exploration. Physical hierarchies are transferred as IC Compiler ILMs, whereas block abstractions are transferred as macros with quick timing models in the block. For multicorner-multimode designs, only the current scenario timing information is transferred to IC Compiler.

When transferring Design Compiler topographical physical hierarchies and block abstractions to IC Compiler for design planning, Design Compiler Graphical automatically sets the `dct_physical_hier` attribute to `true` on the instances.

To return all the Design Compiler topographical hierarchical blocks, including physical hierarchies and block abstractions, use the `dct_physical_hier` attribute as shown:

```
icc_shell> get_cells -hierarchical -filter dct_physical_hier==true
{GPRs Multiplier}
```

The Design Compiler topographical physical hierarchies and block abstractions are preserved when brought back to Design Compiler Graphical.

Floorplan Exploration Limitation

Relative placement constraints are not passed to and from the IC Compiler floorplanning session.

Optimizing Multicorner-Multimode Designs in Design Compiler Graphical

Multicorner-multimode design optimization is a feature available only in the Design Compiler Graphical tool. This feature enables you to analyze and optimize designs across multiple modes and multiple corners concurrently. For more details about other Design Compiler Graphical features, see Using the Design Compiler Graphical Tool.

This section describes multicorner-multimode design optimization, in the following subsections:

- Multicorner-Multimode Concepts
- Multicorner-Multimode Feature Support
- Unsupported Features
- Concurrent Multicorner-Multimode Optimization and Timing Analysis
- Basic Multicorner-Multimode Flow
- Setting Up the Design for a Multicorner-Multimode Flow
- Handling Libraries in the Multicorner-Multimode Flow
- Scenario Management Commands
- Reporting Commands
- Supported SDC Commands

- Multicorner-Multimode Script Example
- Using ILMs in Multicorner-Multimode Designs

Multicorner-Multimode Concepts

Designs must often operate under multiple operating conditions, often called corners, and in multiple modes. Such designs are referred to as multicorner-multimode designs. Design Compiler Graphical extends the topographical technology to analyze and optimize these designs across multiple modes and multiple corners concurrently. The multicorner-multimode feature also provides ease-of-use and compatibility between flows in Design Compiler and IC Compiler due to the similar user interface. The following terms are commonly used to describe multicorner-multimode technology:

- Corner

  A *corner* is defined as a set of libraries characterized for process, voltage and temperature (PVT) variations. Corners are not dependent on functional settings; they are meant to capture variations in the manufacturing process, along with expected variations in the voltage and temperature of the environment in which the chip will operate.

- Mode

  A *mode* is defined by a set of clocks, supply voltages, timing constraints, and libraries. It can also have annotation data, such as SDF or parasitics files. Multicorner-multimode designs can operate in many modes such as the test mode, mission mode, standby mode and so forth.

- Scenario

  A *scenario* is a combination of modal constraints and corner specifications. In a design, the tool uses a scenario or a set of scenarios as the unit for multimode-multicorner analysis and optimization. Some constraints can be part of both the mode and corner specification. Optimization of multicorner-multimode design involves managing the scenarios of the design. For more details on scenario management, see Scenario Management Commands.

Creating a Scenario

To define modes and corners, you use the `create_scenario` command. A scenario definition includes commands that specify the TLUPlus libraries, operating conditions, and constraints, as shown in Example 10-47, which defines the s1 scenario. A scenario definition must include the `set_operating_conditions` and `set_tlu_plus_files` commands. The following sections describe these commands along with the associated library setup information that is needed to run multicorner-multimode design optimization.

Example 10-47 Basic Scenario Definition

```
create_scenario s1
set_operating_conditions WORST -library stdcell.setup.typ.db:stdcell_typ
set_tlu_plus_files -max_tluplus design.tlup -tech2itf_map layermap.txt
read_sdc s1.sdc
```

Multicorner-Multimode Feature Support

The multicorner-multimode feature in Design Compiler Graphical provides compatibility between flows in Design Compiler and IC Compiler through the use of a similar user interface.

Keep the following points in mind when running multicorner-multimode optimizations:

- All options of the `compile_ultra` command are supported.
- Multicorner-multimode technology is supported only in topographical mode with the DC-Extension license. Multicorner-multimode technology is not supported in DC Expert.
- Leakage power optimizations are supported.
- Dynamic power optimizations are supported.
- The UPF flow is supported only for multivoltage designs.

Unsupported Features

The following features are not supported in Design Compiler Graphical for multicorner-multimode designs:

- Power-driven clock gating is not supported.

  However, if you use the `compile_ultra -gate_clock` or the `insert_clock_gating` commands, clock-gate insertion is performed on the design, independent of the scenarios.
- Clock tree estimation is not supported.
- k-factor scaling is not supported.

  Because multicorner-multimode design libraries do not support the use of k-factor scaling, the operating conditions that you specify for each scenario must match the nominal operating conditions of one of the libraries in the list of the link libraries.
- The `set_min_library` command is not supported for individual scenarios.

  The `set_min_library` command applies to all scenarios. If you use set_min_library to define one scenario, the tool will use the library for all scenarios.

Concurrent Multicorner-Multimode Optimization and Timing Analysis

Concurrent multicorner-multimode optimization works on the worst violations across all scenarios, eliminating the convergence problems observed in sequential approaches.

Timing analysis is carried out on all scenarios concurrently, and cost is measured across all scenarios for timing and design rules. As a result, the timing and constraint reports show worst-case timing across all scenarios.

To run timing analysis, use one of the following two methods:

- Traditional minimum-maximum analysis

  To use this analysis method, define your analysis type as `bc_wc`, for example,

  ```
  set_operating_conditions -analysis_type bc_wc
  ```

- Early-late analysis

  To use this analysis method, define your analysis type as `on_chip_variation`, for example,

  ```
  set_operating_conditions -analysis_type on_chip_variation
  ```

  PrimeTime also uses the on-chip variation (OCV) method.

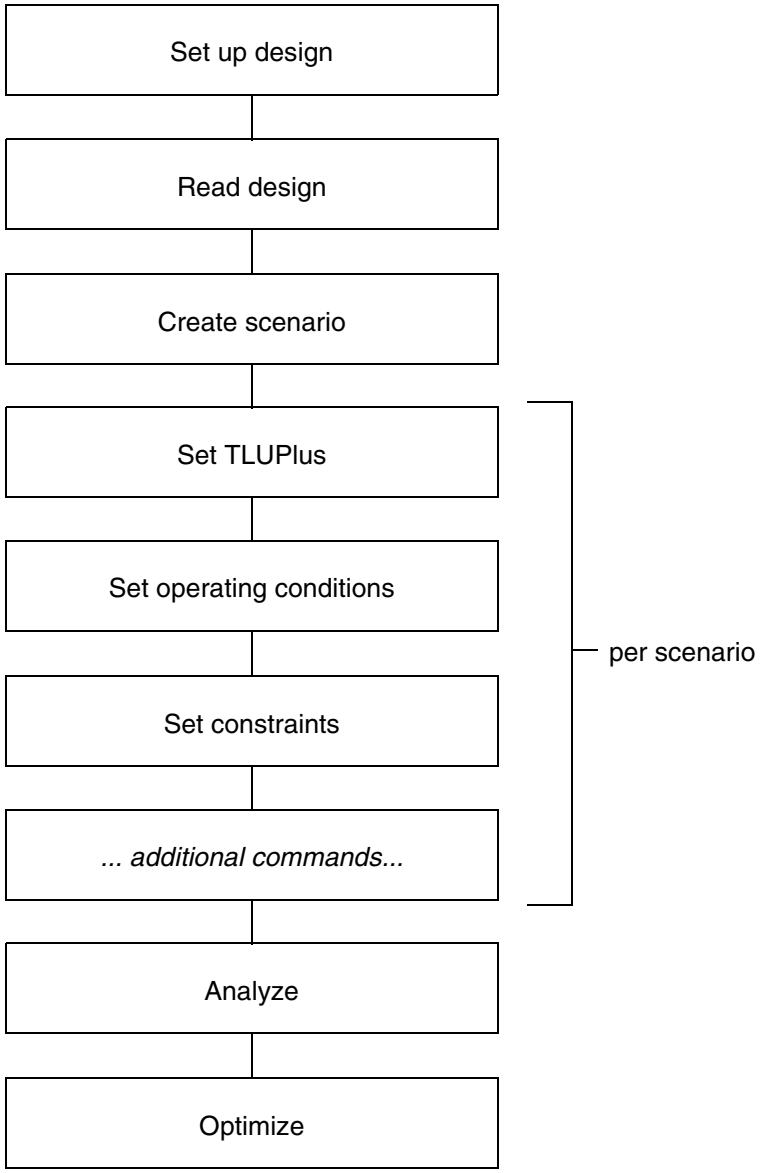
Basic Multicorner-Multimode Flow

Figure 10-35 shows the basic multicorner-multimode flow. The key step in multicorner-multimode optimization involves creating the scenarios. You use the `create_scenario` command to create the scenarios. You can create multiple scenarios. For each scenario, you set constraints specific to the scenario mode and you set operating conditions specific to the scenario corner. For an example script, see Multicorner-Multimode Script Example.

Scenario definitions include commands that specify the TLUPlus libraries, operating conditions, and constraints. However, other commands can be included. For example, you can use the `set_leakage_optimization` command to control leakage power on a per-scenario basis or you can use the `read_sdf` command to set the correct net RC and pin-to-pin delay information in the respective scenarios.

After you configure all the scenarios, you can activate a subset of these scenarios by using the `set_active_scenarios` command.

Figure 10-35 Basic Multimode Flow



Setting Up the Design for a Multicorner-Multimode Flow

To setup a design for a multicorner-multimode flow, you must specify the TLUPlus files, operating conditions, and Synopsys Design Constraints for each scenario. Design Compiler uses the nominal process, voltage, and temperature (PVT) values to group the libraries into different sets. Libraries with the same PVT values are grouped into the same set. For each

scenario, the PVT of the maximum operating condition is used to select the appropriate set. Setup considerations are described in the following sections:

- Specifying TLUPlus Files
- Specifying Operating Conditions
- Specifying Constraints

Specifying TLUPlus Files

Use the `set_tlu_plus_files` command to specify the TLUPlus files for each scenario, as shown in Example 10-48.

Example 10-48 Specifying TLUPlus Files for a Scenario

```
create_scenario s1
set_operating_conditions WORST -library stdcell.setup.typ.db:stdcell_typ
set_tlu_plus_files -max_tluplus design.tlup -tech2itf_map layermap.txt
read_sdc s1.sdc
```

If you do not specify a TLUPlus file, the tool returns an error message similar to the following:

```
Error: tlu_plus files are not set in this scenario s1.
     RC values will be 0.
```

If a TLUPlus library is not correct, the tool issues the following error message:

```
Error: TLU+ sanity check failed (OPT-1429)
```

If you want to enable temperature scaling, the TLUPlus files must contain the `GLOBAL_TEMPERATURE` and `CRT1` variables. The `CRT2` variable is optional. The following example is an excerpt from a TLUPlus file:

```
TECHNOLOGY = 90nm_lib
GLOBAL_TEMPERATURE = 105.0
CONDUCTOR metal8 {THICKNESS= 0.8000
   CRT1=4.39e-3 CRT2=4.39e-7
...
```

Specifying Operating Conditions

You must define the operating condition for each scenario. You specify different operating conditions for different scenarios using the `set_operating_conditions` command, as shown in Example 10-49.

Example 10-49 Specifying Operating Conditions for a Scenario

```
create_scenario s1
set_operating_conditions SLOW_95 -library max_vmax_v95_t125
```

```
set_tlu_plus_files -max_tluplus design.tlup -tech2itf_map layermap.txt
read_sdc s1.sdc
```

If you do not define an operating condition for a scenario, the tool issues MV-020 and MV-021 warnings.

Specifying Constraints

For each scenario, you must specify the Synopsys Design Constraints (SDC) specific to that scenario, as shown in Example 10-50.

Example 10-50 Specifying SDC Constraints for a Scenario

```
create_scenario s1
set_operating_conditions WORST -library stdcell.setup.typ.db:stdcell_typ
set_tlu_plus_files -max_tluplus design.tlup -tech2itf_map layermap.txt
read_sdc s1.sdc
```

The tool discards any previous scenario-specific constraints after you execute the `create_scenario` command and reports an MV-020 warning, as shown in Example 10-51.

Example 10-51 Tool Removes Previous Constraints

```
dc_shell-topo> create_scenario s1
Warning: Any existing scenario-specific constraints
      are discarded. (MV-020)
dc_shell-topo> report_timing
Warning: No operating condition was set in scenario s1 (MV-021)
```

Handling Libraries in the Multicorner-Multimode Flow

The following sections discuss how to handle libraries in multicorner-multimode designs:

- Using Link Libraries That Have the Same PVT Nominal Values
- Using Unique PVT Names to Prevent Linking Problems
- Unsupported k-factors
- Automatic Detection of Driving Cell Library
- Defining Minimum Libraries

Using Link Libraries That Have the Same PVT Nominal Values

The link library lists all the libraries that are to be used for linking the design for all scenarios. Furthermore, because several libraries are often intended for use with a particular scenario, such as a standard cell library and a macro library, Design Compiler automatically groups

the libraries in the link library list into sets and identifies which set must be linked with each scenario.

The tool groups libraries according to the PVT value of the library. Libraries with the same PVT values are grouped into the same set. The tool uses the PVT value of a scenario's maximum operating condition to select the appropriate set for the scenario.

If the tool finds no suitable cell in any of the specified libraries, an error is reported as shown in the following example,

```
Error: cell TEST_BUF2En_BUF1/Z (inx4) is not characterized
      for 0.950000V, process 1.000000,
      temperature -40.000000. (MV-001)
```

You should verify the operating conditions and library setup. You must fix this error before you can optimize your design.

Link Library Example

To understand how library linking works, consider Table 10-17.

Table 10-17 shows the libraries in the link library list, their nominal PVT values, and the operating condition, if any, that is specified in each library. The design has instances of combinational, sequential, and macro cells.

Table 10-17 Link Libraries With PVT and Operating Conditions

Link library (in order)	Nominal PVT	Operating conditions in library (PVT)
Combo_cells_slow.db	1/0.85/130	WORST (1/0.85/130)
Sequentials_fast.db	1/1.30/100	None
Macros_fast.db	1/1.30/100	None
Macros_slow.db	1/0.85/130	None
Combo_cells_fast.db	1/1.30/100	BEST (1/1.3/100)
Sequentials_slow.db	1/0.85/130	None

To create the scenario, s1, with cell instances linked to the Combo_cells_slow, Macros_slow, and Sequential_slow libraries, you run

```
dc_shell-topo> create_scenario s1
dc_shell-topo> set_operating_conditions -max WORST -library slow
```

where the library “slow” is defined within the Combo_cells_slow.db file. Note that using the `-library` option with the `set_operating_conditions` command helps the tool identify the correct PVT for the operating conditions. The PVT of the maximum operating condition is used to find the correct matches in the link library list during linking.

Using this library linking method, you can link libraries that do not have operating condition definitions. The method also enables you to have multiple library files. For example, you can have one library file for standard cells, another for macros, and so forth.

Inconsistent Libraries Warning

When you use multiple libraries, if library cells with the same name are not functionally identical or do not have identical sets of library pins with the same name and order, the tool issues a warning stating that the libraries are inconsistent.

You should run the `check_library` command before running a multicorner-multimode flow, as shown in the following example,

```
set_check_library_options -mcmm
check_library -logic_library_name {a.db b.db}
```

When you use the `-mcmm` option with the `set_check_library_options` command, the `check_library` command performs multicorner-multimode specific checks, such as determining operating condition or power-down inconsistencies. When inconsistencies are detected, the tool generates a report that lists the inconsistencies. In addition, the tool issues the following summary information message:

```
Information: Logic library consistency check FAILED for MCMM.
(LIBCHK-360)
```

When you get a LIBCHK-360 message, check the report to identify the cause of the problem and fix the library inconsistencies. The LIBCHK-360 man page describes possible causes for the library inconsistencies.

Setting the dont_use Attribute on Library Cells in the Multicorner-Multimode Flow

When you set the `dont_use` attribute on a library cell, the multicorner-multimode feature requires that all characterizations of this cell have the `dont_use` attribute. Otherwise, the tool might consider the libraries as inconsistent. You can use the wildcard character to set the `dont_use` attribute as follows:

```
set_dont_use */AN2
```

When the library cells that have a `dont_use` attribute have a pin order that does not match exactly in the libraries of various corners, the tool continues with the flow without any error or warning messages. If you remove the `dont_use` attribute on these cells, the tool issues the MV-087 error message.

Note that you do not have to issue the command multiple times to set the `dont_use` attribute on all characterizations of a library cell.

Using Unique PVT Names to Prevent Linking Problems

To prevent linking problems, make sure your PVT operating conditions have unique names. If the maximum libraries associated with each scenario do not have distinct PVT values, your cell instances might be incorrectly linked, resulting in incorrect timing values. This happens because the nominal PVT values that are used to group the link libraries into sets group the maximum libraries of different corners into one set. Consequently, the cell instances are linked to the first cell with a matching type in that set, for example, the first AND2_4 cell, even though the `-library` option is specified for each of the scenario-specific `set_operating_conditions` commands. That is, the `-library` option locates the operating condition and its PVT but not the library to link.

The following paragraphs describe a linking problem due to non-unique PVT names. For the conditions given in Table 10-18, Table 10-19, and Example 10-52, the tool groups the Ftyp.db and TypHV.db libraries into a set with Ftyp.db as the first library in the set. Therefore, the cell instances in scenario s2 are not linked to the library cells in TypHV.db, as intended. Instead, they are incorrectly linked to the library cells in the Ftyp.db library, assuming that all the libraries include the library cells required to link the design.

Table 10-18 shows the libraries in the link library, listed *in order*; their nominal PVT; and the operating condition that is specified in each library.

Table 10-18 Link Libraries With PVT and Operating Conditions

Link library (in order)	Nominal PVT	Operating conditions in library (PVT)
Ftyp.db	1/1.30/100	WORST (1/1.30/100)
Typ.db	1/0.85/100	WORST (1/0.85/100)
TypHV.db	1/1.30/100	WORST (1/1.30/100)
Holdtyp.db	1/0.85/100	BEST (1/0.85/100)

Table 10-19 shows the operating condition specifications for each of the scenarios; Example 10-52 shows the corresponding scenario creation script.

Table 10-19 Scenarios and Their Operating Conditions

	Scenarios			
	s1	**s2**	**s3**	**s4**
Max Opcond (Library)	WORST (Typ.db)	WORST (TypHV.db)	WORST (Ftyp.db)	WORST (Typ.db)
Min Opcond (Library)	None	None	None	BEST (HoldTyp.db)

Example 10-52 Linking Problem Due to Non-Unique PVT Name

```
create_scenario s1
set_operating_conditions WORST -library Typ.db:Typ
create_scenario s2
set_operating_conditions WORST -library TypHV.db:TypHV
create_scenario s3
set_operating_conditions WORST -library Ftyp.db:Ftyp
create_scenario s4
set_operating_conditions \
      -max WORST -max_library Typ.db:Typ \
      -min BEST -min_library HoldTyp.db:HoldTyp
```

Ambiguous Libraries Warning

The tool issues a warning if your design uses any libraries containing cells with the same name and same nominal PVT. The warning states that the libraries are ambiguous and identifies which libraries are being used and which are being ignored.

Unsupported k-factors

Multicorner-multimode design libraries do not support k-factor scaling. Therefore, the operating conditions that you specify for each scenario must match the nominal operating conditions of one of the libraries in the link library list.

Automatic Detection of Driving Cell Library

In multicorner-multimode flow, the operating condition setting is different for different scenarios. To build the timing arc for the driving cell, different technology libraries are used for different scenarios. You can specify the library using the `-library` option of the

`set_driving_cell` command. But specifying the library is optional because the tool can automatically detect the driving cell library.

When you specify the library using the `-library` option of the `set_driving_cell` command, the tool searches for the specified library in the link library set. If the specified library exists, it is used. If the specified library does not exist in the link library, the tool issues the UID-993 error message as follows:

```
Error: Cannot find the specified driving cell in memory.(UID-993)
```

When you do not use the `-library` option of the `set_driving_cell` command, the tool searches all the libraries for the matching operating conditions. The first library in the link library set that matches the operating condition is used. If no library in the link library set matches the operating condition, the first library in the link library set that contains the matching library cell is used. If no library in the link library set contains the matching library cell, the tool issues the UID-993 error message.

Defining Minimum Libraries

Minimum libraries are usually defined with the `set_operating_conditions` command. You can use the `set_min_library` command, but it is not scenario-specific. If you use set_min_library to define a minimum library for a scenario, the tool uses that library as the minimum library for all scenarios, even if you do not define that library in all your scenarios. If you want to define different minimum libraries for each scenario, use the `set_operating_conditions` command.

Table 10-20 Unsupported *Multiple Minimum Library Configuration*

	Scenarios	
	s1	**s2**
Max library	Slow.db	Slow.db
Min library	Fast_0yr.db	Fast_10yr.db

For example, you could not relate two different minimum libraries – say, Fast_0yr.db and Fast_10yr.db – with the maximum library, Slow.db, in two separate scenarios. The first minimum library you specify would apply to both scenarios. Table 10-20 shows the *unsupported* configuration.

Note, however, that a minimum library can be associated with multiple maximum libraries. As shown in Table 10-21, the minimum library Fast_0yr.db is paired with both the maximum library Slow.db of scenario 1 and the maximum library SlowHV.db of scenario 2.

Table 10-21 Supported Min-Max Library Configuration

	Scenarios	
	s1	**s2**
Max library	Slow.db	SlowHV.db
Min library	Fast_0yr.db	Fast_0yr.db

Scenario Management Commands

Use the following commands to create and manage scenarios:

- `create_scenario`
- `current_scenario`
- `all_scenarios`
- `all_active_scenarios`
- `set_active_scenarios`
- `set_scenario_options`
- `set_preferred_scenario`
- `check_scenarios`
- `remove_scenario`
- `report_scenarios`
- `report_scenario_options`

The following subsections describe how you use these commands to manage scenarios:

- Creating Scenarios
- Defining Active Scenarios
- Scenario Reduction
- Specifying Scenario Options
- Removing Scenarios

Creating Scenarios

You use the `create_scenario` command to create a new scenario. When the first scenario is created, all previous scenario-specific constraints are removed from the design and the following warning is issued:

```
dc_shell-topo> create_scenario s1
Warning: Any existing scenario-specific constraints
      are discarded. (MV-020)
Current scenario is: s1
```

Use the `current_scenario` command to specify the name of the current scenario. Without arguments, the command returns the name of the current scenario. Note that the `current_scenario` command merely specifies the focus scenario and that when you define more than one scenario, the tool performs concurrent analysis and optimization across all scenarios, independent of the current scenario setting.

Use the `set_tlu_plus` command to set the TLUPlus files in the scenario specified by the `current_scenario` command. Note that each scenario must have a set of TLUPlus files specified, or the following error is reported:

```
Error: tlu_plus files are not set in this scenario <name>.
RC values will be 0.
```

Defining Active Scenarios

During concurrent analysis and optimization, you can significantly reduce memory usage and runtime by limiting the number of active scenarios to those that are “essential” or “dominant.” An essential or dominant scenario has the worst slack among all the scenarios for at least one of its constrained objects. (Constrained objects can include delay constraints associated with a pin, the design rule constraints for a net, leakage power, and so on.) Any scenario for which one of its constrained objects is the worst slack value is a dominant scenario.

You use the `set_active_scenarios` command to define active scenarios. Other commands related to active scenarios are `all_scenarios` and `all_active_scenarios`.

Scenario Reduction

Topographical mode automatically performs scenario reduction on the current set of active scenarios to reduce memory and runtime. In general, restricting concurrent analysis and optimization to a subset of dominant scenarios does not lead to a significant difference in QoR. The tool analyzes all the active scenarios and automatically determines a set of dominant scenarios, based on the number of violating endpoints, and optimizes them for timing, power, and design rule. In the current version, topographical mode does not support the `get_dominant_scenarios` command, which IC Compiler supports. However, you can choose a preferred scenario by using the `set_preferred_scenario` command. When you set the preferred scenario, the tool treats it as the most constraining scenario. It still performs

scenario reduction to determine the dominant scenarios but treats the user-specified preferred scenario as the most constraining one.

Specifying Scenario Options

To define specific constraint options, such as leakage power, that you want optimized in a scenario, use the `set_scenario_options` command. You can apply the constraint option to more than one scenario at a time by using the `-scenarios` option. The options can be specified on both active and inactive scenarios. If you do not specify any scenario, the options are applied only to the current scenario.

To enable or disable the leakage power optimization, use the `-leakage_power` option. The default for the `-leakage_power` option is `false`. Set the `-leakage_power` option with the `set_scenario_options` command to `true` to enable leakage power optimization on specific scenarios in a multicorner-multimode design. The following command shows how to enable leakage power optimization on specific scenarios only.

```
dc_shell-topo> set_scenario_options -leakage_power true \
               -scenarios scenario_list
```

To enable or disable the dynamic power optimization, use the `-dynamic_power` option. The default for the `-dynamic_power` option is `false`.

To enable or disable the setup or maximum delay optimization for specified scenarios, use the `-setup` option. The default for the `-setup` option is `true`, so maximum delay optimization is enabled by default. The following example shows how to disable maximum delay optimization on specific scenarios.

```
dc_shell-topo> set_scenario_options -setup false -scenarios
               scenario_list
```

To enable or disable the hold or minimum delay optimization for the specified scenarios, use the `-hold` option. The default for the `-hold` option is `true`. When you set the `-hold` option to `false`, the tool ignores the hold or the minimum delay violations in the specified scenarios.

```
dc_shell-topo> set_scenario_options -reset_all true \
              -scenarios [all_scenarios]
```

To report the scenario options, use the `report_scenario_options` command.

Removing Scenarios

To remove the specified scenarios, use the `remove_scenario` command. All scenario-specific constraints defined in the removed scenario are deleted. For example,

```
dc_shell-topo> remove_scenario
s1 s2
dc_shell-topo> remove_scenario -all
```

```
Removed scenario 's2'
Removed scenario 's1'
dc_shell-topo> all_scenarios
```

Reporting Commands

This section describes the commands that you can use for reporting multicorner-multimode designs in the following subsections:

- report_scenarios Command
- report_scenario_options Command
- Reporting Commands That Support the -scenario Option
- Commands That Report the Current Scenario
- Reporting Examples

report_scenarios Command

The `report_scenarios` command reports the scenario setup information for multicorner-multimode designs. This command reports all the defined scenarios. The scenario specific information includes the logic library used, the operating condition, and TLUPlus files.

The following example shows a report generated by the `report_scenarios` command:

```
****************************************
Report : scenarios
Design : DESIGN1
scenario(s) : SCN1
...
****************************************

All scenarios (Total=4): SCN1 SCN2 SCN3 SCN4
All Active scenarios (Total=1): SCN1
Current scenario     : SCN1

Scenario #0: SCN1 is active.
Scenario options:
Has timing derate: No
Library(s) Used:
 logic library name (File: library.db)

Operating condition(s) Used:
  Analysis Type    : bc_wc
  Max Operating Condition: library:WCCOM
  Max Process    : 1.00
  Max Voltage    : 1.08
```

```
  Max Temperature: 125.00
  Min Operating Condition: library:BCCOM
  Min Process    : 1.00
  Min Voltage    : 1.32
  Min Temperature: 0.00

Tlu Plus Files Used:
  Max TLU+ file: tlu_plus_file.tf
  Tech2ITF mapping file: tf2itf.map
```

report_scenario_options Command

Use the `report_scenario_options` command to report the scenario options set by the `set_scenario_options` command. You can specify a list of scenarios to be reported by using the `-scenarios` option. By default, this command reports scenario options for the current, active scenario.

To illustrate the report generated by the `report_scenario_options` command, consider Example 10-53 where the `set_scenario_options -leakage_power false` command is executed. This command sets the `-leakage_power` option to `false`. With the `-leakage_power` option set to `false`, the tool will not optimize the current, active scenario for leakage power. This status is shown by the `report_scenario_options` report in Example 10-53.

Example 10-53

```
dc_shell-topo> set_scenario_options -leakage_power false
dc_shell-topo> report_scenario_options
****************************************
Report : scenario options
Design : TEST03
...
****************************************

Scenario: MODE1 is active.

setup         : true
hold          : true
leakage_power : false
dynamic_power : true
```

Reporting Commands That Support the -scenario Option

Some reporting commands support the `-scenario` option to report scenario-specific information. You can specify a list of scenarios to the `-scenario` option, and the tool reports scenario details for the specified scenarios.

The following reporting commands support the `-scenario` option:

- `report_timing`
- `report_timing_derate`
- `report_power`
- `report_clock`
- `report_path_group`
- `report_extraction_options`
- `report_tlu_plus_files`
- `report_constraint`

Commands That Report the Current Scenario

The following reporting commands report scenario-specific details for the current scenario. The header section of the report contains the name of the current scenario. No additional options are required to report the scenario-specific details of the current scenario.

- `report_net`
- `report_annotated_check`
- `report_annotated_transition`
- `report_annotated_delay`
- `report_attribute`
- `report_case_analysis`
- `report_ideal_network`
- `report_internal_loads`
- `report_clock_gating_check`
- `report_clock_tree`
- `report_delay_calculation`
- `report_delay_estimate_options`
- `report_transitive_fanout`
- `report_disable_timing`
- `report_latency_adjustment_options`

- report_net
- report_power_calculation
- report_noise
- report_signal_em
- report_timing_derate
- report_timing_requirements
- report_transitive_fanin
- report_crpr
- report_clock_timing

Reporting Examples

This section contains sample reports for some of the multicorner-multimode reporting commands.

report_qor Command

The report_qor command reports by default the QoR details for all the scenarios in the design. The following example shows a report generated by the report_qor command:

```
****************************************
Report : qor
Design : DESIGN1
****************************************
  Scenario 's1'
  Timing Path Group 'reg2reg'
  -----------------------------------
  Levels of Logic:              33.00
  Critical Path Length:        694.62
  Critical Path Slack:        -144.52
  Critical Path Clk Period:    650.00
  Total Negative Slack:      -4533.01
  No. of Violating Paths:      136.00
  -----------------------------------
  Scenario 's2'
  Timing Path Group 'reg2reg'
  -----------------------------------
  Levels of Logic:              33.00
  Critical Path Length:        393.61
  Critical Path Slack:          61.18
  Critical Path Clk Period:    500.00
  Total Negative Slack:          0.00
  No. of Violating Paths:        0.00
  -----------------------------------
```

report_timing -scenario

This command reports timing results for the active scenarios in the design. You can specify a list of scenarios with the `-scenario` option. When the `-scenario` option is not specified only the current scenario is reported.

```
****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : DESIGN1
...
****************************************

* Some/all delay information is back-annotated.

# A fanout number of 1000 was used for high fanout net computations.

Startpoint: TEST_BUF2En
              (input port clocked by clk)
Endpoint: TEST1/TEST2_SYN/latch_3
            (non-sequential rising-edge timing check clocked by clk)
Scenario: s1
Path Group: clk
Path Type: max
Point                                     Incr       Path      Lib:OC
-------------------------------------------------------------------------------
clock clk (rise edge)                     0.00       0.00
clock network delay (propagated)          0.00       0.00
input external delay                    450.00     450.00 f
TEST_BUF2En (in)                          0.00     450.00 f  stdcell_typ:WORST
TEST_BUF2En_BUF1/Z (inx4)                 9.75     459.75 r  stdcell_typ:WORST
U468/Z (inx10)                           10.21     469.96 f  stdcell_typ:WORST
TEST_BUF2En_BUF/Z (inx11)                 8.74     478.70 r  stdcell_typ:WORST
U293/Z (inx11)                            9.30     488.00 f  stdcell_typ:WORST
TEST1/TEST2_SYN/U74963/Z (nr2x4)         12.78     500.78 r  stdcell_typ:WORST
U31662/Z (inx4)                          10.58     511.37 f  stdcell_typ:WORST
TEST1/TEST2_SYN/U75093/Z (aoi21x6)       18.98     530.34 r  stdcell_typ:WORST
U42969/Z (nd2x6)                         14.16     544.51 f  stdcell_typ:WORST
TEST1/TEST2_SYN/U53046/Z (inx8)          13.35     557.86 r  stdcell_typ:WORST
U2765/Z (inx8)                           11.48     569.33 f  stdcell_typ:WORST
U32442/Z (inx6)                           7.61     576.94 r  stdcell_typ:WORST
U33615/Z (nd2x3)                         18.14     595.09 f  stdcell_typ:WORST
U32269/Z (nd2x6)                          8.74     603.82 r  stdcell_typ:WORST
TEST1/TEST2_SYN/clk_gate/EN (cklan2x1)    0.00     603.82 r  stdcell_typ:WORST
data arrival time                                  603.82
```

```
clock clk (rise edge)                   650.00     650.00
clock network delay (propagated)          0.00     650.00
TEST1/TEST2_SYN/clk_gate/CLK (cklan2x1)
                                          0.00     650.00 r
library setup time                      -56.25     593.75
data required time                                 593.75
-----------------------------------------------------------
data required time                                 593.75
data arrival time                                 -603.82
-----------------------------------------------------------
slack (VIOLATED)                                   -10.07
```

report_constraint

This command reports constraints for all active scenarios. Each scenario is reported separately. When used with the `-scenario` option, it reports constraints for a specified list of scenarios.

```
****************************************
Report : constraint
Design : DESIGN1
Scenarios: 0, 1
...
****************************************
                                             Weighted
Group (max_delay/setup)       Cost     Weight     Cost    Scenario
-------------------------------------------------------------------
CLK                           10.07      1.00     10.07    s1
in2out                       372.89      1.00    372.89    s1
in2reg                       199.73      1.00    199.73    s1
reg2out                      467.99      1.00    467.99    s1
reg2reg                      171.16      1.00    171.16    s1
default                        0.00      1.00      0.00    s1
CLK                           90.60      1.00     90.60    s2
in2out                       474.97      1.00    474.97    s2
in2reg                       166.88      1.00    166.88    s2
reg2out                      326.46      1.00    326.46    s2
reg2reg                        0.00      1.00      0.00    s2
default                        0.00      1.00      0.00    s2
-------------------------------------------------------------------
max_delay/setup                                 4404.52

...
                                           Multi-Scenario
Constraint                                    Cost
-----------------------------------------------------
multiport_net                                    0.00 (MET)
min_capacitance                                  0.00 (MET)
max_transition                                  45.28 (VIOLATED)
max_fanout                                     150.00 (VIOLATED)
max_capacitance                                  0.00 (MET)
max_delay/setup                               4404.52 (VIOLATED)
critical_range                                4404.52 (VIOLATED)
min_delay/hold                                   0.00 (MET)
max_area                                    714233.56 (VIOLATED)
```

report_tlu_plus_files

This command reports the TLUPlus files associations; it shows each minimum and maximum TLUPlus and layer map file per scenario:

```
dc_shell-topo> current_scenario s1
Current scenario is: s1

dc_shell-topo> report_tlu_plus_files
Max TLU+ file: /snps/testcase/s1max.tluplus
Min TLU+ file: /snps/testcase/s1min.tluplus
Tech2ITF mapping file: /snps/testcase/tluplus_map.txt
```

report_power

The `report_power` command supports the `-scenario` option. Without the `-scenario` option, only the current scenario is reported. To report power information for all scenarios, use the `report_power -scenarios [all_scenarios]` command.

Note:

In the multicorner-multimode flow, the `report_power` command does not perform clock tree estimation. The command reports only the netlist power in this flow.

The following example shows the report generated by the `report_power -scenario` command.

```
****************************************
Report : power
Design : Design_1
Scenario(s): s1
...
****************************************

Library(s) Used: slow (File: slow.db)

Global Operating Voltage = 1.08
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000pf
    Time Units = 1ns
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = Unitless

Warning: Could not find correlated power. (PWR-725)

Power Breakdown
---------------
                          Cell       Driven Net  Tot Dynamic    Cell
                          Internal   Switching    Power (mW)    Leakage
Cell                      Power (mW)  Power (mW)  (% Cell/Tot)   Power(nW)
------------------------------------------------------------------------
Netlist Power               4.8709     1.2889  6.160e+00 (79%) 1.351e+05
Estimated Clock Tree Power   N/A         N/A        (N/A)        N/A
------------------------------------------------------------------------
```

Supported SDC Commands

Table 10-22 lists the SDC commands supported in the multicorner-multimode flow.

Table 10-22 Supported SDC Commands

Commands	
all_clocks	set_fanout_load
create_clock	set_input_delay
create_generated_clock	set_input_transition
get_clocks	set_latency_adjustment_options
group_path	set_load
set_annotated_delay	set_max_capacitance
set_capacitance	set_max_delay
set_case_analysis	set_dynamic_optimization
set_clock_gating_check	set_leakage_optimization
set_clock_groups	set_max_time_borrow
set_clock_latency	set_max_transition
set_clock_transition	set_min_delay
set_clock_uncertainty	set_multicycle_path
set_data_check	set_output_delay
set_disable_timing	set_propagated_clock
set_drive	set_resistance
set_false_path	set_timing_derate

Multicorner-Multimode Script Example

Example 10-54 shows a basic sample script for the multicorner-multimode flow.

Example 10-54 Basic Script to Run a Multicorner-Multimode Flow

```
#......path settings......
set search_path ". $DESIGN_ROOT $lib_path/dbs \
          $lib_path/mwlibs/macros/LM"
set target_library "stdcell.setup.ftyp.db \
          stdcell.setup.typ.db stdcell.setup.typhv.db"
set link_library [concat * $target_library \
          setup.ftyp.130v.100c.db setup.typhv.130v.100c.db \
          setup.typ.130v.100c.db]
set_min_library stdcell.setup.typ.db -min_version stdcell.hold.typ.db

#......MW setup......
#......load design......

create_scenario s1
set_operating_conditions WORST -library stdcell.setup.typ.db:stdcell_typ
set_tlu_plus_files -max_tluplus design.tlup -tech2itf_map layermap.txt
read_sdc s1.sdc
set_scenario_options -scenarios s1-setup false -hold false \
-leakage_power true

create_scenario s2
set_operating_conditions BEST -library stdcell.setup.ftyp.db:stdcell_ftyp
set_tlu_plus_files -max_tluplus design.tlup -tech2itf_map layermap.txt
read_sdc s2.sdc

create_scenario s3
set_operating_conditions NOM -library stdcell.setup.ftyp.db:stdcell_ftyp
set_tlu_plus_files -max_tluplus design.tlup -tech2itf_map layermap.txt
read_sdc s3.sdc

set_active_scenarios {s1 s2}
report_scenarios
compile_ultra -scan -gate_clock
report_qor
report_constraint
report_timing -scenario [all_scenarios]
.
.
insert_dft
.
.
compile_ultra -incr -scan
```

Using ILMs in Multicorner-Multimode Designs

An interface logic model (ILM) is a structural model of a circuit that is modeled as a smaller circuit representing the interface logic of the block. The model contains cells whose timing is affected by or affects the external environment of a block. ILMs enhance capacity and reduce runtime for the optimization of the top-level design. For ILM details, see Using Hierarchical Models.

ILMs are compatible with multicorner-multimode scenarios. You can apply multicorner-multimode constraints to an ILM and use the ILM in a top-level design.

The following requirements apply to the use of ILMs with multicorner-multimode scenarios:

- For each top-level multicorner-multimode scenario, an identically named scenario must exist in each of the ILM blocks used in the design.

  If there is a mismatch, use the `select_block_scenario` command to map the scenarios in the ILM to the top-level design. Mapping scenarios enables Design Compiler Graphical to support name mismatches between the top-level design and the ILM.

  The syntax for using the `select_block_scenario` command is:

  ```
  select_block_scenario
        [-scenarios top_scenarios]
        [-block_references list_design_names]
        -block_scenario block_scenario_name | -reset
  ```

  To reset user-specified multicorner-multimode scenario mapping, use the `-reset` option.

- An ILM can have additional scenarios that are not used at the top level.
- By default, in a top-level design without multicorner-multimode scenarios, only ILMs without multicorner-multimode scenarios can be used. To use multicorner-multimode ILMs with non multicorner-multimode top-level designs, specify the scenario mapping by using the `select_block_scenario` command.
- For each TLUPlus file that is used, the ILM stores the extraction data and the specified operating condition. In the top-level design, you cannot use additional TLUPlus files or define additional temperature corners for the existing TLUPlus files.

Methodology for Using ILMs With Scenarios at the Top Level

Follow these steps to use an ILM at the top level when it has scenario information:

1. Set the current design to the top-level design.
2. Remove all scenarios.

   ```
   remove_scenario -all
   ```

3. Define scenarios for the top-level design.

   The scenarios defined in the top-level design must match the scenario definitions in the ILM blocks of the design. In the top-level design, all the scenarios must be defined before the next step.

4. Perform optimization.

   ```
   compile_ultra
   ```

   At the beginning of compilation, the `compile_ultra` command performs the following sanity checks to ensure that there are no scenario mismatches between the top-level design and the ILMs. The compilation is terminated when any of the following mismatches are encountered:

   - The number of scenarios in the top-level design match the number of scenarios in the ILM blocks.

     If the tool detects that the top-level design has more scenarios than ILM blocks, an ILM-70 error message is issued, and compilation is terminated.

     ```
     Error: Scenario S6 is not available in ILM Block1. (ILM-70)
     ```

   - The scenario information in the top-level design is consistent with the scenario information in the ILM blocks.

     If scenarios are not defined in the top-level design and the ILM blocks have scenario definitions, an ILM-73 error message is issued and compilation is terminated.

     ```
     Error: Inconsistent use of ILM BlockInit in the multicorner-multimode
     flow. ILM BlockInit has scenarios defined while top design Top does
     not have scenarios defined. (ILM-73)
     ```

You can also use the `check_scenarios` command to check consistency between scenarios.

11

Using a Milkyway Database

The Milkyway database is the unifying design storage format for the Synopsys Galaxy™ Design Platform. The database provides persistent storage of physical design data that links Galaxy platform tools together. The database is periodically updated with new features to support advances in EDA technology.

Design Compiler writes a mapped, uniquified design into the Milkyway database, including the netlist, synthesis constraints, and any physical guidance information, when you use the `write_milkyway` command. You can use a single Milkyway library across the entire Galaxy flow.

Note:
Design Compiler does not support the `read_milkyway` command.

When you use a Milkyway database, you do not need to use an intermediate netlist file exchange format such as Verilog or VHDL to communicate with other Synopsys Galaxy platform tools. Before you can use a Milkyway database within Design Compiler, you must prepare a design library and a reference library and understand the following concepts and tasks:

- About the Milkyway Database
- Guidelines for Using the Milkyway Database
- Preparing to Use the Milkyway Database
- Writing the Milkyway Database

About the Milkyway Database

The Milkyway database stores design data in the Milkyway design library and physical library data in the Milkyway reference library.

- Milkyway design library

  The Milkyway directory structure used to store design data—that is, the uniquified, mapped netlist and constraints—is referred to as the Milkyway design library. You specify the Milkyway design library for the current session by setting the `mw_design_library` variable to the root directory path.

- Milkyway reference library

  The Milkyway directory structure used to store physical library data is referred to as the Milkyway reference library. Reference libraries contain standard cells, macro cells, and pad cells. For information about creating reference libraries, see the Milkyway documentation.

  You specify the Milkyway reference library for the current session by setting the `mw_reference_library` variable to the root directory path. The order in the list implies priority for reference conflict resolution. If more than one reference library has a cell with the same name, the first reference library has precedence.

Required License and Files

Before you use a Milkyway database, you need to have the following required license and files:

- The Milkyway-Interface license

  Design Compiler provides this license, which is used to run the `write_milkyway` command.

- Source for logic libraries (.lib)
- Compiled databases
  - Logic libraries (.db), which contains standard cell timing, power, function, test, and so forth
  - Milkyway library (FRAM), which contains technology data

Invoking the Milkyway Environment Tool

To invoke the Milkyway Environment tool, enter

```
% Milkyway -galaxy
```

The command checks out the Milkyway-Interface license. The Milkyway Environment tool is a graphical user interface (GUI) that enables manipulation of the Milkyway libraries. You can use the tool to maintain your Milkyway design library, such as delete unused versions of your design.

For information about using the Milkyway Environment tool, see the Milkyway documentation.

Guidelines for Using the Milkyway Database

When you use the `write_milkyway` command, observe these guidelines.

- Make sure all the cells present in the Milkyway reference library have corresponding cells in the timing library. The port direction of the cells in the Milkyway reference libraries are set from the port direction of cells in the timing library. If cells are present in the Milkyway reference library but are not in the timing library, the port direction of cells present in the Milkyway reference library is not set.
- Run the `uniquify` command before you run the `write_milkyway` command.
- You must make sure the units in the logic library and the Milkyway technology file are consistent.

  The SDC file does not contain unit information. If the units in the logic library and Milkyway technology file are inconsistent, the `write_milkyway` command cannot convert them automatically. For example, if the logic library uses femtofarad as the capacitance unit and the Milkyway technology file uses picofarad as the capacitance unit, the output of the `write_sdc` command shows different net load values.

  In the following example, the capacitance units in the logic library and the Milkyway technology file are not consistent. The following `set_load` information is shown for the net gpdhi_word_d_21_ before the `write_milkyway` command is run:

  ```
  set_load 1425.15 [get_nets {gpdhi_word_d_21_}]
  ```

  After the `write_milkyway` command is run, the SDC file shows

  ```
  set_load 8.36909 [get_nets {gpdhi_word_d_21_}]
  ```

- Design Compiler is case-sensitive. You can use the tool in case-insensitive mode by doing the following tasks before you run the `write_milkyway` command:
  - Prepare uppercase versions of the libraries used in the link library.
  - Use the `change_names` command to make sure the netlist is uppercased.

Preparing to Use the Milkyway Database

You use the Milkyway design library to specify physical libraries and save designs in the Milkyway format. The inputs required to create a Milkyway design library are the Milkyway reference library and the Milkyway technology file.

To create a Milkyway design library,

1. Create the Milkyway design library by using the `create_mw_lib` command.

   For example,

   ```
   dc_shell-topo> create_mw_lib -technology $mw_tech_file \
    -mw_reference_library $mw_reference_library $mw_design_library_name
   ```

2. Open the Milkyway library that you created by using the `open_mw_lib` command.

   For example,

   ```
   dc_shell-topo> open_mw_lib $mw_design_library_name
   ```

3. (Optional) Attach the TLUPlus files by using the `set_tlu_plus_files` command.

   For example,

   ```
   dc_shell-topo> set_tlu_plus_files -max_tluplus $max_tlu_file \
      -min_tluplus $min_tlu_file -tech2itf_map $prs_map_file
   ```

4. In subsequent sessions, use the `open_mw_lib` command to open the Milkyway library. If you are using the TLUPlus files for RC estimation, use the `set_tlu_plus_files` command to attach these files.

   For example,

   ```
   dc_shell-topo> open_mw_lib $mw_design_library_name
   dc_shell-topo> set_tlu_plus_files -max_tluplus $max_tlu_file \
      -min_tluplus $min_tlu_file -tech2itf_map $prs_map_file
   ```

For more information about using TLUPlus files to provide more accurate capacitance and resistance data, see Using TLUPlus Files for RC Estimation.

Writing the Milkyway Database

To save the design data in a Milkyway design library, use the `write_milkyway` command. The `write_milkyway` command writes netlist and physical data from memory to the Milkyway design library format, re-creates the hierarchy preservation information, and saves the design data for the current design in a design file. The path for the design file is *design_dir*/CEL/*file_name*:*version*, where *design_dir* is the location you specified in the `mw_design_library` variable.

For example, you use the `-output` option to specify the file name.

```
dc_shell-topo> write_milkyway -output my_file -overwrite
```

To write design information from memory to a Milkyway library named testmw and name the design file TOP, enter

```
dc_shell-topo> set_app_var mw_design_library testmw
dc_shell-topo> write_milkyway -output TOP
```

Example 11-1 shows a script to set up and write a Milkyway database.

Example 11-1 Script to Set Up and Write a Milkyway Database

```
set_app_var search_path "$search_path ./libraries"
set_app_var link_library "* max_lib.db"
set_app_var target_library "max_lib.db"

create_mw_lib -technology $mw_tech_file -mw_reference_library \
                $mw_reference_library $mw_lib_name
open_mw_lib $mw_lib_name
read_file -format ddc design.ddc
current_design TopDesign
link
write_milkyway -output myTop
```

Important Points About the write_milkyway Command

When you use the `write_milkyway` command, keep the following points in mind:

- You must run the `create_mw_lib` command before running the `write_milkyway` command.
- If a design file already exists (that is, you ran the `write_milkyway` command on the design with the same output directory), the `write_milkyway` command creates an additional design file and increments the version number. You must ensure that you open the correct version in Milkyway; by default, Milkyway opens the latest version. To avoid creating an additional version, specify the `-overwrite` option to overwrite the current version of the design file and save disk space.
- The command does not modify in-memory data.
- Attributes present in the design in memory that have equivalent attributes in Milkyway are translated (not all attributes present in the design database are translated).
- A hierarchical netlist translated by using the `write_milkyway` command retains its hierarchy in the Milkyway database.

Limitations When Writing Milkyway Format

The following limitations apply when you write your design in Milkyway format:

- The design must be mapped.

  Because the Milkyway format describes physical information, it supports mapped designs only. You cannot use the Milkyway format to store design data for unmapped designs.

- The design must not contain multiple instances.

  You must uniquify your design before saving it in Milkyway format. Use the `check_design -multiple_designs` command to report information related to multiply-instantiated designs.

- The `write_milkyway` command saves the entire hierarchical design in a single Milkyway design file. You cannot generate separate design files for each subdesign.
- When you save a design in Milkyway format, the `write_milkyway` command does not save the interface logic model (ILM) instances in the Milkyway design library. Design Compiler in topographical mode cannot save ILMs using the `write_milkyway` command. You must explicitly save each ILM in .ddc format. For more information about ILM usage, see Using Hierarchical Models.

12

Analyzing and Resolving Design Problems

Use the reports generated by Design Compiler to analyze and debug your design. You can generate reports both before and after you compile your design. Generate reports before compiling to check that you have set attributes, constraints, and design rules properly. Generate reports after compiling to analyze the results and debug your design.

To learn about analyzing and resolving design problems, see

Instantiating RTL PG Pins in Non-UPF Designs

Design Compiler can accept RTL designs containing a small number of power/ground pin connections on macros in non-UPF designs. The tool does not support a full PG netlist for a block. For example, the tool only supports designs that contain a small number of analog macros that have PG pins.

To instantiate PG pins in your RTL design, set the `dc_allow_rtl_pg` variable to `true`. The default is `false`. To preserve the PG connections in a Verilog output, execute the `write_file -pg -format verilog` command. To preserve the PG connections in a .ddc format output, execute the `write_file -format ddc` command. Note that when saving the design in .ddc format, you do not need to use the `-pg` option. When reading the .ddc file back into Design Compiler, make sure that the `dc_allow_rtl_pg` variable is set to `true`; otherwise, the tool issues a DDC-21 error:

```
Error: The feature used to generate this DDC file is not supported by
this tool or is not enabled in the current session. (DDC-21)
Information: This .ddc file contains RTL PG data. Set the dc_allow_rtl_pg
variable to true before reading the file back into Design Compiler.
```

To preserve the PG connections in a Milkyway database, use the `write_milkyway` command. To pass the design netlist to IC Compiler, PrimeTime, or Formality, you can use a Verilog output, .ddc file, or Milkyway database.

To use PG pins in your RTL design, observe the following guidelines:

- PG libraries are required
- FRAM must always have correct PG information
- The tool will not display the PG nets and pins; the `get_pins` command will not show PG pins
- The RTL design must represent all PG pins as wires, not as supply0, supply1, and so on
- The RTL design must instance PG pins by name, such as ref U1 (.pin(net), …);
- PG nets can only connect to the following:
  - Macro cells
  - PG pins on leaf power management cells (power switches, level shifters, and isolation cells)
  - Hierarchical ports
- PG nets should reach the top level, but do not have to connect to top-level ports
- The tool will mark cells with PG pins with the `dont_touch` attribute

- If any UPF commands are executed, the derived PG network is converted to UPF and the UPF flow is followed

Example 12-1 shows Verilog RTL code that instantiates two PG pins: my_vdd and my_vss.

Example 12-1 Coding PG Pins in the RTL Design

```
module  my_design(a, b, c, my_vdd, my_vss);
input a, b, my_vdd, my_vss;
output c;
   my_macro  U1(.a(a), .b(b), .c(c), .VDD(my_vdd), .VSS(my_vss));
endmodule
```

Resolving Bus Versus Bit-Blasted Mismatches Between the RTL and Macros

Typically, the RTL pin names and the logical library pin names match. Signals are defined one way in both the RTL and the library. They are defined as buses or the bus is defined by its individual wires. Occasionally, mismatches occur during development when, for example, you port from one technology to another, or when the libraries and the RTL are in flux. When mismatches occur, you can set the `enable_bit_blasted_bus_linking` variable to `true` and read your design into Design Compiler, even though your RTL and libraries do not match with respect to bus verses bit-blasted pin names. When the `enable_bit_blasted_bus_linking` variable is set to `true`, the linker rules are relaxed such that you can read your design when mismatched pin names occur. The default for the `enable_bit_blasted_bus_linking` variable is `false`.

Note that when the `enable_bit_blasted_bus_linking` variable is set to `true`, the tool will match names in accordance with the `bus_inference_style` and `bus_inference_descending_sort` variable settings.

Fixing Errors Caused by New Unsupported Technology File Attributes

Occasionally, IC Compiler adds support for new technology file attributes that are not yet supported in Design Compiler. In these cases, a TFCHK-009 error message is issued in Design Compiler but not in IC Compiler when using the same technology file.

If you get TFCHK-009 errors when reading in your technology file, check the spelling of the attributes and make sure that the attributes are spelled correctly. If you still have TFCHK-009 errors, you can use either of the following two methods to remove the TFCHK-009 errors. Before using either of the following methods, make sure that you can safely remove or ignore the new attributes without affecting the tool functionality. In most cases, new attributes are associated with new routing rules and should not affect the functionality of Design Compiler.

- Remove the new attributes from the specified section in the technology file.
- If the new attributes are safe to ignore, you can set the `ignore_tf_error` variable to `true` which enables the tool to ignore the unsupported attributes. This allows the tool to ignore all TFCHK-009 errors, but the errors will still be issued in the log file.

Using Register Replication to Solve Timing QoR, Congestion, and Fanout Problems

Design Compiler can replicate registers to address timing quality of results (QoR), congestion, and fanout issues. This feature is supported in both Design Compiler topographical mode and wire-load mode. To enable register replication, use the `set_register_replication` command.

For Design Compiler topographical, register replication is placement-aware, which can help reduce congestion in some cases. For wire-load mode, the load of the original replicated register is evenly distributed among the new replicated registers. For details, see the *Design Compiler Optimization Reference Manual*.

Comparing Design Compiler Topographical and IC Compiler Environments

Sometimes Design Compiler topographical and IC Compiler have different environment settings. These differences can lead to correlation problems. To help fix correlation issues between Design Compiler topographical and IC Compiler, use the `consistency_checker` command in your UNIX shell to compare the respective environments. However, before you can compare the two environments, you need to determine the environments of each tool. To do this, use the `write_environment` command as shown in Example 12-2 and Example 12-3. Use the `-output` option to specify the name of your output report. In Design Compiler, execute `write_environment` before `compile_ultra`; in IC Compiler, execute `write_environment` before `place_opt`.

Example 12-2 Using write_environment in Design Compiler

```
dc_shell-topo> write_environment -consistency -output DCT.env
```

Example 12-3 Using write_environment in IC Compiler

```
icc_shell> write_environment -consistency -output ICC.env
```

After you have the environments for each tool, compare the environments using the `consistency_checker` command, as shown in Example 12-4.

Example 12-4 Comparing Design Compiler and IC Compiler Environments

```
unix_shell> consistency_checker -file1 DCT.env -file2 ICC.env \
                                -folder temp \
                                -html html_report | tee cc.log
```

The resulting HTML output report lists mismatched commands and variables. In Example 12-4, the HTML output report is written to the ./html_report directory. To access this report, open the ./html_report/index.html file in any Web browser. To reduce correlation problems, fix these mismatches before proceeding to optimization.

In addition to the HTML output report, the tool prints a summary report to the shell while the `consistency_checker` command is running. You can use the UNIX tee command, as shown in Example 12-4, to save this report to a file. In Example 12-4, the report is saved to the ./cc.log file.

Note:

The ./temp directory stores intermediate files the tool uses when executing the `consistency_checker` command. After the command completes, you can remove the ./temp directory.

For more details, see SolvNet article 026366.

Assessing Design and Constraint Feasibility in Mapped Designs

To help debug missing timing constraints and assess design and constraint feasibility in mapped designs, use the `set_zero_interconnect_delay_mode` command, as shown in Example 12-5. In this example, you set the `set_zero_interconnect_delay_mode` command to `true`, run `report_qor`, and set the `set_zero_interconnect_delay_mode` command back to its default of `false` before proceeding with further optimization commands. When `set_zero_interconnect_delay_mode` is set to `true`, the tool analyzes your design with only cell delays and the capacitance of the pin-load on all the wires in the design to determine if your design meets timing goals. The tool does not consider wire capacitance due to timing paths.

Always set `set_zero_interconnect_delay_mode` back to its default of `false` before running your optimization step. The tool reports warning messages if you use optimization commands when `set_zero_interconnect_delay_mode` is `true`.

Example 12-5 set_zero_interconnect_delay_mode Sample Script

```
...
compile
...
set_zero_interconnect_delay_mode true
report_qor
set_zero_interconnect_delay_mode false
```

```
...
```

When `set_zero_interconnect_delay_mode` is set to `true`, the tool reports the following warning when you execute `report_constraint` or `report_qor`:

```
Warning: Timer is in zero interconnect delay mode. (TIM-177)
```

Checking for Design Consistency

A design is consistent when it does not contain errors, such as unconnected ports, constant-valued ports, cells with no input or output pins, mismatches between a cell and its reference, multiple driver nets, connection class violations, or recursive hierarchy definitions.

Design Compiler runs the `check_design -summary` command on all designs that are compiled. You can also run the command explicitly to verify design consistency. The command reports a list of warning and error messages as follows:

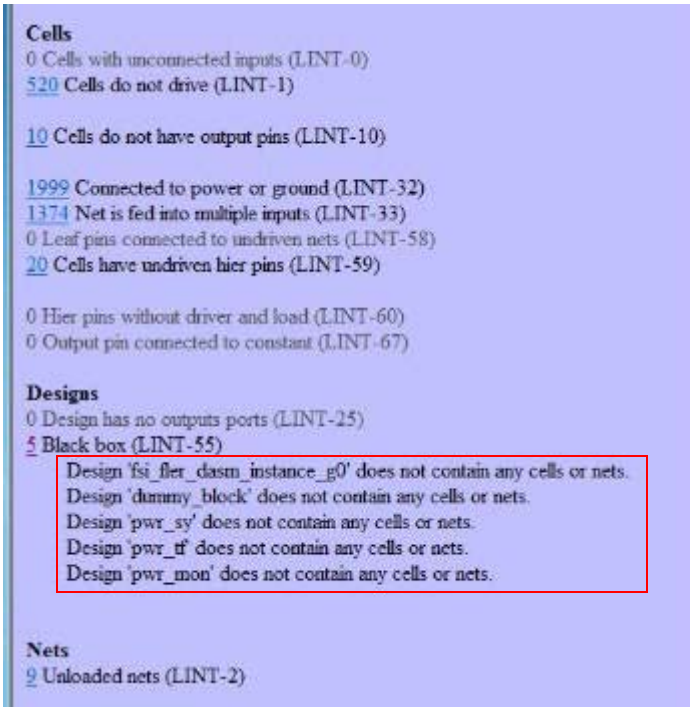
- It reports an error if it finds a problem that Design Compiler cannot resolve. For example, recursive hierarchy (when a design references itself) is an error. You cannot compile a design that has `check_design` errors.
- It reports a warning if it finds a problem that indicates a corrupted design or a design mistake not severe enough to cause the `compile` command to fail.

The report is displayed in standard output format by default. The report begins with a summary section that shows the type of check and how many violations have been detected. Next, it lists the error and warning messages.

You can specify a report in HTML format by using the `-html_file_name` option with the `check_design` command. When you specify this option and the file name, Design Compiler creates an HTML file in the current directory.

Figure 12-1 shows an example of a `check_design` report in HTML format. The report organizes the information into sections, such as Input/Outputs, Cells, Designs, Nets, and so on. Each section lists the type of check and the number of violations. If there are violations, the number is an HTML link that you can click to display more information. The report in Figure 12-1 shows five black box violations. Clicking the number 5 expands the report to show the five highlighted violations.

Figure 12-1 HTML check_design Report Example



By default, during compilation or execution of the `check_design` command, Design Compiler issues a warning message if a tristate bus is driven by a non-tristate driver. You can have Design Compiler display an error message instead by setting the `check_design_allow_non_tri_drivers_on_tri_bus` variable to `false`. The default is `true`. When the variable is set to `false`, the `compile` command stops after reporting the error; however, the `check_design` command continues to run after the error is reported.

You can use options with the `check_design` command to filter the messages based on the type of check, such as warning messages related to ports, nets, and cells, information messages related to multiply-instantiated designs, and so on. For example, if you specify the `-multiple_designs` option, the report displays a list of multiply-instantiated designs along with instance names and associated attributes, such as `dont_touch`, `black_box`, and `ungroup`. By default, messages related to multiply-instantiated designs are suppressed.

Use the `-no_connection_class` option when you are working on a GTECH design or netlist in which connection class violations are expected. Doing so improves runtime, especially if the GTECH design is large and has several connection class violations.

Checking Designs and Libraries Before Synthesis

Before synthesizing your design, check that all designs and libraries have the necessary data for the compilation to run successfully by using the `compile_ultra` command with the `-check_only` option. The `-check_only` option reports potential problems that could cause the tool to stop during the `compile_ultra` run or to produce unsatisfactory correlation with your physical implementation. Use the `-check_only` option to help debug such problems as

- Missing TLUPlus files and missing physical library cells
- Multiple logic library cells with the same names
- Discrepancies in technology data between multiple physical libraries
- Missing placement location for cells and ports of physical hierarchical modules or ILMs
- Missing `dont_touch` attributes for physical hierarchical modules or ILMs
- Missing floorplan information such as core area constraints (through site row definition), port location, macro location, physical hierarchy location, and ILM location.

Important:

Always execute the `compile_ultra` optimization step with the same set of options that were used when you executed the `compile_ultra -check_only` command.

Analyzing Your Design During Optimization

To learn about capabilities for analyzing your design during optimization, see

- Customizing the Compile Log
- Saving Intermediate Design Databases

Customizing the Compile Log

The compile log records the status of the compile run. Each optimization task has an introductory heading, followed by the actions taken while that task is performed. There are three tasks in which Design Compiler works to reduce the compile cost function:

- Delay optimization
- Design rule fixing
- Area optimization

While completing these tasks, Design Compiler performs many trials to determine how to reduce the cost function. For this reason, these tasks are collectively known as the trials phase of optimization.

By default, Design Compiler logs each action in the trials phase by providing the following information:

- Elapsed time
- Design area
- Worst negative slack
- Total negative slack
- Design rule cost
- Endpoint being worked on

You can customize the trials phase output by setting the `compile_log_format` variable. Table 12-1 lists the available data items and the keywords used to select them.

For information about generating a log file in HTML format, see Compile Log Files.

Table 12-1 Compile Log Format Keywords

Column	Column header	Keyword	Column description
Area	AREA	area	Shows the area of the design.
CPU seconds	CPU SEC	cpu	Shows the process CPU time used (in seconds).
Design rule cost	DESIGN RULE COST	drc	Measures the difference between the actual results and user-specified design rule constraints.
Elapsed time	ELAPSED TIME	elap_time	Tracks the elapsed time since the beginning of the current compile or reoptimization of the design.
Endpoint	ENDPOINT	endpoint	Shows the endpoint being worked on. When delay violations are being fixed, the endpoint is a cell or a port. When design rule violations are being fixed, the endpoint is a net. When area violations are being fixed, no endpoint is printed.
Maximum delay cost	MAX DELAY COST	max_delay	Shows the maximum delay cost of the design.
Megabytes of memory	MBYTES	mem	Shows the process memory used (in MB).
Minimum delay cost	MIN DELAY COST	min_delay	Shows the minimum delay cost of the design.
Path group	PATH GROUP	group_path	Shows the path group of an endpoint.
Time of day	TIME OF DAY	time	Shows the current time.

Table 12-1 Compile Log Format Keywords (Continued)

Column	Column header	Keyword	Column description
Total negative slack	TOTAL NEG SLACK	tns	Shows the total negative slack of the design.
Trials	TRIALS	trials	Tracks the number of transformations that the optimizer tried before making the current selection.
Worst negative slack	WORST NEG SLACK	wns	Shows the worst negative slack of the current path group.

Saving Intermediate Design Databases

Design Compiler provides the capability to output an intermediate design database during the trials phase of the optimization process. This capability is called checkpointing. Checkpointing saves the entire hierarchy of the intermediate design. You can use this intermediate design to debug design problems, as described in Analyzing Design Problems. To checkpoint automatically between each phase of compile, set the `compile_checkpoint_phases` variable to true.

Analyzing Design Problems

Table 12-2 shows the design analysis commands provided by Design Compiler.

Table 12-2 Commands to Analyze Design Objects

Object	Command	Description

Design	`report_design`
	`report_area`
	`report_hierarchy`
	`report_resources`
		Reports design characteristics.
		Reports design size and object counts.
		Reports design hierarchy.
		Reports resource implementations.

Instances	`report_cell`	Displays information about instances.
References	`report_reference`	Displays information about references.
Pins	`report_transitive_fanin` `report_transitive_fanout`	Reports fanin logic. Reports fanout logic.

Table 12-2 Commands to Analyze Design Objects (Continued)

Object	Command	Description
Ports	`report_port`	Displays information about ports.
	`report_bus`	Displays information about bused ports.
	`report_transitive_fanin`	Reports fanin logic.
	`report_transitive_fanout`	Reports fanout logic.
Nets	`report_net`	Reports net characteristics.
	`report_bus`	Reports bused net characteristics.
	`report_transitive_fanin`	Reports fanin logic.
	`report_transitive_fanout`	Reports fanout logic.
Clocks	`report_clock`	Displays information about clocks.

Analyzing Area

Use the `report_area` command to display area information and statistics for the current design or instance. The command reports combinational, non-combinational, and total area. If you have set the current instance, the report is generated for the design of that instance; otherwise, the report is generated for the current design. Use the `-hierarchy` option to report area used by cells across the hierarchy.

Example 12-6 shows a report generated by the `report_area` command:

Example 12-6 report_area Report Example

```
dc_shell > report_area

****************************************
Report : area
Design : TEST_TOP
Version: E-2010.12
Date   : Sun Oct 24 02:08:37 2010
****************************************

Library(s) Used:
test_lib (File: test_lib.db)

Number of ports:                          565
Number of nets:                          9654
Number of cells:                         8120
Number of combinational cells:           6594
Number of sequential cells:              1526
Number of macros:                           0
Number of buf/inv:                       1439
Number of references:                     163
```

```
Combinational area:       562404.432061
Noncombinational area:    6720840.351067
Net Interconnect area:      undefined  (Wire load has zero net area)
Total cell area:          7283244.783128
Total area:                 undefined
```

Analyzing Timing

Use the `report_timing` command to generate timing reports for the current design or the current instance. By default, the command lists the full path of the longest maximum delay timing path for each path group. Design Compiler groups paths based on the clock controlling the endpoint. All paths not associated with a clock are in the default path group. You can also create path groups by using the `group_path` command.

Before you begin debugging timing problems, verify that your design meets the following requirements:

- You have defined the operating conditions.
- You have specified realistic constraints.
- You have appropriately budgeted the timing constraints.
- You have properly constrained the paths.
- You have described the clock skew.

If your design does not meet these requirements, make sure it does before you proceed.

After producing the initial mapped netlist, use the `report_constraint` command to check your design's performance.

Table 12-3 lists the timing analysis commands.

Table 12-3 Timing Analysis Commands

Command	Analysis task description
`report_design`	Shows operating conditions, wire load model and mode, timing ranges, internal input and output, and disabled timing arcs.
`check_timing`	Checks for unconstrained timing paths and clock-gating logic.
`report_port`	Shows unconstrained input and output ports and port loading.
`report_timing_requirements`	Shows all timing exceptions set on the design.
`report_clock`	Checks the clock definition and clock skew information.

Table 12-3 Timing Analysis Commands (Continued)

Command	Analysis task description
`report_path_group`	Shows all timing path groups in the design.
`report_timing`	Checks the timing of the design.
`report_constraint`	Checks the design constraints.
`report_delay_calculation`	Reports the details of a delay arc calculation.

Reporting Quality of Results

You can generate a report on the quality of results (QoR) for the design in its current state by using reporting commands, such as `create_qor_snapshot`, `query_qor_snapshot`, and `report_qor`. The `create_qor_snapshot` command measures and reports the quality of the design in terms of timing, design rules, area, power, congestion, clock tree synthesis, routing, and so on. It stores the quality information in a set of snapshot files. You can later retrieve the snapshot with the `query_qor_snapshot` command and view the information in a categorized timing report. You can also selectively retrieve, sort, and display the information based on your preferences. The `report_qor` command displays QoR information and statistics for the current design.

For information about generating and viewing QoR reports, see

- Measuring Quality of Results
- Analyzing Quality of Results
- Displaying Quality of Results

Measuring Quality of Results

Note:

The `create_qor_snapshot` command is available only in Design Compiler in topographical mode.

To measure the quality of results of the design in its current state and store the quality information in a set of report files, use the `create_qor_snapshot` command. You can capture the QoR information when using different optimization strategies or at different stages of the design and compare the quality of results. For example, you can use the `create_qor_snapshot` command to create a snapshot after the first time you compile the design, after DFT insertion, and again after an incremental compilation.

The command options let you specify the conditions for analysis, such as zero wire load, maximum paths per timing group, and maximum paths per endpoint.

When you use the `create_qor_snapshot` command to create a snapshot, you must at least specify a name for the snapshot by using the `-name` option. For example,

```
dc_shell-topo> create_qor_snapshot -name snapshot1
```

The report is written to a set of files in a directory called “snapshot” in the current working directory. The set of files act as a database for the snapshot report. You do not need to access these files, and you must not modify them. Later, when you run the `query_qor_snapshot` command and specify the `-name` option with the same name you used to create the snapshot (snapshot1 in this example), the `query_qor_snapshot` command displays the results in text or HTML format.

Analyzing Quality of Results

Note:

The `query_qor_snapshot` command is available only in Design Compiler in topographical mode.

To read a QoR snapshot previously generated by the `create_qor_snapshot` command, analyze the results, and display the information in a categorized timing report, use the `query_qor_snapshot` command. The `query_qor_snapshot` command does not perform any additional analysis of the design but merely processes the report information already saved in the snapshot files.

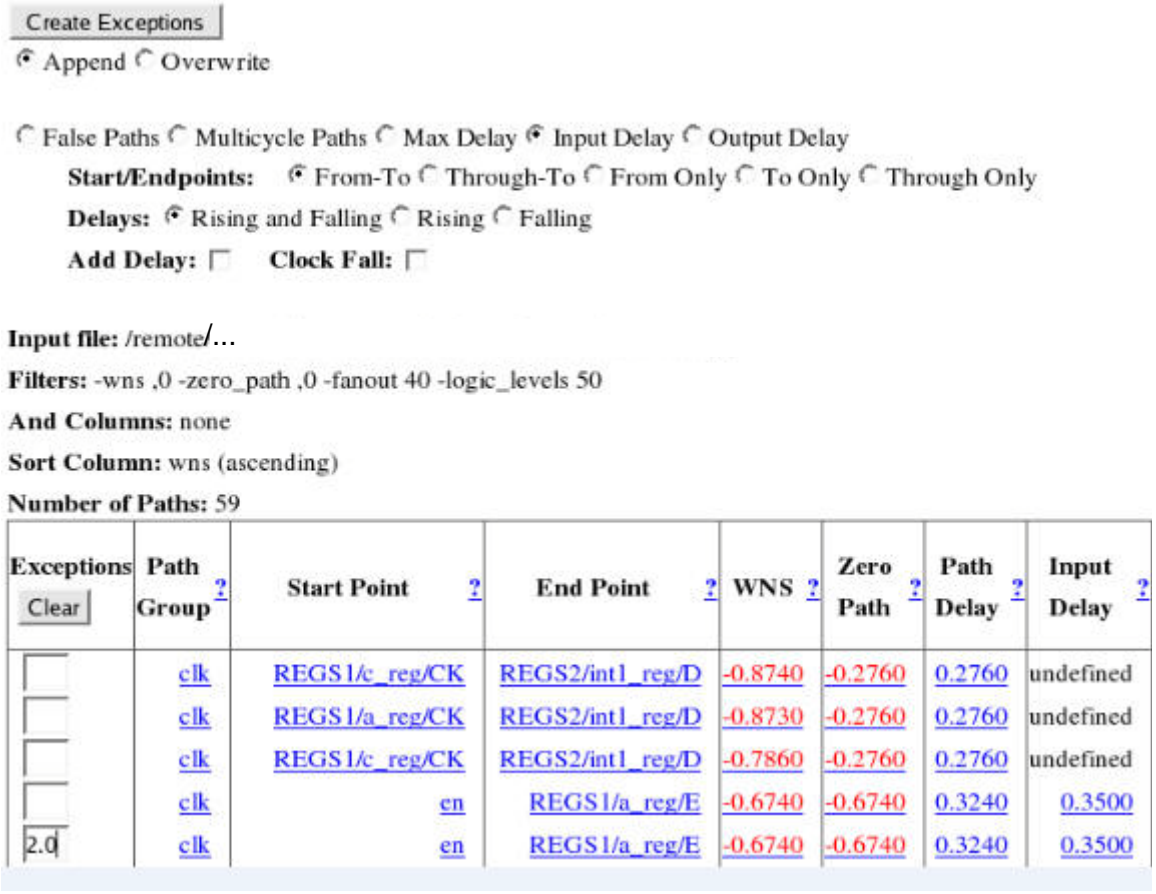
The `query_qor_snapshot` command can display the categorized timing report in HTML format or in text format. If you open the report in HTML format, you can modify the constraints and generate a new categorized timing report. The text version of the report allows you to view the results but does not allow you to make modifications.

Use the following command to specify a snapshot and specify the name of the generated output file. You can include an .html or .txt file extension to specify the file format. If you do not specify a file format, Design Compiler creates both a text and an HTML output file.

```
dc_shell-topo> query_qor_snapshot -name snapshot1 -output_file file_name
```

Figure 12-2 shows a categorized timing report in HTML format.

Figure 12-2 Categorized Timing Report in HTML Format



Create Exceptions

Append Overwrite

False Paths Multicycle Paths Max Delay Input Delay Output Delay

Start/Endpoints: From-To Through-To From Only To Only Through Only

Delays: Rising and Falling Rising Falling

Add Delay: **Clock Fall:**

Input file: /remote/...

Filters: -wns ,0 -zero_path ,0 -fanout 40 -logic_levels 50

And Columns: none

Sort Column: wns (ascending)

Number of Paths: 59

Exceptions Clear	Path Group ?	Start Point ?	End Point ?	WNS ?	Zero Path ?	Path Delay ?	Input Delay ?
	clk	REGS1/c_reg/CK	REGS2/int1_reg/D	-0.8740	-0.2760	0.2760	undefined
	clk	REGS1/a_reg/CK	REGS2/int1_reg/D	-0.8730	-0.2760	0.2760	undefined
	clk	REGS1/c_reg/CK	REGS2/int1_reg/D	-0.7860	-0.2760	0.2760	undefined
	clk	en	REGS1/a_reg/E	-0.6740	-0.6740	0.3240	0.3500
2.0	clk	en	REGS1/a_reg/E	-0.6740	-0.6740	0.3240	0.3500

The HTML report lets you quickly find paths with certain problems, such as large fanouts or transition degradation. You can then modify the constraints and generate a new HTML report based on the constraints you specified. To change the constraints and generate a new HTML report, perform the following steps:

1. Select a parameter that you want to change.

   For example, in Figure 12-2, Input Delay is selected.

2. Enter a new value for a specific path or paths.

   In Figure 12-2, a new input delay value of 2.0 is set on the enable input pin of the REGS1/a_reg/E register.

3. Click the Create Exceptions button.

   Design Compiler opens a new browser window that contains a set of commands based on the constraints you specified. Update your SDC constraints with these commands, and then rerun Design Compiler.

The `query_qor_snapshot` command options let you filter and sort the search results. You can specify which data columns to display. For example, you can generate `set_false_path` constraints for a specified value range, such as WNS, logic level, and so on. (The `set_false_path` command removes timing constraints from specified paths that you know do not affect circuit operation.)

You apply filters to the paths by using the `-filters` option. For example, the following command reports all paths that have a worst negative slack less than zero:

```
dc_shell-topo> query_qor_snapshot -name my_snapshot -filters "-wns ,0"
```

The following example reports all paths that meet any of the specified filter requirements:

```
dc_shell-topo> query_qor_snapshot -name my_snapshot \
   -filters "-wns -4.0 -fanout 20"
```

If you do not specify the `-filters` option, the `query_qor_snapshot` command automatically applies a set of default filters. The default filter settings for the maximum condition are shown in the following example:

```
dc_shell-topo> query_qor_snapshot -name my_snapshot \
   -filters "-wns ,0 -zero_path ,0 -fanout 40 -logic_level 50"
```

For a complete list of filters, see the `query_qor_snapshot` man page.

The following additional examples demonstrate the `query_qor_snapshot` command usage.

Example 12-7 analyzes the snapshot named my_snapshot1 and reports the paths having a worst negative slack between –4.0 and –3.0 time units.

Example 12-7 Reporting Paths With Worst Negative Slack

```
dc_shell-topo> query_qor_snapshot -name my_snapshot1 -filters "-wns
-4.0,-3.0"
...
  Path Group       Start Point       End Point       WNS
  ...              ...               ...          -3.312
  ...              ...               ...          -3.277
```

Example 12-8 reports the paths having a slack worse than –1.0 time units and a fanout greater than 40.

Example 12-8 Reporting Paths for Slack and Fanout

```
dc_shell-topo> query_qor_snapshot -name my_snapshot2 \
    -filters "-wns ,-1.0 -fanout 40"
...
```

```
  Path Group       Start Point       End Point        WNS   Large Fanout
  ...              ...               ...           -3.312   42
  ...              ...               ...           -3.277   42
```

Example 12-9 reads the snapshot named max_logic_level and specifies the `-filters` option to report all paths between logic levels 2 and 14.

Example 12-9 Reporting All Paths Between Specified Logic Levels

```
dc_shell-topo> query_qor_snapshot -name max_logic_level \
   -filters "-logic_level 2,15"
```

Note that the paths of logic level 2 are included in the report, but the paths of logic level 15 are excluded.

Sometimes the paths with the worst timing start in one hierarchical block and end in another. Identifying and grouping these paths can be a challenging task. The `-hierarchy` option enables the top-level view and automates the process of finding potential path groupings that cross hierarchical boundaries so that the problem paths can be analyzed more efficiently. When you use the `-hierarchy` option by itself, as shown in Example 12-10, Design Compiler finds all the timing violations that cross ILMs and hard macros in the design.

Example 12-10 Reporting Timing Violations Across Hierarchical Boundaries

```
dc_shell-topo> query_qor_snapshot -name snap_shot3 -hierarchy
```

The `query_qor_snapshot` command automatically generates a report of all violating paths between the extracted timing models (ETMs), ILMs, and hard macros. You can apply filters or expand the top-level paths by specifying the `-to`, `-from`, and `-through` options. Example 12-11 finds violating paths that pass through modules A, B, or C:

Example 12-11 Reporting Violating Paths That Pass Through Specified Modules

```
dc_shell-topo> query_qor_snapshot -name snap_shot4 -hierarchy \
     -through {A B C}
```

Example 12-12 uses the `-incremental` option, which keeps the previous analysis results (through modules A, B, and C), and applies an additional query about paths starting from module A or B and ending at module A or B.

Example 12-12 Incremental Reporting

```
dc_shell-topo> query_qor_snapshot -hierarchy -incremental \
    -from {A/* B/*} -to {A/* B/*}
```

Example 12-13 reports paths that start from module C and end at module B, having a worst negative slack worse than –3.0 ns or a fanout more than 40.

Example 12-13 Reporting Paths for WNS and Fanout Passing Through Specified Modules

```
dc_shell-topo> query_qor_snapshot -name my_snapshot5 \
   -hierarchy -from top/A/C/* -to top/B/* \
   -filters "-wns ,-3.0 -fanout 40"
```

Displaying Quality of Results

To display information about the quality of results and other statistics for the current design, use the `report_qor` command. The command reports information about timing path group details and cell count and current design statistics, including combinational, noncombinational, and total area. The command also reports static power, design rule violations, and compile time details. Note that the `report_qor` command is not part of the `create_qor_snapshot` and `query_qor_snapshot` command set.

Example 12-14 shows a report generated by the `report_qor` command. In the `Overall Compile Wall Clock Time` section, the reported time is the combined wall clock time, combining the `compile_ultra` and `compile_ultra -incremental` command runs. In the `Cell Count` section, the report shows the number of macros in the design. To qualify as a macro cell, a cell must be nonhierarchical and have the `is_macro_cell` attribute set on its library cell.

Example 12-14 report_qor Report Example

```
dc_shell > report_qor

****************************************
Report : qor
Design : TEST_TOP
...
****************************************

  Timing Path Group 'clk'
  -----------------------------------
  Levels of Logic:               1.00
  Critical Path Length:          0.02
  Critical Path Slack:           -0.68
  Critical Path Clk Period:     8.00
  Total Negative Slack:          -393.61
  No. of Violating Paths:        1066.00
  Worst Hold Violation:          0.00
  Total Hold Violation:          0.00
  No. of Hold Violations:        0.00
  -----------------------------------


 Cell Count
  -----------------------------------
  Hierarchical Cell Count:          7
  Hierarchical Port Count:       2360
```

```
Leaf Cell Count:               43978
Buf/Inv Cell Count:             6764
CT Buf/Inv Cell Count:             4
Combinational Cell Count:      37212
Sequential Cell Count:          6773
Macro Count:                       0
-----------------------------------

Area
-----------------------------------
Combinational Area:   562404.432061
Noncombinational Area:
                     6720840.351067
Net Area:                  0.000000
Net XLength        :     2836623.25
Net YLength        :     2555007.75
-----------------------------------
Cell Area:           7283244.783128
Design Area:         7283244.783128
Net Length         :      5391631.00

Design Rules
-----------------------------------
Total Number of Nets:          47207
Nets With Violations:              5
Max Trans Violations:              3
-----------------------------------

Hostname: machine
Compile CPU Statistics
---------------------------------------
Resource Sharing:                  21.54
Logic Optimization:               182.63
Mapping Optimization:             230.79
---------------------------------------
Overall Compile Time:             631.32
Overall Compile Wall Clock Time:  288.11
```

Debugging Cells and Nets with dont_touch

If Design Compiler did not optimize or remove a cell or net, it is often due to a dont_touch on the object. Explicit `dont_touch` attributes are straightforward to debug. However, it can be difficult to determine why an object has an implicit dont_touch, especially when there are multiple overlapping types of implicit dont_touch on the object. Even if you remove the dont_touch setting on the object, you might still see the object reported as having a dont_touch. This happens in cases where the object has a dont_touch for multiple reasons.

For information about debugging why an object in a cell or net is marked as dont_touch, see

- Reporting dont_touch Cells and Nets
- Creating a Collection of dont_touch Cells and Nets

Reporting dont_touch Cells and Nets

To report all the dont_touch cells and nets in the design and the types of dont_touch that apply to each reported object, use the `report_dont_touch` command. The dont_touch objects are reported for the complete design hierarchy.

Objects that are not dont_touch cells or nets are skipped. You can also specify a collection of cells or nets with the `report_dont_touch` command.

The following example shows a report of the dont_touch types for a specific net. In this example, the `report_dont_touch` command is used to investigate the types of dont_touch on the Multiplier/Product[0] net. The net is listed with all the types of dont_touch that are on the net. The table legend provides a detailed description for each type of dont_touch that is reported. There are many possible dont_touch types, but only the relevant types are listed.

```
dc_shell> report_dont_touch [get_nets Multiplier/Product[0]]
...
Description for Net dont_touch Types:
  inh_parent    -  Net inherits dont_touch from parent
  mv_iso        -  Prevents buffering between isolation cells and
                   power domain boundary


Object                          Class       Types
-------------------------------------------------------------------
Multiplier/Product[0]           net         inh_parent, mv_iso
-------------------------------------------------------------------
```

As the example shows, the Multiplier/Product[0] net inherited one dont_touch from a parent cell. This net is inside a hierarchical block on which an explicit dont_touch was set on the parent hierarchical cell. The net also has an implicit dont_touch on it as a result of multivoltage synthesis.

The following example shows a dont_touch report for all the cells in the design:

```
dc_shell> report_dont_touch -class cell -nosplit
...
Description for Cell dont_touch Types:
  inh_ref          -  Cell inherits dont_touch from reference design or library cell
  ph_fixed_placement - Cell with fixed placement


Object                                                  Class       Types
--------------------------------------------------------------------------------
headerfooter5_HDRDID1BWPHVT_R5_C0                       cell        ph_fixed_placement
headerfooter6_HDRDID1BWPHVT_R6_C0                       cell        ph_fixed_placement
u_des_soft_macro/u_8/u_mem                              cell        inh_ref
--------------------------------------------------------------------------------
Total 3 dont_touch cells
1
```

To return an alphabetically sorted list of currently defined cell and net dont_touch types and their descriptions, use the `list_dont_touch_types` command. This command is useful when you want to look up the types available for building a collection of dont_touch objects. By default, the `list_dont_touch_types` command lists all dont_touch types for cell and net object classes.

Use the `-class` *`class_name`* option to limit the list of dont_touch types to either the cell or net object class. The *`class_name`* value can be either `cell` or `net`.

The following example shows an excerpt of the default `list_dont_touch_types` command output, which lists all dont_touch types for nets and cells:

```
dc_shell> list_dont_touch_types

************************************************
Report : Types of dont_touch
...
************************************************
Class  Type            Description
-------------------------------------------------------------------------------------
net    cts_synthesized Net is synthesized with clock tree synthesis
net    dft_scanbuf     Net connected to input pin of scan compression buffer cell
net    dtn             Net in dont_touch network set by set_dont_touch_network command
net    dtn_charz       Net in dont_touch network through characterization
net    fp_abutted      Net is floorplan abutted net
net    fp_border       Net is dangling on floorplan border
net    idn             Net in ideal network set by set_ideal_network command
net    ilm             Net is part of an interface logic model
net    inh_parent      Net inherits dont_touch from parent
net    mv_ao           Net has derived dont_touch from always-on synthesis
...
cell   cg_mo           Clock-gating cell has dont_touch derived from map_only setting
cell   charz           Cell has dont_touch derived from characterization
cell   dft_scan        Cell is a DFT scan cell
cell   dft_scandef     Cell is SCANDEF generation cell
cell   dft_scg         DFT scan cell has dont_touch because it is replaced by
                       clock-gating cell
cell   dt_conn         Cell connected to dont_touch net
cell   dtn             Cell in dont_touch network set by set_dont_touch_network
```

```
                        command
...
--------------------------------------------------------------------------------
1
```

Creating a Collection of dont_touch Cells and Nets

You can create a collection of dont_touch cells and nets in the current design, relative to the current instance, by using the `get_dont_touch_cells` and `get_dont_touch_nets` commands, respectively. If no cells or nets match the specified criteria, the command returns an empty string.

You can use the `get_dont_touch_cells` and `get_dont_touch_nets` commands at the command prompt, or you can nest them as an argument to another command, such as the `query_objects` command. You can also assign the command results to a variable.

When issued from the command prompt, the `get_dont_touch_cells` and `get_dont_touch_nets` commands behave as if the `query_objects` command has been called to display the objects in the collection. By default, the commands display a maximum of 100 objects. You can change this value by setting the `collection_result_display_limit` variable.

The following example queries the cells with dont_touch due to an `ideal_network` setting under the InstDecode block:

```
dc_shell> get_dont_touch_cells -type idn InstDecode/*
{InstDecode/reset_UPF_LS InstDecode/U38 InstDecode/U36}
```

The following example queries the multivoltage type dont_touch nets that begin with NET in a block named block1:

```
dc_shell> get_dont_touch_nets -type mv* block1/NET*
{block1/NET1QNX block1/NET2QNX}
```

Reporting size_only Cells

You can generate a report of cells with the size_only setting and why the cells have a size_only setting by using the `report_size_only` command. You can generate a report for all the cells in the design, a collection of cells, or a specific cell instance. The command reports size_only cells for the complete design hierarchy. Cells that are not size_only cells are skipped.

The `report_size_only` command can help debug your design. For example, run the `report_size_only` command before running the `compile_ultra` command to report the size_only settings from user constraints. Run the `report_size_only` command again after running `compile_ultra` to report additional size_only settings as a result of optimization.

The following example shows a report of all size_only cells in the current design. It shows a U1 cell with a size_only setting because it is an isolation cell. Cells can contain more than one size_only setting, as shown in cell A/U2. The A/U2 cell has a size_only setting because it is a cross-supply driver and it has an isolation-control signal pin.

```
dc_shell-topo> report_size_only
************************************************
Report : size_only
************************************************

Description of size_only Types:
timing               - Cell with timing constraint
upf_cross_supply_dr  - Cell is cross supply driver
upf_iso              - Isolation cell
upf_iso_ctrl         - Cell with isolation control signal pin
user                 - Cell has specific user-set size_only attribute

Cell    Types
--------------------------------------------------------------------------------
U1      upf_iso
A/U2    upf_cross_supply_dr, upf_iso_ctrl
B/U3    timing
U5      user

...
--------------------------------------------------------------------------------
Summary:

Total 2026 cells with size-only

*****************************
Distribution By Size-Only Type
*****************************
Type                     Size-Only
-----------------------------------
timing                   1299
upf_cross_supply_dr      4
upf_iso                  716
upf_iso_ctrl             4
user                     986
```

To return an alphabetically sorted list of cell size_only types and their descriptions, use the `list_size_only_types` command:

```
dc_shell-topo> list_size_only_types
```

A

Design Example

Optimizing a design can involve using different compile strategies for different levels and components in the design. This appendix shows a design example that uses several compile strategies. Earlier chapters provide detailed descriptions of how to implement each compile strategy. Note that the design example used in this appendix does not represent a real-life application.

This appendix includes the following sections:

You can access the files described in these sections at $SYNOPSYS/doc/syn/guidelines, where $SYNOPSYS is the path to the installation directory.

Design Description

The design example shows how you can constrain designs by using a subset of the commonly used dc_shell commands and how you can use scripts to implement various compile strategies.

The design uses synchronous RTL and combinational logic with clocked D flip-flops.

Figure A-1 shows the block diagram for the design example. The design contains seven modules at the top level: Adder16, CascadeMod, Comparator, Multiply8x8, Multiply16x16, MuxMod, and PathSegment.

Figure A-1 Block Diagram for the Design Example

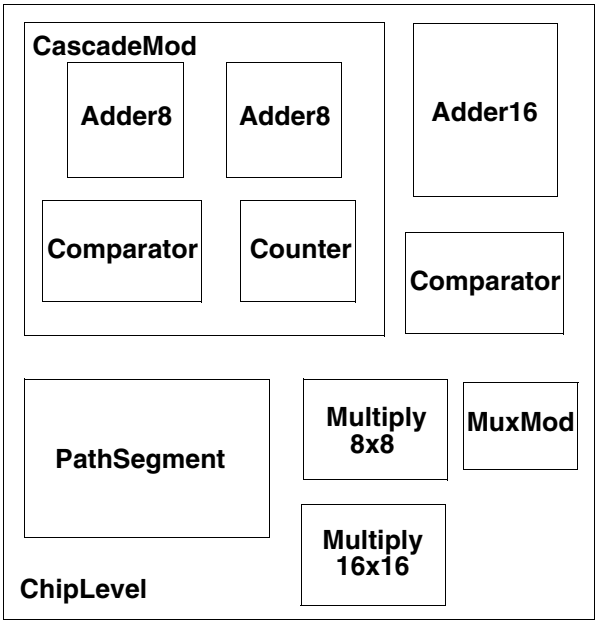
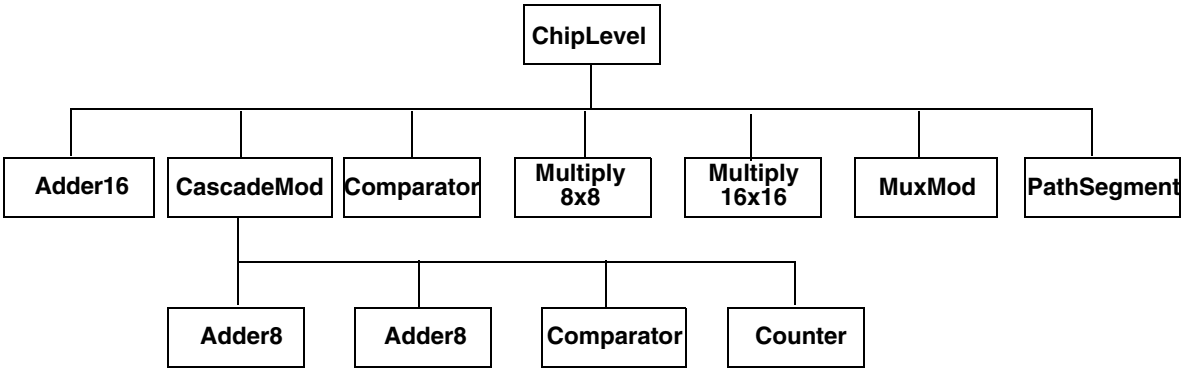


Figure A-2 shows the hierarchy for the design example.

Figure A-2 Hierarchy for the Design Example



The top-level modules and the compilation strategies for optimizing them are

Adder16

Uses registered outputs to make constraining easier. Because the endpoints are the data pins of the registers, you do not need to set output delays on the output ports.

CascadeMod

Uses a hierarchical compile strategy. The compile script for this design sets the constraints at the top level (of CascadeMod) before compilation.

The CascadeMod design instantiates the Adder8 design twice. The script uses the compile-once-don't-touch method for the Comparator module.

Comparator

Is a combinational block. The compile script for this design uses the virtual clock concept to show the use of virtual clocks in a design.

The ChipLevel design instantiates Comparator twice. The compile script (for CascadeMod) uses the compile-once-don't-touch method to resolve the multiple instances.

The compile script specifies wire load model and mode instead of using automatic wire load selection.

Multiply8x8

Shows the basic timing and area constraints used for optimizing a design.

Multiply16x16

Ungroups DesignWare parts before compilation. Ungrouping your hierarchical module might help achieve better synthesis results. The compile script for this module defines a two-cycle path at the primary ports of the module.

MuxMod

Is a combinational block. The script for this design uses the virtual clock concept.

PathSegment

Uses path segmentation within a module. The script uses the `set_multicycle_path` command for a two-cycle path within the module and the `group` command to create a new level of hierarchy.

Example A-1 through Example A-11 provide the Verilog source code for the ChipLevel design.

Example A-1 ChipLevel.v

```
/* Date: May 11, 1995 */
/* Example Circuit for Baseline Methodology for Synthesis */
/* Design does not show any real-life application but rather
   it is used to illustrate the commands used in the Baseline
   Methodology */

module ChipLevel (data16_a, data16_b, data16_c, data16_d, clk, cin, din_a,
                  din_b, sel, rst, start, mux_out, cout1, cout2, s1, s2, op,
                  comp_out1, comp_out2, m32_out, regout);

   input [15:0] data16_a, data16_b, data16_c, data16_d;
   input [7:0] din_a, din_b;
   input [1:0] sel;
   input clk, cin, rst, start;
   input s1, s2, op;
   output [15:0] mux_out, regout;
   output [31:0] m32_out;
   output cout1, cout2, comp_out1, comp_out2;

   wire [15:0] ad16_sout, ad8_sout, m16_out, cnt;

Adder16 u1 (.ain(data16_a), .bin(data16_b), .cin(cin), .cout(cout1),
            .sout(ad16_sout), .clk(clk));

CascadeMod u2 (.data1(data16_a), .data2(data16_b), .cin(cin), .s(ad8_sout),
               .cout(cout2), .clk(clk), .comp_out(comp_out1), .cnt(cnt),
               .rst(rst), .start(start) );

Comparator u3 (.ain(ad16_sout), .bin(ad8_sout), .cp_out(comp_out2));

Multiply8x8 u4 (.op1(din_a), .op2(din_b), .res(m16_out), .clk(clk));

Multiply16x16 u5 (.op1(data16_a), .op2(data16_b), .res(m32_out), .clk(clk));

MuxMod  u6 (.Y_IN(mux_out), .MUX_CNT(sel), .D(ad16_sout), .R(ad8_sout),
            .F(m16_out), .UPC(cnt));

PathSegment u7 (.R1(data16_a), .R2(data16_b), .R3(data16_c), .R4(data16_d),
                .S2(s2), .S1(s1), .OP(op), .REGOUT(regout), .clk(clk));
endmodule
```

Example A-2 Adder16.v

```
module Adder16 (ain, bin, cin, sout, cout, clk);
/* 16-Bit Adder Module */
output [15:0] sout;
output cout;
input [15:0] ain, bin;
input cin, clk;

wire [15:0] sout_tmp, ain, bin;
wire cout_tmp;
reg [15:0] sout, ain_tmp, bin_tmp;
reg cout, cin_tmp;

always @(posedge clk) begin
   cout = cout_tmp;
   sout = sout_tmp;
   ain_tmp = ain;
   bin_tmp = bin;
   cin_tmp = cin;
end
   assign {cout_tmp,sout_tmp} = ain_tmp + bin_tmp + cin_tmp;
endmodule
```

Example A-3 CascadeMod.v

```
module CascadeMod (data1, data2, s, clk, cin, cout, comp_out, cnt, rst, start);
input [15:0] data1, data2;
output [15:0] s, cnt;
input clk, cin, rst, start;
output cout, comp_out;
wire co;

Adder8 u10 (.ain(data1[7:0]), .bin(data2[7:0]), .cin(cin), .clk(clk),
.sout(s[7:0]), .cout(co));
Adder8 u11 (.ain(data1[15:8]), .bin(data2[15:8]), .cin(co), .clk(clk),
.sout(s[15:8]), .cout(cout));
Comparator u12 (.ain(s), .bin(cnt), .cp_out(comp_out));

Counter u13 (.count(cnt), .start(start), .clk(clk), .rst(rst));
endmodule
```

Example A-4 Adder8.v

```
module Adder8 (ain, bin, cin, sout, cout, clk);
/* 8-Bit Adder Module */
output [7:0] sout;
output cout;
input [7:0] ain, bin;
input cin, clk;

wire [7:0] sout_tmp, ain, bin;
wire cout_tmp;
reg [7:0] sout, ain_tmp, bin_tmp;
reg cout, cin_tmp;

always @(posedge clk) begin
   cout = cout_tmp;
   sout = sout_tmp;
   ain_tmp = ain;
   bin_tmp = bin;
   cin_tmp = cin;
end
   assign {cout_tmp,sout_tmp} = ain_tmp + bin_tmp + cin_tmp;
endmodule
```

Example A-5 Counter.v

```
module Counter (count, start, clk, rst);
/* Counter module */
  input clk;
  input rst;
  input start;
  output [15:0] count;

  wire clk;
  reg [15:0] count_N;
  reg [15:0] count;

  always @ (posedge clk or posedge rst)
    begin : counter_S
      if (rst) begin
        count = 0; // reset logic for the block
        end
      else begin
        count = count_N; // set specified registers of the block
        end
    end

  always @ (count or start)
    begin : counter_C
      count_N = count; // initialize outputs of the block
      if (start) count_N = 1; // user specified logic for the block
      else count_N = count + 1;
    end
endmodule
```

Example A-6 Comparator.v

```
module Comparator (cp_out, ain, bin);
/* Comparator for 2 integer values */
output cp_out;
input [15:0] ain, bin;
   assign cp_out = ain < bin;
endmodule
```

Example A-7 Multiply8x8.v

```
module Multiply8x8 (op1, op2, res, clk);
/* 8-Bit multiplier */
input [7:0] op1, op2;
output [15:0] res;
input clk;

wire [15:0] res_tmp;
reg [15:0] res;

always @(posedge clk) begin
   res = res_tmp;
end
assign res_tmp = op1 * op2;
endmodule
```

Example A-8 Multiply16x16.v

```
module Multiply16x16 (op1, op2, res, clk);
/* 16-Bit multiplier */
input [15:0] op1, op2;
output [31:0] res;
input clk;

wire [31:0] res_tmp;
reg [31:0] res;

always @(posedge clk) begin
   res = res_tmp;
end
assign res_tmp = op1 * op2;
endmodule
```

Example A-9 def_macro.v

```
`define DATA 2'b00
`define REG 2'b01
`define STACKIN 2'b10
`define UPCOUT 2'b11
```

Example A-10 MuxMod.v

```
module  MuxMod (Y_IN, MUX_CNT, D, R, F, UPC);
`include "def_macro.v"
   output [15:0] Y_IN;
   input [ 1:0] MUX_CNT;
   input [15:0] D, F, R, UPC;

   reg [15:0] Y_IN;

always @ ( MUX_CNT or D or R or F or UPC ) begin
   case ( MUX_CNT )
      `DATA :
         Y_IN = D ;
      `REG :
         Y_IN = R ;
      `STACKIN :
         Y_IN = F ;
      `UPCOUT :
         Y_IN = UPC;
   endcase
end

endmodule
```

Example A-11 PathSegment.v

```
module PathSegment (R1, R2, R3, R4, S2, S1, OP, REGOUT, clk);
/* Example for path segmentation */
input [15:0] R1, R2, R3, R4;
input S2, S1, clk;
input OP;
output [15:0] REGOUT;

reg [15:0] ADATA, BDATA;
reg [15:0] REGOUT;
reg MODE;

wire [15:0] product ;

always @(posedge clk)
begin : selector_block
   case(S1)
            1'b0: ADATA <= R1;
            1'b1: ADATA <= R2;
            default: ADATA <= 16'bx;
   endcase
   case(S2)
            1'b0: BDATA <= R3;
            1'b1: BDATA <= R4;
            default: ADATA <= 16'bx;
   endcase
end

/* Only Lower Byte gets multiplied */
```

```
// instantiate DW02_mult
DW02_mult #(8,8) U100 (.A(ADATA[7:0]), .B(BDATA[7:0]), .TC(1'b0),
.PRODUCT(product));

always @(posedge clk)
begin : alu_block
   case (OP)
      1'b0 : begin
                REGOUT <= ADATA + BDATA;
             end
      1'b1 : begin
                 REGOUT <= product;
             end
      default : REGOUT <= 16'bx;
   endcase
end

endmodule
```

Setup File

When running the design example, copy the project-specific setup file in Example A-12 to your project working directory. This setup file is written in the Tcl subset and can be used in the Tcl command language. For more information about the Tcl subset, see *Using Tcl With Synopsys Tools*.

For details on the synthesis setup files, see The Setup Files.

Example A-12 .synopsys_dc.setup File

```
# Define the target logic library, symbol library,
# and link libraries
set target_library lsi_10k.db
set symbol_library lsi_10k.sdb
set link_library [concat $target_library "*"]
set search_path [concat $search_path ./src]
set designer "Your Name"
set company "Synopsys, Inc."
# Define path directories for file locations
set source_path "./src/"
set script_path "./scr/"
set log_path "./log/"
set ddc_path "./ddc/"
set db_path "./db/"
set netlist_path "./netlist/"
```

Default Constraints File

The file shown in Example A-13 defines the default constraints for the design. In the scripts that follow, Design Compiler reads this file first for each module. If the script for a module contains additional constraints or constraint values different from those defined in the default constraints file, Design Compiler uses the module-specific constraints.

Example A-13 defaults.con

```
# Define system clock period
set clk_period 20

# Create real clock if clock port is found
if {[sizeof_collection [get_ports clk]] > 0} {
   set clk_name clk
   create_clock -period $clk_period clk
}

# Create virtual clock if clock port is not found
if {[sizeof_collection [get_ports clk]] == 0} {
   set clk_name vclk
   create_clock -period $clk_period -name vclk
}

# Apply default drive strengths and typical loads
# for I/O ports
set_load 1.5 [all_outputs]
set_driving_cell -lib_cell IV [all_inputs]

# If real clock, set infinite drive strength
if {[sizeof_collection [get_ports clk]] > 0} {
   set_drive 0 clk
}

# Apply default timing constraints for modules
set_input_delay 1.2 [all_inputs] -clock $clk_name
set_output_delay 1.5 [all_outputs] -clock $clk_name
set_clock_uncertainty -setup 0.45 $clk_name

# Set operating conditions
set_operating_conditions WCCOM

# Turn on auto wire load selection
# (library must support this feature)
set auto_wire_load_selection true
```

Read Script

Example A-15 provides the Tcl script used to read in the ChipLevel design.

The `read.tcl` script reads design information from the specified Verilog files into memory.

Example A-14 read.tcl

```
read_file -format verilog ChipLevel.v
read_file -format verilog Adder16.v
read_file -format verilog CascadeMod.v
read_file -format verilog Adder8.v
read_file -format verilog Counter.v
read_file -format verilog Comparator.v
read_file -format verilog Multiply8x8.v
read_file -format verilog Multiply16x16.v
read_file -format verilog MuxMod.v
read_file -format verilog PathSegment.v
```

Compile Scripts

Example A-15 through Example A-26 provide the Tcl scripts used to compile the ChipLevel design.

The compile script for each module is named for that module to ease recognition. The initial Tcl script files have the .tcl suffix. Scripts generated by the `write_script` command have the .wtcl suffix.

Example A-15 run.tcl

```
# Initial compile with estimated constraints
source "${script_path}initial_compile.tcl"

current_design ChipLevel
if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}ChipLevel_init.db"
} else {
write -format ddc -hier -o "${ddc_path}ChipLevel_init.ddc"}

# Characterize and write_script for all modules
source "${script_path}characterize.tcl"


# Recompile all modules using write_script constraints
remove_design -all
source "${script_path}recompile.tcl"

current_design ChipLevel
if {[shell_is_in_xg_mode]==0}{
write -hier -out "${db_path}ChipLevel_final.db"
```

```
} else {
write -format ddc -hier -out "${ddc_path}ChipLevel_final.ddc"}
```

Example A-16 initial_compile.tcl

```
# Initial compile with estimated constraints
source "${script_path}read.tcl"

current_design ChipLevel
source "${script_path}defaults.con"

source "${script_path}adder16.tcl"
source "${script_path}cascademod.tcl"
source "${script_path}comp16.tcl"
source "${script_path}mult8.tcl"
source "${script_path}mult16.tcl"
source "${script_path}muxmod.tcl"
source "${script_path}pathseg.tcl"
```

Example A-17 adder16.tcl

```
# Script file for constraining Adder16
set rpt_file "adder16.rpt"
set design "adder16"

current_design Adder16
source "${script_path}defaults.con"

# Define design environment
set_load 2.2 sout
set_load 1.5 cout
set_driving_cell -lib_cell FD1 [all_inputs]
set_drive 0 $clk_name

# Define design constraints
set_input_delay 1.35 -clock $clk_name {ain bin}
set_input_delay 3.5 -clock $clk_name cin
set_max_area 0

compile

if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}${design}.db"
} else {
write -format ddc -hier -o "${ddc_path}${design}.ddc"}

source "${script_path}report.tcl"
```

Example A-18 cascademod.tcl

```
# Script file for constraining CascadeMod
# Constraints are set at this level and then a
# hierarchical compile approach is used

set rpt_file "cascademod.rpt"
set design "cascademod"

current_design CascadeMod
source "${script_path}defaults.con"

# Define design environment
set_load 2.5 [all_outputs]
set_driving_cell -lib_cell FD1 [all_inputs]
set_drive 0 $clk_name

# Define design constraints
set_input_delay 1.35 -clock $clk_name {data1 data2}
set_input_delay 3.5 -clock $clk_name cin
set_input_delay 4.5 -clock $clk_name {rst start}
set_output_delay 5.5 -clock $clk_name comp_out
set_max_area 0

# Use compile-once, dont_touch approach for Comparator
set_dont_touch u12

compile

if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}${design}.db"
} else {
write -format ddc -hier -o "${ddc_path}${design}.ddc"}

source "${script_path}report.tcl"
```

Example A-19 comp16.tcl

```
# Script file for constraining Comparator
set rpt_file "comp16.rpt"
set design "comp16"

current_design Comparator
source "${script_path}defaults.con"

# Define design environment
set_load 2.5 cp_out
set_driving_cell -lib_cell FD1 [all_inputs]

# Override auto wire load selection
set_wire_load_model -name "05x05"
set_wire_load_mode enclosed

# Define design constraints
set_input_delay 1.35 -clock $clk_name {ain bin}
set_output_delay 5.1 -clock $clk_name {cp_out}
set_max_area 0

compile

if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}${design}.db"
} else {
write -format ddc -hier -o "${ddc_path}${design}.ddc"}

source "${script_path}report.tcl"
```

Example A-20 mult8.tcl

```
# Script file for constraining Multiply8x8
set rpt_file "mult8.rpt"
set design "mult8"

current_design Multiply8x8
source "${script_path}defaults.con"

# Define design environment
set_load 2.2 res
set_driving_cell -lib_cell FD1P [all_inputs]
set_drive 0 $clk_name

# Define design constraints
set_input_delay 1.35 -clock $clk_name {op1 op2}
set_max_area 0

compile

if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}${design}.db"
} else {
write -format ddc -hier -o "${ddc_path}${design}.ddc"}

source "${script_path}report.tcl"
```

Example A-21 mult16.tcl

```
# Script file for constraining Multiply16x16
set rpt_file "mult16.rpt"
set design "mult16"

current_design Multiply16x16
source "${script_path}defaults.con"

# Define design environment
set_load 2.2 res
set_driving_cell -lib_cell FD1 [all_inputs]
set_drive 0 $clk_name

# Define design constraints
set_input_delay 1.35 -clock $clk_name {op1 op2}
set_max_area 0

# Define multicycle path for multiplier
set_multicycle_path 2 -from [all_inputs] \
   -to [all_registers -data_pins -edge_triggered]

# Ungroup DesignWare parts
set designware_cells [get_cells \
   -filter "@is_oper==true"]
if {[sizeof_collection $designware_cells] > 0} {
   set_ungroup $designware_cells true
}

compile

if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}${design}.db"
} else {
write -format ddc -hier -o "${ddc_path}${design}.ddc"}

source "${script_path}report.tcl"
report_timing_requirements -ignore \
   >> "${log_path}${rpt_file}"
```

Example A-22 muxmod.tcl

```
# Script file for constraining MuxMod
set rpt_file "muxmod.rpt"
set design "muxmod"

current_design MuxMod
source "${script_path}defaults.con"

# Define design environment
set_load 2.2 Y_IN
set_driving_cell -lib_cell FD1 [all_inputs]

# Define design constraints
set_input_delay 1.35 -clock $clk_name {D R F UPC}
set_input_delay 2.35 -clock $clk_name MUX_CNT
set_output_delay 5.1 -clock $clk_name {Y_IN}
set_max_area 0

compile

if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}${design}.db"
} else {
write -format ddc -hier -o "${ddc_path}${design}.ddc"}

source "${script_path}report.tcl"
```

Example A-23 pathseg.tcl

```
# Script file for constraining path_segment
set rpt_file "pathseg.rpt"
set design "pathseg"

current_design PathSegment
source "${script_path}defaults.con"

# Define design environment
set_load 2.5 [all_outputs]
set_driving_cell -lib_cell FD1 [all_inputs]
set_drive 0 $clk_name

# Define design rules
set_max_fanout 6 {S1 S2}

# Define design constraints
set_input_delay 2.2 -clock $clk_name {R1 R2}
set_input_delay 2.2 -clock $clk_name {R3 R4}
set_input_delay 5 -clock $clk_name {S2 S1 OP}
set_max_area 0

# Perform path segmentation for multiplier
group -design mult -cell mult U100
set_input_delay 10 -clock $clk_name mult/product*
set_output_delay 5 -clock $clk_name mult/product*
set_multicycle_path 2 -to mult/product*

compile

if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}${design}.db"
} else {
write -format ddc -hier -o "${ddc_path}${design}.ddc"}

source "${script_path}report.tcl"
report_timing_requirements -ignore \
   >> "${log_path}${rpt_file}"
```

Example A-24 characterize.tcl

```
# Characterize and write_script for all modules
current_design ChipLevel
characterize u1
current_design Adder16
write_script > "${script_path}adder16.wtcl"

current_design ChipLevel
characterize u2
current_design CascadeMod
write_script -format tcl"${script_path}cascademod.wtcl"

current_design ChipLevel
characterize u3
current_design Comparator
write_script -format tcl > "${script_path}comp16.wtcl"

current_design ChipLevel
characterize u4
current_design Multiply8x8
write_script -format tcl > "${script_path}mult8.wtcl"

current_design ChipLevel
characterize u5
current_design Multiply16x16
write_script -format tcl > "${script_path}mult16.wtcl"

current_design ChipLevel
characterize u6
current_design MuxMod
write_script -format tcl > "${script_path}muxmod.wtcl"

current_design ChipLevel
characterize u7
current_design PathSegment

echo "current_design PathSegment" > \
   "${script_path}pathseg.wtcl"

echo "group -design mult -cell mult U100" >> \
   "${script_path}pathseg.wtcl"
write_script -format tcl >> "${script_path}pathseg.wtcl"
```

Example A-25 recompile.tcl

```
source "${script_path}read.tcl"

current_design ChipLevel
source "${script_path}defaults.con"

source "${script_path}adder16.wtcl"
compile
if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}adder16_wtcl.db"
} else {
write -format ddc -hier -o "${ddc_path}adder16_wtcl.ddc"}
set rpt_file adder16_wtcl.rpt
source "${script_path}report.tcl"

source "${script_path}cascademod.wtcl"
dont_touch u12
compile
if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}cascademod_wtcl.db"
} else {
write -format ddc -hier -o "${ddc_path}cascademod_wtcl.ddc"}
set rpt_file cascade_wtcl.rpt
source "${script_path}report.tcl"

source "${script_path}comp16.wtcl"
compile
if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}comp16_wtcl.db"
} else {
write -format ddc -hier -o "${ddc_path}comp16_wtcl.ddc"}
set rpt_file comp16_wtcl.rpt
source "${script_path}report.tcl"

source "${script_path}mult8.wtcl"
compile
if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}mult8_wtcl.db"
} else {
write -format ddc -hier -o "${ddc_path}mult8_wtcl.ddc"}
set rpt_file mult8_wtcl.rpt
source "${script_path}report.tcl"

source "${script_path}mult16.wtcl"
compile -ungroup_all
if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}mult16_wtcl.db"
} else {
write -format ddc -hier -o "${ddc_path}mult16_wtcl.ddc"}
set rpt_file mult16_wtcl.rpt
source "${script_path}report.tcl"
report_timing_requirements -ignore \
```

```
            >> "${log_path}${rpt_file}"

source "${script_path}muxmod.wtcl"
compile
if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}muxmod_wtcl.db"
} else {
write -format ddc -hier -o "${ddc_path}muxmod_wtcl.ddc"}
set rpt_file muxmod_wtcl.rpt
source "${script_path}report.tcl"

source "${script_path}pathseg.wtcl"
compile
if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}pathseg_wtcl.db"
} else {
write -format ddc -hier -o "${ddc_path}pathseg_wtcl.ddc"}
set rpt_file pathseg_wtcl.rpt
source "${script_path}report.tcl"
report_timing_requirements -ignore \
   >> "${log_path}${rpt_file}"
```

Example A-26 report.tcl

```
# This script file creates reports for all modules
set maxpaths 15

check_design > "${log_path}${rpt_file}"
report_area >> "${log_path}${rpt_file}"
report_design >> "${log_path}${rpt_file}"
report_cell >> "${log_path}${rpt_file}"
report_reference >> "${log_path}${rpt_file}"
report_port -verbose >> "${log_path}${rpt_file}"
report_net >> "${log_path}${rpt_file}"
report_compile_options >> "${log_path}${rpt_file}"
report_constraint -all_violators -verbose \
   >> "${log_path}${rpt_file}"
report_timing -path end >> "${log_path}${rpt_file}"
report_timing -max_path $maxpaths \
   >> "${log_path}${rpt_file}"
report_qor >> "${log_path}${rpt_file}"
```

B

Basic Commands

This appendix lists the basic dc_shell commands for synthesis and provides a brief description for each command. The commands are grouped in the following sections:

- Commands for Defining Design Rules
- Commands for Defining Design Environments
- Commands for Setting Design Constraints
- Commands for Analyzing and Resolving Design Problems

Within each section the commands are listed in alphabetical order.

Commands for Defining Design Rules

The commands that define design rules are

`set_max_capacitance`

Sets a maximum capacitance for the nets attached to the specified ports or to all the nets in a design.

`set_max_fanout`

Sets the expected fanout load value for output ports.

`set_max_transition`

Sets a maximum transition time for the nets attached to the specified ports or to all the nets in a design.

`set_min_capacitance`

Sets a minimum capacitance for the nets attached to the specified ports or to all the nets in a design.

Commands for Defining Design Environments

The commands that define the design environment are

`set_drive`

Sets the drive value of input or inout ports. The `set_drive` command is superseded by the `set_driving_cell` command.

`set_driving_cell`

Sets attributes on input or inout ports, specifying that a library cell or library pin drives the ports. This command associates a library pin with an input port so that delay calculators can accurately model the drive capability of an external driver.

`set_fanout_load`

Defines the external fanout load values on output ports.

`set_load`

Defines the external load values on input and output ports and nets.

`set_operating_conditions`

Defines the operating conditions for the current design.

`set_wire_load_model`

Sets the wire load model for the current design or for the specified ports. With this command, you can specify the wire load model to use for the external net connected to the output port.

Commands for Setting Design Constraints

The basic commands that set design constraints are

`create_clock`

Creates a clock object and defines its waveform in the current design.

`set_clock_latency, set_clock_uncertainty, set_propagated_clock, set_clock_transition`

Sets clock attributes on clock objects or flip-flop clock pins.

`set_input_delay`

Sets input delay on pins or input ports relative to a clock signal.

`set_max_area`

Specifies the maximum area for the current design.

`set_output_delay`

Sets output delay on pins or output ports relative to a clock signal.

The advanced commands that set design constraints are

`group_path`

Groups a set of paths or endpoints for cost function calculation. This command is used to create path groups, to add paths to existing groups, or to change the weight of existing groups.

`set_false_path`

Marks paths between specified points as false. This command eliminates the selected paths from timing analysis.

`set_max_delay`

Specifies a maximum delay target for selected paths in the current design.

`set_min_delay`

Specifies a minimum delay target for selected paths in the current design.

`set_multicycle_path`

Allows you to specify the time of a timing path to exceed the time of one clock signal.

Commands for Analyzing and Resolving Design Problems

The commands for analyzing and resolving design problems are

`all_connected`

Lists all fanouts on a net.

`all_registers`

Lists sequential elements or pins in a design.

`check_design`

Checks the internal representation of the current design for consistency and issues error and warning messages as appropriate.

`check_timing`

Checks the timing attributes placed on the current design.

`get_attribute`

Reports the value of the specified attribute.

`link`

Locates the reference for each cell in the design.

`report_area`

Provides area information and statistics on the current design.

`report_attribute`

Lists the attributes and their values for the selected object. An object can be a cell, net, pin, port, instance, or design.

`report_cell`

Lists the cells in the current design and their cell attributes.

`report_clock`

Displays clock-related information about the current design.

`report_constraint`

Lists the constraints on the current design and their cost, weight, and weighted cost.

`report_delay_calculation`

Reports the details of a delay arc calculation.

`report_design`

Displays the operating conditions, wire load model and mode, timing ranges, internal input and output, and disabled timing arcs defined for the current design.

`report_hierarchy`

Lists the subdesigns of the current design.

`report_net`

Displays net information for the design of the current instance, if set; otherwise, displays net information for the current design.

`report_path_group`

Lists all timing path groups in the current design.

`report_port`

Lists information about ports in the current design.

`report_qor`

Displays information about the quality of results and other statistics for the current design.

`report_resources`

Displays information about the resource implementation.

`report_timing`

Lists timing information for the current design.

`report_timing_requirements`

Lists timing path requirements and related information.

`report_transitive_fanin`

Lists the fanin logic for selected pins, nets, or ports of the current instance.

`report_transitive_fanout`

Lists the fanout logic for selected pins, nets, or ports of the current instance.

C

Predefined Attributes

This appendix contains tables that list the Design Compiler predefined attributes for each object type.

Table C-1 Clock Attributes

Attribute name	Value
`clock_fall_transition`	float
`clock_min_fall_transition`	float
`clock_min_rise_transition`	float
`clock_rise_transition`	float
`dont_touch_network`	{true, false}
`dont_touch_network_no_propagate`	{true, false}
`fall_delay`	float
`fall_min_delay`	float
`fix_hold`	boolean
`full_name`	string

Table C-1 Clock Attributes (Continued)

Attribute name	Value
max_time_borrow	float
minus_uncertainty	float
name	string
object_class	string
period	float
plus_uncertainty	float
propagated_clock	boolean
rise_delay	float
rise_min_delay	float

Table C-2 Design Attributes

Attribute name	Value
actual_max_net_capacitance	float
actual_min_net_capacitance	float
boundary_optimization	{true, false}
default_flip_flop_type	internally generated string
default_flip_flop_type_exact	library_cell_name
default_latch_type	library_cell_name
design_type	{equation, fsm, pla, netlist}
dont_touch	{true, false}
dont_touch_network	{true, false}
driven_by_logic_one	{true, false}

Table C-2 Design Attributes (Continued)

Attribute name	Value
driven_by_logic_zero	{true, false}
driving_cell_dont_scale	string
driving_cell_fall	string
driving_cell_from_pin_fall	string
driving_cell_from_pin_rise	string
driving_cell_library_fall	string
driving_cell_library_rise	string
driving_cell_multiplier	float
driving_cell_pin_fall	string
driving_cell_pin_rise	string
driving_cell_rise	string
fall_drive	float
fanout_load	float
flatten	{true, false}
flatten_effort	{true, false}
flatten_minimize	{true, false}
flatten_phase	{true, false}
flip_flop_type	internally generated string
flip_flop_type_exact	library_cell_name
is_black_box	{true, false}
is_combinational	{true, false}
is_hierarchical	{true, false}

Table C-2 Design Attributes (Continued)

Attribute name	Value
`is_mapped`	{true, false}
`is_sequential`	{true, false}
`is_test_circuitry`	{true, false}
`is_unmapped`	{true, false}
`latch_type`	internally generated string
`latch_type_exact`	library_cell_name
`load`	float
`local_link_library`	design_or_lib_file_name
`max_capacitance`	float
`max_fanout`	float
`max_time_borrow`	float
`max_transition`	float
`min_capacitance`	float
`minus_uncertainty`	float
`output_not_used`	{true, false}
`pad_location` (XNF only)	string
`part` (XNF only)	string
`plus_uncertainty`	float
`port_direction`	{in, inout, out, unknown}
`port_is_pad`	{true, false}
`ref_name`	reference_name
`rise_drive`	float

Table C-2 Design Attributes (Continued)

Attribute name	Value
structure	{true, false}
ungroup	{true, false}
wired_logic_disable	{true, false}

Table C-3 Library Attributes

Attribute name	Value
default_values	float
k_process_values	float
k_temp_values	float
k_volt_values	float
nom_process	float
nom_temperature	float
nom_voltage	float

Table C-4 Library Cell Attributes

Attribute name	Value
area	float
dont_touch	{true, false}
dont_use	{true, false}
preferred	{true, false}

Table C-5 Net Attributes

Attribute name	Value
ba_net_resistance	float
dont_touch	{true, false}
load	float
subtract_pin_load	{true, false}
wired_and	{true, false}
wired_or	{true, false}

Table C-6 Pin Attributes

Attribute name	Value
disable_timing	{true, false}
max_time_borrow	float
pin_direction	{in, inout, out, unknown}

Table C-7 Reference Attributes

Attribute name	Value
dont_touch	{true, false}
is_black_box	{true, false}
is_combinational	{true, false}
is_hierarchical	{true, false}
is_mapped	{true, false}
is_sequential	{true, false}
is_unmapped	{true, false}
ungroup	{true, false}

Glossary

annotation
A piece of information attached to an object in the design, such as a capacitance value attached to a net; the process of attaching such a piece of information to an object in the design.

back-annotate
To update a circuit design by using extraction and other post-processing information that reflects implementation-dependent characteristics of the design, such as pin selection, component location, or parasitic electrical characteristics. Back-annotation allows a more accurate timing analysis of the final circuit. The data is generated by another tool after layout and passed to the synthesis environment. For example, the design database might be updated with actual interconnect delays; these delays are calculated after placement and routing—after exact interconnect lengths are known.

cell
See instance.

clock
A source of timed pulses with a periodic behavior. A clock synchronizes the propagation of data signals by controlling sequential elements, such as flip-flops and registers, in a digital circuit. You define clocks with the `create_clock` command.

Clocks you create by using the `create_clock` command ignore delay effects of the clock network. Therefore, for accurate timing analysis, you describe the clock network in terms of its latency and skew. See also clock latency and clock skew.

clock gating
The control of a clock signal by logic (other than inverters or buffers), either to shut down the clock signal at selected times or to modify the clock pulse characteristics.

clock latency
The amount of time that a clock signal takes to be propagated from the clock source to a specific point in the design. Clock latency is the sum of source latency and network latency.

Source latency is the propagation time from the actual clock origin to the clock definition point in the design. Network latency is the propagation time from the clock definition point in the design to the clock pin of the first register.

You use the `set_clock_latency` command to specify clock latency.

clock skew
The maximum difference between the arrival of clock signals at registers in one clock domain or between clock domains. Clock skew is also known as clock uncertainty. You use the `set_clock_uncertainty` command to specify the skew characteristics of one or more clock networks.

clock source
The pin or port where the clock waveform is applied to the design. The clock signal reaches the registers in the transitive fanout of all its sources. A clock can have multiple sources.

You use the `create_clock` command with the `source_object` option to specify clock sources.

clock tree
The combinational logic between a clock source and registers in the transitive fanout of that source. Clock trees, also known as clock networks, are synthesized by vendors based on the physical placement data at registers in one clock domain or between clock domains.

clock uncertainty
See clock skew.

core
A predesigned block of logic employed as a building block for ASIC designs.

critical path
The path through a circuit with the longest delay. The speed of a circuit depends on the slowest register-to-register delay. The clock period cannot be shorter than this delay or the signal will not reach the next register in time to be clocked.

datapath
A logic circuit in which data signals are manipulated using arithmetic operators such as adders, multipliers, shifters, and comparators.

current design
The active design (the design being worked on). Most commands are specific to the current design, that is, they operate within the context of the current design. You specify the current design with the `current_design` command.

current instance
The instance in a design hierarchy on which instance-specific commands operate by default. You specify the current instance with the `current_instance` command.

design constraints
The designer's specification of design performance goals, that is, the timing and environmental restrictions under which synthesis is to be performed. Design Compiler uses these constraints—for example, low power, small area, high-speed, or minimal cost—to direct the optimization of a design to meet area and timing goals.

There are two categories of design constraints: design rule constraints and design optimization constraints.

- Design rule constraints are supplied in the logic library. For proper functioning of the fabricated circuit, they must not be violated.
- Design optimization constraints define timing and area optimization goals.

Design Compiler optimizes the synthesis of the design in accordance with both sets of constraints; however, design rule constraints have higher priority.

false path
A path that you do not want Design Compiler to consider during timing analysis. An example of such a path is one between two multiplexed blocks that are never enabled at the same time, that is, a path that cannot propagate a signal.

You use the `set_false_path` command to disable timing-based synthesis on a path-by-path basis. The command removes timing constraints on the specified path.

fanin
The pins driving an endpoint pin, port, or net (also called sink). A pin is considered to be in the fanin of a sink if there is a timing path through combinational logic from the pin to the sink. Fanin tracing starts at the clock pins of registers or valid startpoints. Fanin is also known as transitive fanin.

You use the `report_transitive_fanin` command to report the fanin of a specified sink pin, port, or net.

fanout
The pins driven by a source pin, port, or net. A pin is considered to be in the fanout of a source if there is a timing path through combinational logic from the source to that pin or port. Fanout tracing stops at the data pin of a register or at valid endpoints. Fanout is also known as transitive fanout or timing fanout.

You use the `report_transitive_fanout` command to report the fanout of a specified source pin, port, or net.

fanout load
A unitless value that represents a numerical contribution to the total fanout. Fanout load is not the same as load, which is a capacitance value.

Design Compiler models fanout restrictions by associating a `fanout_load` attribute with each input pin and a `max_fanout` attribute with each output (driving) pin on a cell and ensures that the sum of fanout loads is less than the `max_fanout` value.

flatten
To convert combinational logic paths of the design to a two-level, sum-of-products representation. During flattening, Design Compiler removes all intermediate terms, and therefore all associated logic structure, from a design. Flattening is constraint based.

forward-annotate
To transfer data from the synthesis environment to other tools used later in the design flow. For example, delay and constraints data in Standard Delay Format (SDF) might be transferred from the synthesis environment to guide place and route tools.

generated clock
A clock signal that is generated internally by the integrated circuit itself; a clock that does not come directly from an external source. An example of a generated clock is a divide-by-2 clock generated from the system clock. You define a generated clock with the `create_generated_clock` command.

hold time
The time that a signal on the data pin must remain stable after the active edge of the clock. The hold time creates a minimum delay requirement for paths leading to the data pin of the cell.

You calculate the hold time by using the formula

```
hold = max clock delay - min data delay
```

ideal clock
A clock that is considered to have no delay as it propagates through the clock network. The ideal clock type is the default for Design Compiler. You can override the default behavior (using the `set_clock_latency` and `set_propagated_clock` commands) to obtain nonzero clock network delay and specify information about the clock network delays.

ideal net
Nets that are assigned ideal timing conditions—that is, latency, transition time, and capacitance are assigned a value of zero. Such nets are exempt from timing updates, delay optimization, and design rule fixing. Defining certain high-fanout nets that you

intend to synthesize separately (such as scan-enable and reset nets) as ideal nets can reduce runtime.

You use the `set_ideal_net` command to specify nets as ideal nets.

input delay
A constraint that specifies the minimum or maximum amount of delay from a clock edge to the arrival of a signal at a specified input port.

You use the `set_input_delay` command to set the input delay on a pin or input port relative to a specified clock signal.

instance
An occurrence in a circuit of a reference (a library component or design) loaded in memory; each instance has a unique name. A design can contain multiple instances; each instance points to the same reference but has a unique name to distinguish it from other instances. An instance is also known as a cell.

leaf cell
A fundamental unit of logic design. A leaf cell cannot be broken into smaller logic units. Examples are NAND gates and inverters.

link library
A logic library that Design Compiler uses to resolve cell references. Link libraries can contain technology libraries and design files. Link libraries also contain the descriptions of cells (library cells as well as subdesigns) in a mapped netlist.

Link libraries include both local link libraries (`local_link_library` attribute) and system link libraries (`link_library` variable).

logic library
A library of ASIC cells that are available to Design Compiler during the synthesis process. A logic library can contain area, timing, power, and functional information about each ASIC cell. The logic of each library is specific to a particular ASIC vendor.

multicycle path
A path for which data takes more than one clock cycle to propagate from the startpoint to the endpoint.

You use the `set_multicycle_path` command to specify the number of clock cycles Design Compiler should use to determine when data is required at a particular endpoint.

netlist
A file in ASCII or binary format that describes a circuit schematic— the netlist contains a list of circuit elements and interconnections in a design. Netlist transfer is the most common way of moving design information from one design system or tool to another.

operating conditions
The process, voltage, and temperature ranges a design encounters. Design Compiler optimizes your design according to an operating point on the process, voltage, and temperature curves and scales cell and wire delays according to your operating conditions.

By default, operating conditions are specified in a logic library in an `operating_conditions` group.

optimization
The step in the logic synthesis process in which Design Compiler attempts to implement a combination of logic library cells that best meets the functional, timing, and area requirements of the design.

output delay
A constraint that specifies the minimum or maximum amount of delay from an output port to the sequential element that captures data from the output port. This constraint establishes the times at which signals must be available at the output port to meet the setup and hold requirements of the sequential element.

You use the `set_output_delay` command to set the output delay on a pin or output port relative to a specified clock signal.

pad cell
A special cell at the chip boundaries that allows connection or communication with integrated circuits outside the chip.

path group
A group of related paths, grouped either implicitly by the `create_clock` command or explicitly by the `group_path` command. By default, paths whose endpoints are clocked by the same clock are assigned to the same path group.

pin
A part of a cell that provides for input and output connections. Pins can be bidirectional. The ports of a subdesign are pins within the parent design.

propagated clock
A clock that incurs delay through the clock network. Propagated clocks are used to determine clock latency at register clock pins. Registers clocked by a propagated clock have edge times skewed by the path delay from the clock source to the register clock pin.

You use the `set_propagated_clock` command to specify that clock latency be propagated through the clock network.

real clock
A clock that has a source, meaning its waveform is applied to pins or ports in the design. You create a real clock by using a `create_clock` command and including a source list of ports or pins. Real clocks can be either ideal or propagated.

reference
A library component or design that can be used as an element in building a larger circuit. The structure of the reference may be a simple logic gate or a more complex design (RAM core or CPU). A design can contain multiple occurrences of a reference; each occurrence is an instance. See also instance.

RTL
RTL, or register transfer level, is a register-level description of a digital electronic circuit. In a digital circuit, registers store intermediate information between clock cycles; thus, RTL describes the intermediate information that is stored, where it is stored within the design, and how it is transferred through the design. RTL models circuit behavior at the level of data flow between a set of registers. This level of abstraction typically contains little timing information, except for references to a set of clock edges and features.

setup time
The time that a signal on the data pin must remain stable before the active edge of the clock. The setup time creates a maximum delay requirement for paths leading to the data pin of a cell.

You calculate the setup time by using the formula

```
setup = max data delay - min clock delay
```

slack
A value that represents the difference between the actual arrival time and the required arrival time of data at the path endpoint in a mapped design. Slack values can be positive, negative, or zero.

A positive slack value represents the amount by which the delay of a path can be increased without violating any timing constraints. A negative slack value represents the amount by which the delay of a path must be reduced to meet its timing constraints.

structuring
To add intermediate variables and logic structure to a design, which can result in reduced design area. Structuring is constraint based. It is best applied to noncritical timing paths.

By default, Design Compiler structures your design.

synthesis
A software process that generates an optimized gate-level netlist, which is based on a logic library, from an input IC design. Synthesis includes reading the HDL source code and optimizing the design from that description.

symbol library
A library that contains the schematic symbols for all cells in a particular ASIC library. Design Compiler uses symbol libraries to generate the schematic. You can use Design Vision to view the schematic.

target library
The logic library to which Design Compiler maps during optimization. Target libraries contain the cells used to generate the netlist and definitions for the design's operating conditions.

timing exception
An exception to the default (single-cycle) timing behavior assumed by Design Compiler. For Design Compiler to analyze a circuit correctly, you must specify each timing path in the design that does not conform to the default behavior. Examples of timing exceptions include false paths, multicycle paths, and paths that require a specific minimum or maximum delay time different from the default calculated time.

timing path
A point-to-point sequence that dictates data propagation through a design. Data is launched by a clock edge at a startpoint, propagated through combinational logic elements, and captured at an endpoint by another clock edge. The startpoint of a timing path is an input port or clock pin of a sequential element. The endpoint of a timing path is an output port or a data pin of a sequential element.

transition delay
A timing delay caused by the time it takes the driving pin to change voltage state.

ungroup
To remove hierarchy levels in a design. Ungrouping merges subdesigns of a given level of the hierarchy into the parent cell or design.

You use the `ungroup` command or the `compile` command with the `auto_ungroup` option to ungroup designs.

uniquify
To resolve multiple cell references to the same design in memory.

The uniquify process creates unique design copies with unique design names for each instantiated cell that references the original design.

virtual clock
A clock that exists in the system but is not part of the block. A virtual clock does not clock any sequential devices within the current design and is not associated with a pin or port. You use a virtual clock as a reference for specifying input and output delays relative to a clock outside the block.

You use the `create_clock` command without a list of associated pins or ports to create a virtual clock.

wire load model

An estimate of a net's RC parasitics based on the net's fanout, in the absence of placement and routing information. The estimated capacitance and resistance are used to calculate the delay of nets. After placement and routing, you should back-annotate the design with detailed information about the net delay.

The wire load model is shipped with the logic library; vendors develop the wire load model based on statistical information specific to the vendor's process. You can also custom-generate the model based on back-annotation. The model includes coefficients for area, capacitance, and resistance per unit length, and a fanout-to-length table for estimating net lengths (the number of fanouts determines a nominal length).

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W