DC Explorer
User Guide

Version O-2018.06, June 2018

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.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

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

Copyright Notice for the Command-Line Editing Feature

Copyright Notice for the Line-Editing Library

Contents

Preface

This preface includes the following sections:

About This Manual

The DC Explorer tool enables early RTL exploration that leads to a better starting point for RTL synthesis and speeds up design implementation. The *DC Explorer User Guide* introduces synthesis concepts and commands, describes tolerance for incomplete or mismatched data, and provides examples for basic synthesis strategies.

This manual does not cover asynchronous design, I/O pad synthesis, test synthesis, simulation, physical design techniques (such as floorplanning or place and route), 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.

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 DC Explorer, see the documentation on Synopsys 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 Compiler®
- Design Vision™
- DesignWare® components
- DFT Compiler and DFTMAX™
- HDL Compiler™

- IC Compiler™
- IC Compiler™ II
- Library Compiler™
- PrimeTime® Suite
- Power Compiler™

Release Notes

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

To see the *DC Explorer Release Notes*,

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

   https://solvnet.synopsys.com/DownloadCenter

2. Select DC Explorer, 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

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

To access the SolvNet site, 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 sign up for an account.

If you need help using the SolvNet site, 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 the SolvNet site 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 https://www.synopsys.com/support/global-support-centers.html
- Telephone your local support center.
  - Call (800) 245-8005 from within North America.
  - Find other local support center telephone numbers at https://www.synopsys.com/support/global-support-centers.html

1 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 early RTL exploration, leading to a better starting point for RTL synthesis and accelerating design implementation.

With tolerance for incomplete design data and significantly faster runtimes than full synthesis, DC Explorer provides early visibility into implementation results that are typically within 10 percent of the results produced by Design Compiler topographical mode. The tool lets you efficiently 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. It also generates an early netlist that can be used to begin physical exploration in IC Compiler. DC Explorer lets you create and modify floorplans very early in the design cycle with push-button access to IC Compiler design planning.

To learn more about the basics of DC Explorer, see

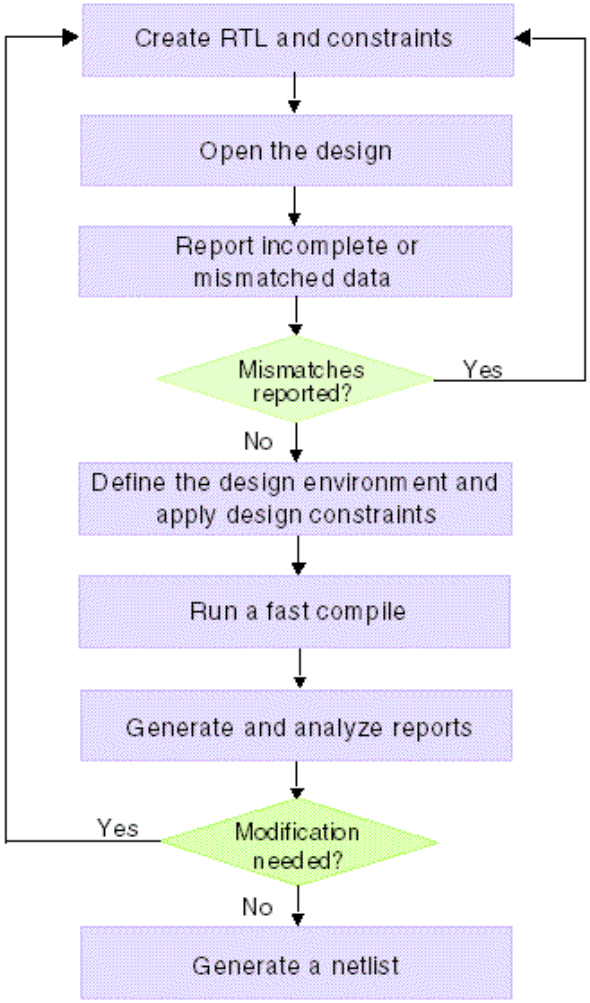
- DC Explorer Design Flow
- Running the DC Explorer Design Flow
- Benefits of Using DC Explorer
- DC Explorer and Design Compiler Compatibility
- Floorplan Exploration in IC Compiler

DC Explorer Design Flow

The DC Explorer design flow provides fast synthesis and helps you detect incomplete and mismatched data. DC Explorer takes the following input files: RTL netlists, SDC files, logic libraries, physical libraries (optional), and floorplans (optional). The tool generates reports, messages, and a netlist that can be used as input to IC Compiler for early floorplanning.

Figure 1-1 shows a typical design flow in DC Explorer.

Figure 1-1: DC Explorer Design Flow



See Also

- Running the DC Explorer Design Flow

Running the DC Explorer Design Flow

To run the DC Explorer design flow,

1. Create the RTL and constraints.

   The input design files for DC Explorer are RTL descriptions, Synopsys Design Constraints (SDC) files, logic libraries, physical libraries (optional), and floorplans (optional).

2. Open the design.

   You can use either of the following methods to open the design:

   - The `analyze` and `elaborate` commands
   - The read command, such as the `read_file` command

3. Report incomplete or mismatched data by using the `current_design`, `link`, `check_design` and `report_design_mismatch` commands.

   You might not need to fix the incomplete or mismatched data before you proceed.

4. Define the design environment and apply the design constraints.

   - Use the following commands to set up the design environment: `set_operating_conditions`, `set_driving_cell`, `set_load`, and `set_fanout_load`.
   - To apply the design constraints, use the `read_sdc` command.

5. Run a fast compile and optimize the design by using the `compile_exploration` command.

6. Create categorized timing reports in HTML format by using the `create_qor_snapshot` and `query_qor_snapshot` commands.

   Check the reports for missing constraints and infeasible paths; you might need to fix these mismatches before you proceed.

7. Write the netlist by using the `write_file` command with the `-format` option.

See Also

- DC Explorer Design Flow
- Tolerance for Incomplete or Mismatched Data
- Setting Up the Libraries
- Opening Designs
- Defining the Design Environment

- Defining Design Constraints
- Optimization
- Reporting Quality of Results
- Saving Designs
- Design File Management for Synthesis

Benefits of Using DC Explorer

DC Explorer can improve your productivity during development of RTL and constraints by enabling fast synthesis in the early design stages. Using DC Explorer provides the following benefits:

- Produces fast synthesis results
- Tolerates incomplete or mismatched design data
- Provides accurate preliminary synthesis with timing, area, and power values typically within 10 percent of the final implementation, using the same set of constraints
- Reconciles design mismatches automatically and reports them in a concise format
- Detects infeasible timing paths while optimizing other critical paths
- Generates easy-to-navigate HTML timing reports for fast debugging
- Creates a netlist that can be used for floorplanning in IC Compiler

DC Explorer and Design Compiler Compatibility

DC Explorer is compatible with Design Compiler in the following ways. By default, DC Explorer

- Supports most Design Compiler commands in the following categories: constraints, scripting, reports, design handling, design inputs and outputs, modeling, and library setup
- Runs existing Design Compiler scripts
- Skips execution of any unsupported Design Compiler commands and issues a warning message

  To see a list of unsupported Design Compiler commands, enter the `help` command at the DC Explorer shell prompt.

DC Explorer is different from Design Compiler in the following ways:

- The DC Explorer command-line interface (de_shell) operates in topographical mode by default, so specifying the `-topographical_mode` option is not required.
- If you attempt to run the `compile_ultra` command, DC Explorer invokes the `compile_exploration` command instead.

  If you use an unsupported option in the `compile_ultra` or `compile` command, DC Explorer issues a warning and skips the unsupported option. If you use the `-incremental` option, DC Explorer issues an error message and skips execution of the command entirely.
- DC Explorer ignores or translates unsupported Design Compiler commands and proceeds with script execution.
  - Unsupported commands causing a minimum impact on the design or QoR are ignored with a DESH-009 warning message. For example,

    ```
    Warning: Command 'set_fix_hold' is not supported in DC
    Explorer.
    The command is ignored.(DESH-009)
    ```
  - Unsupported commands causing a significant impact are ignored with a DESH-008 error message. For example,

    ```
    Error: Command 'insert_dft' is not supported in DC Explorer.
    (DESH-008)
    ```
  - Automated chip synthesis commands are not recognized, and DC Explorer issues a CMD-005 error message.
- DC Explorer supports test design rule checking for pre-DFT rule violations. Note the following limitations:
  - You can run the majority of DFT specification commands on a mapped design, but not on an unmapped design.
  - Unsupported commands are ignored with a DESH-009 warning or DESH-008 error message depending on the impact of the command.
  - The tool does not support post-DFT DRC, that is, running the `dft_drc` command on a stitched design.

  For more information about the DFT flow, see the DFT Compiler documentation.

Script Compatibility With Design Compiler

You should follow these guidelines when running Design Compiler scripts in DC Explorer:

- If your script checks for dc_shell as the program name and prompt, you need to change the prompt from de_shell to dc_shell. To do this, set the `de_rename_shell_name_to_dc_shell` variable to `true`. This sets the `synopsys_program_name` environment variable (a read-only variable) to dc_shell.
- To make a script that executes certain commands in DC Explorer and not in Design Compiler, use the following structure:

  For example,

  ```
  if {[shell_is_in_exploration_mode]} {do_DC_Explorer_setup}
  ```

- To optimize leakage power, replace the `set_max_leakage_power` command with the `set_multi_vth_constraint` command.
- Replace multiple `compile_ultra` commands with one `compile_exploration` command.

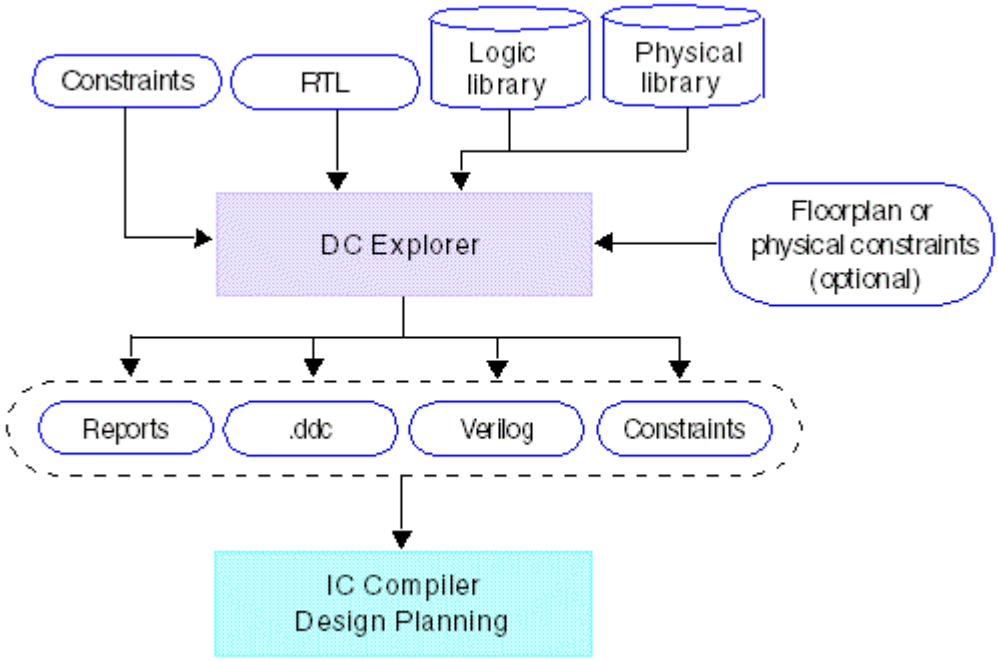
Floorplan Exploration in IC Compiler

By default, DC Explorer uses topographical technology, which enables you to accurately predict post-layout timing, area, and power during RTL synthesis without the need for wire load model-based timing approximations. It uses Synopsys' placement and optimization technologies to drive accurate timing prediction within synthesis, ensuring better correlation to the final physical design. It also results in a better starting point for physical implementation.

You can use DC Explorer to create a database for the floorplan exploration flow in IC Compiler. The supported database is a mapped netlist in .ddc or ASCII Verilog format. Most occurrences of the incomplete or mismatched data in the netlist are resolved except the ones between the logic libraries and physical libraries. You must resolve these inconsistencies before you proceed to floorplan exploration in IC Compiler.

This figure shows an overview of the design flow from DC Explorer to design planning in IC Compiler.

Figure 1-2: Flow From DC Explorer to Floorplan Exploration in IC Compiler



See Also

- Performing Floorplan Exploration
- Using Floorplan Physical Constraints
- *Design Vision User Guide*
- Design Vision Help

2 Working With DC Explorer

DC Explorer offers two interfaces for synthesis and timing analysis: the de_shell command-line interface (or shell) and the Design Vision graphical user interface (GUI). The de_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 DC Explorer environment, see

- Running DC Explorer
- Licensing
- Using Multicore Technology
- Running Commands in Parallel

Running DC Explorer

The following sections contain information about running DC Explorer using de_shell and the Design Vision GUI:

Starting DC Explorer

DC Explorer operates in the X windows environment on Linux. Before starting DC Explorer, ensure that the path to the bin directory is included in your $PATH variable. You start DC Explorer by entering the de_shell command in a Linux shell. For example,

```
% de_shell
```

By default, this command starts the tool in the command-line interface (de_shell).

You can also include other options on the command line when you start DC Explorer. For example, you can use

- `-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
- `-f` to execute a script file before displaying the initial de_shell prompt
- `-x` to include a de_shell statement that is executed at startup
- `-no_init` to specify that de_shell is not to execute 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 de_shell session.

- `-no_home_init` to specify that de_shell is not to execute any home .synopsys_dc.setup startup files
- `-no_local_init` to specify that de_shell is not to execute any local .synopsys_dc.setup startup files
- -gui to start de_shell with the GUI
- `-no_gui` to start de_shell without the GUI

For a detailed list of options, see the de_shell man page.

At startup, de_shell does the following tasks:

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

Note: If you enter the de_shell command as a path, you must specify the full absolute path, not a relative path. If you use a relative path (..\), DC Explorer cannot access the libraries that are located in the root directory.

The following example shows the correct way to indicate the Synopsys root containing the DC Explorer installation:
`/tools/synopsys/2012.06/bin/de_shell`

Entering de_shell Commands

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

- Enter individual commands interactively at the de_shell> prompt
- Enter individual commands interactively on the console command line in the Design Vision graphical user interface (GUI)
- Run one or more Tcl command scripts, which are text files containing de_shell commands

You enter commands in de_shell the same way you enter commands in a standard Linux shell. When the GUI is open, you can enter commands on the console command line and use the DC Explorer commands available through the menu interface.

To enter a command in de_shell,

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

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 de_shell> prompt button or press Return.

DC Explorer echoes the command output, including processing messages and any warnings or error messages, in both de_shell and the console log view.

To display the options used with a de_shell command, enter the command name and `-help` on the de_shell command line. For example, to see the options used with the `compile_exploration` command, enter

```
de_shell> compile_exploration -help
```

When entering a command, an option, or a file name, you can minimize your typing by pressing the Tab key when you have typed enough characters to specify a unique name; DC Explorer completes the remaining characters. If the characters you typed could be used for more than one name, DC Explorer 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 de_shell command line, and clicking the middle mouse button.

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 de_shell. To interrupt or terminate a command, press Ctrl-C.

Some commands and processes, such as `update_timing`, cannot be interrupted. To stop these commands or processes, you must terminate de_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, de_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.

Opening or Closing the GUI in de_shell

When you start the command-line interface, the de_shell> prompt appears in the Linux shell. After starting the command-line interface, you can use the Design Vision graphical user interface (GUI).

Design Vision operates in the X windows environment on Linux. Before opening the GUI, ensure your $DISPLAY environment variable is set to the name of your Linux system display.

If you start de_shell without the GUI, you can open the GUI by entering the `gui_start` command at the de_shell> prompt. For example,

```
de_shell> gui_start
```

You can also open the GUI when you start DC Explorer by specifying the `-gui` option with the `de_shell` command. You must have a Design Vision license to use the GUI from a de_shell session.

You can open or close the GUI without exiting DC Explorer at any time during the session. For example, if you need to save system resources, you can close the GUI and leave DC Explorer running as a command-line interface.

To close the GUI without exiting Design Vision, do either of the following:

- Choose File > Close GUI
- ```
  de_shell> gui_stop
  ```

When you open the GUI, it reads a 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. For more information about the .synopsys_dv_gui.tcl files, see the *Design Vision User Guide*.

In addition to reading the setup files, the GUI 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.

You can set preferences that control how text appears in GUI windows and whether commands for selection or interactive operations appear in the session log. You can also set various global, schematic view, and layout view default controls.
  ```

See Also

- The “Setting GUI Preferences” topic in Design Vision Help

Getting Help on the Command Line

The following online information resources are available when you are using the DC Explorer tool:

- Command help, which is a list of options and arguments used with a specified de_shell command, displayed in the DC Explorer shell and in the console log view when the GUI is open
- Man pages displayed in the DC Explorer 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 when you use the GUI

To display a brief description of a de_shell command,

- Enter `help` followed by the command name:

```
de_shell> help command_name
```

To display a list of command options and arguments,

- Enter the command name followed by the `-help` option:

```
de_shell> command_name -help
```

To display a man page in de_shell and in the console log view if the GUI is open,

- Enter `man` followed by the command or variable name in de_shell:

```
de_shell> man command_or_variable_name
```

To display a man page in the GUI man page viewer, do either of the following:

- Enter `man` followed by the command or variable name on the console command line in the GUI:

```
de_shell> man command_or_variable_name
```

- Enter `gui_show_man_page` followed by the command or variable name in de_shell or on the console command line in the GUI:

```
de_shell> gui_show_man_page command_or_variable_name
```

To view man pages interactively in the GUI man page viewer,

1. Choose Help > Man Pages.

   The man page viewer opens.

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

   A list of man pages appears.

3. Select the man page to view.

Setup Files

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

DC Explorer reads the three .synopsys_dc.setup files from three directories in the following order:

1. The Synopsys root directory
2. Your home directory
3. The current working directory

   This is the directory from which you invoke DC Explorer.

Table 2-1 describes the function of the three setup files.

Table 2-1: Setup Files

File	Location	Function
System-wide .synopsys_ dc.setup file	Synopsys root directory ($SYNOPSYS/admin/setup)	This file contains system variables defined by Synopsys and general DC Explorer setup information for all users at your site. Only the system administrator can modify this file. Note: $SYNOPSYS is the path to the DC Explorer installation directory.
User-defined .synopsys_ dc.setup file	Your home directory	This file contains variables that define your preferences for the DC Explorer working environment. The variables in this file override the corresponding variables in the system-wide setup file.

Design-specific .synopsys_ dc.setup file	Working directory from which you started DC Explorer	This file contains project- or design-specific variables that affect the optimizations of all designs in this directory. To use the file, you must invoke DC Explorer from this directory. Variables defined in this file override the corresponding variables in the user-defined and system-wide setup files.

Example 2-1 shows an example of the .synopsys_dc.setup file.

Example 2-1: .synopsys_dc.setup File

```
# Define the target library, symbol library,
# and link libraries
set_app_var target_library lsi_10k.db
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"
```

Finding Session Information in the Log Files

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

- Command Log Files
- Compile Log Files
- Filename Log Files

Command Log Files

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

You can change the name of the command.log file by using the `sh_command_log_file` variable in the .synopsys_dc.setup file. You should make any changes to these variables before you start DC Explorer. If your user-defined or project-specific .synopsys_dc.setup file does not contain this variable, DC Explorer automatically creates the command.log file.

Each DC Explorer session overwrites the command log file. To save a command log file, move it 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, DC Explorer generates an ASCII log, an HTML log file, and a .DE_log_snapshot_date_process_id directory:

- ASCII log

  The log displays the output, such as the commands processed and error messages, of each DC Explorer run on the screen for quick viewing and debugging. A summary at the end of the log shows how many messages are redirected to each specific log file under the .DE_log_snapshot_date_process_id directory.

- HTML log file

  The file resides under the current working directory. It contains the complete content of the ASCII log and all redirected messages 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 click the message ID link to display the message. By default, the file name is default.html. To change the file name, set the `de_log_html_filename` variable. For example,

  ```
  de_shell> set_app_var de_log_html_filename test1.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

  By default, the tool redirects noncritical warning and information messages from the ASCII log file to the HTML log file. To disable the redirection, set the `de_log_redirect_enable` variable to `false`.

- .DE_log_snapshot_date_process_id directory

  The directory stores information and warning messages that are redirected from the ASCII log. The error messages are displayed in the ASCII log and saved in the HTML log file. This directory contains the following log files:

  - read_design_log
  - read_library_log
  - timing_message_log
  - power_message_log

- ungroup_hierarchies_log
- register_removal_log
- optimization_message_log

Filename Log Files

DC Explorer writes names of the files that it has read to the filename log file in the directory from which you invoked de_shell. You can use the filename log file to identify data files needed to reproduce an error if DC Explorer terminates abnormally. To specify the name of the filename log file, set the `filename_log_file` variable in the .synopsys_dc.setup file.

Running Tcl Scripts

You can use Tcl scripts to accomplish routine, repetitive, or complex tasks. You can create a command script file by placing a sequence of de_shell commands in a text file. Any de_shell command can be executed within a script file. To run a script file from the de_shell command line, enter

```
de_shell> source file_name
```

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

```
# This is a comment
```

To run a script in the GUI, use the File > Execute Scripts dialog box.

See Also

- *Using Tcl With Synopsys Tools*

Exiting DC Explorer

You can exit DC Explorer at any time and return to the operating system. By default, de_shell saves the session information in the command.log file. However, if you change the name of the `sh_command_log_file` file after you start DC Explorer, session information might be lost. Also, de_shell does not automatically save the designs loaded in memory. If you want to save these designs before exiting, use the `write_file` command. For example,

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

To exit de_shell, do one of the following:

- Enter `quit`.
- Enter `exit`.
- Press Ctrl-d, if you are running DC Explorer in interactive mode and the tool is busy.

To exit DC Explorer in the Design Vision GUI, choose File > Exit.

Licensing

Specific license requirements apply to each of these design flows:

- DC Explorer
  - RTL-Exploration license
  - DC-Explorer-Shell license
- DC Explorer floorplan exploration
  - One set of DC Explorer licenses, one DC-Extension license, and one ICC-DP license

    or
  - Two sets of DC Explorer licenses and two DC-Extension licenses
- Pre-DFT design rule checking
  - DFT Compiler license

The Design Vision (GUI) and DC-Extension licenses are automatically checked out as needed.

Note: When using multicore processing, you need one license for every two cores. To remove additional licenses checked out during multicore processing, use the `remove_license` command.

You can view a list of the licenses you are currently using and a list of all licenses that are currently checked out. You can also check out additional licenses, queue licenses that are not currently available, and release licenses you no longer need. To determine what licenses are in use and how to obtain and release licenses, see

- Listing the Licenses in Use
- Obtaining Licenses
- Releasing Licenses
- Enabling License Queuing

Listing the Licenses in Use

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

```
de_shell> list_licenses
Licenses in use:
          DC-Explorer-Shell
          Design-Vision
```

```
          RTL-Exploration
1
```

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

```
de_shell> license_users
jack@eng1 RTL-Exploration
john@eng2 RTL-Exploration, Design-Vision
2 users listed.
1
```

Obtaining Licenses

When you invoke DC Explorer, 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.

You can obtain licenses in de_shell or in the GUI. To start the GUI from a de_shell session, you must have a Design Vision license.

- Using de_shell

  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 de_shell.

  For example, the following command checks out one license for the HDL Compiler feature:

  ```
  de_shell> get_license HDL-Compiler
  ```

  For multicore processing, where multiple licenses might be required, you can specify the total number of licenses needed by using the `-quantity` option with the `get_license` command. If you have already checked out licenses, the command acquires only the additional licenses needed to bring the total to the specified quantity.

- Using the GUI

  - To display the current license information, 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 other licenses you can use.

  - To check out an additional license, choose File > Licenses, click the license name in the Available Licenses list, and 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.

See Also

- Listing the Licenses in Use
- Releasing Licenses

Releasing Licenses

You can release licenses in de_shell or in the GUI. To start the GUI from a de_shell session, you must have a Design Vision license.

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

```
de_shell> remove_license HDL-Compiler
```

You can specify the number of licenses to be retained for each feature after the command has completed by using the `-keep` option. If you do not specify this option, all licenses for every feature are released when you run the `remove_license` command.

To release a license in the GUI,

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

See Also

- Listing the Licenses in Use
- Obtaining Licenses

Enabling License Queuing

DC Explorer has a license queuing functionality that allows your application to wait for licenses to become available when all licenses are in use. To enable this functionality, set the `SNPSLMD_QUEUE` environment variable to `true`. The following message is displayed:

```
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 will wait forever.

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 for the first key license that you require.
- The SNPS_MAX_QUEUETIME variable specifies the maximum wait time for checking out subsequent licenses within the same de_shell process. You use this variable after you have successfully checked out the first license to start de_shell.

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)
```

Using Multicore Technology

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

Enabling Multicore Processing

To enable multicore processing in DC Explorer, use the set_host_options command. For example, to enable the tool to use six cores to run your processes, enter

```
de_shell> set_host_options -max_cores 6
```

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

When you enable multicore processing, the log file contains an information message similar to the following:

```
Information: Running optimization using a maximum of 6 cores. (OPT-1500)
```

For maximum efficiency, the -max_cores setting should be no larger than the actual number of available cores.

Reporting Runtime

To report the overall compile wall clock time, run the report_qor command, as shown in the following example. The command reports the wall clock time for running the compile_exploration command.

```
de_shell > report_qor
****************************************
Report : qor
Design : TEST_TOP
```

```
Version: G-2012.06
Date   : Mon Jun 11 02:02:17 2012
*************************************
...

  Hostname: machine
  Compile CPU Statistics
  ----------------------------------------
  ...
  ----------------------------------------
  Overall Compile Time:              631.32
  Overall Compile Wall Clock Time:   288.11
```

The reported wall clock time is similar to the time reported by the following script:

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

When you measure the runtime speedup using multicore optimization, use the wall clock time of the process. The CPU time is not the correct indicator for multicore runtime speedup.

See Also

- Reporting Quality of Results

Running Commands in Parallel

Executing checking or reporting commands serially in a script can consume a significant portion of the overall runtime. To improve the runtime and generate the same reports, you can run these commands in parallel by using parallel command execution.

- The parallel_execute Command
- License Requirements
- Multicore Processing
- Supported commands

The parallel_execute Command

To enable parallel command execution, list the checking or reporting commands to execute in parallel by using the `parallel_execute` command in your script. If any listed command needs a timing update that was not performed before the parallel command execution block, the listed command automatically invokes the `update_timing` command during parallel command execution.

For example, the following script updates the timing information and then executes the report_timing, report_qor, report_cell, and report_area commands in parallel using eight cores:

```
set_host_options -max_cores 8
update_timing
parallel_execute [list \
   "report_timing > $mylogfile" \
   "report_qor >> $mylogfile" \
   "report_cell" \
   "report_area"]
```

The following script uses variables, including $MAX, $NWORST, $cstr_log, and $qor_log, for the option arguments and log files in the command strings:

```
set MAX 10000; set_app_var NWORST 10
set rpt_const_options "-all_violators"
set cstr_log "rpt_cstr.log"
set qor_log "rpt_qor.log"
parallel_execute [list \
 "report_timing -max $MAX -nworst $NWORST > rpt_tim.log" \
 "report_constraints $rpt_const_options > $cstr_log" \
 "report_qor > $qor_log"]
```

License Requirements

The same license requirements for running commands serially apply to parallel command execution using multicore processing. By default, you need one set of DC Explorer licenses for every eight cores.

Multicore Processing

Parallel command execution can use up to eight cores. The maximum number of parallel commands executed at one time is determined by the -max_cores option of the set_host_options command. If you specify a number larger than eight, the tool issues a warning message and overrides it with the number of available cores. The tool blocks the de_shell until the longest-running command in the parallel execution list is completed.

The following example runs parallel command execution using three cores:

```
de_shell> parallel_execute [list \
"report_cell" "report_timing" "report_area"]
Information: Running parallel report using a maximum of 3 cores. (RPT-
100)
```

The following example specifies eight cores for parallel command execution:

```
de_shell> set_host_options -max_cores 8
```

Supported commands

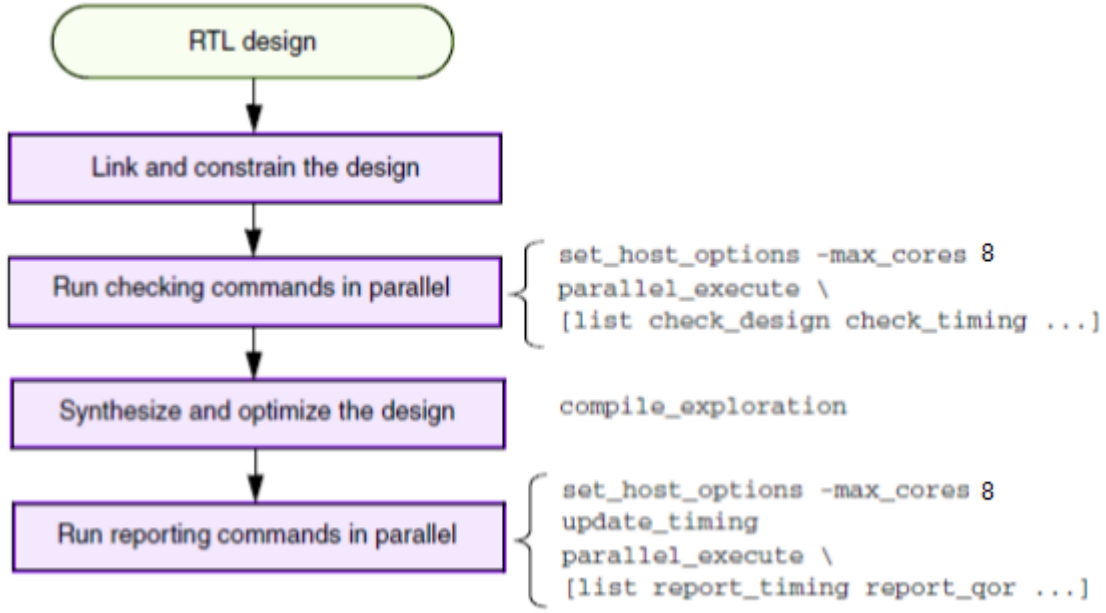
Use parallel command execution for reporting and checking commands only. To find out the supported commands, specify the `-list_all` option with the `parallel_execute` command. If you specify an unsupported command, the tool skips the command and issues a warning message similar to the following:

```
de_shell> parallel_execute [list report_libcell_subset]
Warning: 'report_libcell_subset' report command can't run in parallel_
execute mode. (RPT-106)
```

In the Design Flow

This figure shows how to run the checking and reporting commands in parallel to improve the runtime in the DC Explorer design flow.

Figure 2-1: Parallel Command Execution in the DC Explorer Design Flow



See Also

- Licensing
- Using Multicore Technology
- SolvNet article 1911827, "DC Explorer Video: Running Commands in Parallel"

3 Setting Up the Libraries

DC Explorer uses technology, symbol, and synthetic or DesignWare libraries to implement synthesis and to display synthesis results graphically. The following sections describe how to set up these libraries and carry out simple library commands so that DC Explorer uses the library data correctly:

Library Requirements

DC Explorer uses these libraries:

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

Logic Libraries

Logic libraries, which are maintained and distributed by semiconductor vendors, contain information about the characteristics and functions of each cell, such as cell names, pin names, area, delay arcs, and pin loading. Logic libraries 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.

DC Explorer supports logic libraries that use nonlinear delay models (NLDMs), Composite Current Source (CCS) models (either compact or noncompact), or both NLDM and CCS models. DC Explorer 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, DC Explorer uses the CCS models. During logic synthesis and preroute optimization, the tool might not use all the available CCS data to save runtime.

DC Explorer requires the logic libraries to be in .db format. In most cases, your semiconductor vendor provides you with .db-format libraries. If you are provided with only library source code, see the Library Compiler documentation for information about generating logic libraries in .db format. To set up logic libraries, see Specifying Logic Libraries.

DC Explorer uses logic libraries for the following purposes:

- Implementing the design function

  The logic libraries that DC Explorer 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 that are used to compile or translate a design become the local link libraries for the design. DC Explorer saves this information in the design's `local_link_library` attribute.

- Resolving cell references

  The logic libraries that DC Explorer 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 in the `link_library` variable.

- 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 consumed

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

DC Explorer uses the first logic library found in the `link_library` variable as the main library. If other libraries have measurement units different from the main library units, DC Explorer converts all units to those specified in the main library. DC Explorer obtains the following default values and settings from the main library:

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

The logic library setups contain target libraries and link libraries:

- Target libraries

  DC Explorer 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 DC Explorer 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.

- 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

references. To resolve references, DC Explorer uses the link libraries set by the following variables and attribute:

- The `link_library` system variable lists the libraries and design files that DC Explorer uses to resolve references.

  DC Explorer 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 DC Explorer searches loaded libraries in memory for the reference. For example, if you set the `link_library` variable to {"*" lsi_10k.db}, DC Explorer 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. DC Explorer 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 DC Explorer can continue the search for unresolved references after it searches the link libraries.

Symbol Libraries

Symbol libraries, which are maintained and distributed by semiconductor vendors, contain definitions of the graphic symbols that represent library cells in the design schematics.

When you generate the design schematic, DC Explorer performs a one-to-one mapping of cells in the netlist to cells in the symbol library. To view the design schematic, use Design Vision.

DesignWare Libraries

A DesignWare library is a collection of reusable intellectual property (IP) building blocks or components that are tightly integrated into the Synopsys synthesis environment. DesignWare components are available to implement many of the built-in HDL operators, such as +, -, *, <, >, <=, >=, and the operations defined by `if` and `case` statements.

You can develop additional DesignWare libraries 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 for the components.

Physical Libraries

If you want to use DC Explorer to generate an early netlist that can be used to begin physical exploration in IC Compiler design planning, you need to specify physical libraries in addition to logic 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 contain the Milkyway reference library and the Milkyway technology file:

- Milkyway reference library

  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).

- Milkyway technology file

  The Milkyway technology file (.tf), contains technology-specific information required to route a design. DC Explorer 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.

See Also

- Specifying Logic Libraries
- Specifying Physical Libraries
- Specifying a Library Search Path
- Linking Designs
- *Power Compiler User Guide*

Specifying Logic Libraries

Table 3-1 lists the variables that control library reading for each library type and the typical file name. You use these variables to specify logic libraries and DesignWare libraries.

Table 3-1: Library Variables

Library type	Variable	Default	File extension
Target library	`target_library`	`"your_library.db"`	.db
Link library	`link_library`	`"* your_library.db"`	.db
DesignWare library bottom	`synthetic_library`		.sldb

To set up access to the logic libraries, you must specify the target libraries and link libraries.

In the following example, the target library is the first link library. To simplify the link library definition, the example includes the `additional_link_lib_files` user-defined variable for libraries such as pads and macros.

```
de_shell> set_app_var target_library [list_of_standard_cell_libraries]
de_shell> set_app_var synthetic_library [sldb_files_for_designware]
de_shell> set additional_link_lib_files [additional_libraries]
de_shell> 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.

Specifying the DesignWare Libraries

You do not need to specify the standard synthetic library, standard.sldb, which implements the built-in HDL operators. DC Explorer 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 example, to use the DesignWare minPower components, include the dw_minpower.sldb library in the setting of the `synthetic_library` variable. No power optimization is performed.

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

See Also

- Library Requirements
- Specifying Minimum Timing Libraries
- Specifying a Library Search Path

Specifying a Library Search Path

You specify the library location by using either the complete path or only the file name. If you specify only the file name, DC Explorer 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 path to the installation directory. DC Explorer looks for the library files, starting from 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. DC Explorer uses the my_lib.db file found in the lib directory because it finds the lib directory first.

```
de_shell> set_app_var search_path "lib vhdl default"
```

You can use the `which` command to list the library files in the order as found by DC Explorer.

```
de_shell> which my_lib.db
/usr/lib/my_lib.db, /usr/vhdl/my_lib.db
```

See Also

- Library Requirements
- Specifying Logic Libraries

Specifying 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,

```
de_shell> set_app_var link_library "* maxlib.db"
de_shell> 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.

See Also

- Library Requirements

Specifying Physical Libraries

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. Define the power and ground nets. For example, set the following variables:

```
de_shell> set_app_var mw_power_net VDD
de_shell> set_app_var mw_ground_net VSS
de_shell> set_app_var mw_logic1_net VDD
de_shell> set_app_var mw_logic0_net VSS
de_shell> set_app_var mw_power_port VDD
de_shell> set_app_var mw_ground_port VSS
```

If you do not set these variables, power and ground connections are not made during execution of `write_milkyway`. Instead power and ground nets can get translated to signal nets.

2. Create the Milkyway design library by using the `create_mw_lib` command. For example,

```
de_shell> create_mw_lib -technology $mw_tech_file \
-mw_reference_library $mw_reference_library $mw_design_library_name
```

3. Open the Milkyway library that you created by using the `open_mw_lib` command. For example,

```
de_shell> open_mw_lib $mw_design_library_name
```

4. (Optional) Attach the TLUPlus files by using the `set_tlu_plus_files` command. For example,

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

5. In subsequent sessions, you 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,

```
de_shell> open_mw_lib $mw_design_library_name
de_shell> set_tlu_plus_files -max_tluplus$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`.

See Also

- Library Requirements
- Using a Milkyway Database
- Using TLUPlus or nxtgrd Files for RC Estimation

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
- Saving Libraries

Loading Libraries

DC Explorer 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. For example,

```
de_shell> read_file my_lib.db
de_shell> 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

DC Explorer 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. For example,

```
de_shell> 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
- 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 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

```
de_shell> 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

```
de_shell> set_disable_timing [get_pins my_lib/AND4/Z]
```

Excluding Cells From the Target Libraries

When DC Explorer 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 DC Explorer from using the INV_HD high-drive inverter, enter

```
de_shell> 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,

```
de_shell> remove_attribute MY_LIB/INV_HD dont_use
MY_LIB/INV_HD
```

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.

```
de_shell> check_library
```

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, specify the `-logic_vs_physical` option with the `report_check_library_options` command.

Removing Libraries From Memory

To remove libraries from de_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. Use the `list_libs` command to see the path for each library in memory.

Saving Libraries

The `write_lib` command saves (writes to disk) a compiled library in the Synopsys database or VHDL format.

See Also

- Library Requirements
- Specifying Logic Libraries
- Specifying a Library Search Path

Target Library Subsets

By default, the tool can select any library cell from the target library during optimization. To restrict the library cells to a subset of the target library or filter the target library cells for particular design blocks, use the `set_target_library_subset` command. For example, you can exclude specific double-height cells from the target library for some design blocks during optimization.

When you use the `set_target_library_subset` command, remember the following points:

- The subset restriction applies only to new cells that are created or mapped during optimization. It does not affect cells that are already mapped unless the tool modifies the cells during optimization.
- The command enforces the subset restriction to the specified designs and their subdesigns in the hierarchy, except those subdesigns on which a different subset restriction is set.
- A subset specified at a lower level overrides any subset specified at a higher level.
- A subdesign set by this command cannot be ungrouped by the `ungroup` command or the auto ungrouping capability of the tool.
- The subset restriction applies to the current design only.

The following table shows the `set_target_library_subset` options:

Table 3-2: The set_target_library_subset Options

To do this	Use this option
Specify the libraries that are a subset of the target library to optimize the specified design instances.	*library_list*
Specify a top-level design instance, hierarchical cell instances, or both as the design. If you do not specify either option, the default is the `-top` option.	`-top	-object_list` *cells*
Exclude library cells from the target library during optimization.	`-dont_use` *lib_cells*
Specify library cells in addition to the ones listed in the *library_list* list for the specified blocks for optimization.	`-use` *lib_cells*
Specify this option to restrict the scope of the command to only cells in clock paths.	`-clock_path`
Specify to use the library subset only for the specified design.	`-only_here` *lib_cells*
Specify the Milkyway reference library paths to associate with the identified instances.	`-milkyway_reflibs` *milkyway_reflib_paths*

To learn how to set target library subsets, see Setting Target Library Subset Restrictions.

Setting Target Library Subset Restrictions

To restrict optimization of a design block using a target library subset, perform the following tasks:

- Specifying Target Library Subsets
- Checking Target Library Subsets
- Reporting Target Library Subsets
- Removing Target Library Subsets

Specifying Target Library Subsets

Before you set target library subset restrictions, you must first set the `target_library` variable because you use the library cells listed in the variable to set the restrictions. To specify a target library subset or filter the target library cells for particular design blocks during optimization, use the `set_target_library_subset` command.

For example, the following commands set the target library list to include libraries lib1.db and lib2.db and restrict the library cells used in block u1 to the cells in library lib2.db:

```
de_shell> set_app_var target_library "lib1.db lib2.db"
de_shell> set_target_library_subset "lib2.db" \
 -object_list [get_cells u1]
```

The following example restricts the use of the INVX1, MUX1, and NOR2 library cells, so they are used on the clock path during optimization:

```
de_shell> set_target_library_subset clocklib.db \
 -clock_path -use [INVX1 MUX1 NOR2]
```

Checking Target Library Subsets

To check and display inconsistent settings introduced by target library subsets, use either the `check_target_library_subset` or `check_mv_design -target_library_subset` command. The commands check for inconsistent settings among the target library, target library subsets, and operating conditions.

The `check_mv_design -target_library_subset` command searches the design for any cells that do not follow the rules for the subset. The tool reports the cells found as warnings because they might have already existed in the design before the subset settings. You can run this command before and after optimization to identify issues and to check if you get the same results. You need a Power-Optimization license to run this command.

Reporting Target Library Subsets

To find out which target library subsets have been defined for the specified designs, use the `report_target_library_subset` command.

Reports that are generated by reporting commands, such as `report_cell` and `report_timing`, show the `td` attribute attached to the cells that are specified by the `-dont_use` or `-only_here` option.

Removing Target Library Subsets

To remove a target library subset restriction from a design or a design instance, use the `remove_target_library_subset` or `reset_design` command.

See Also

- Target Library Subsets

Using TLUPlus or nxtgrd 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, you should use them for RC estimation. As an alternative to TLUPlus files, you can use nxtgrd files to align the RC estimation of DC Explorer and the signoff tool.

Using TLUPlus Files

TLUPlus files provide more accurate capacitance and resistance data, thereby improving correlation with back-end results.

Use the `set_tlu_plus_files` command to specify TLUPlus files. 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,

```
de_shell> 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 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 postroute extraction.

Using nxtgrd Files

The nxtgrd file contains the reference parasitics for the manufacturing process of the chip. Using an nxtgrd file obtained from a foundry is the most accurate way of analyzing the parasitics of a specific process technology. You can also create an nxtgrd file using the grdgenxo tool, which is a StarRC utility program. The grdgenxo tool generates an nxtgrd file from a file written in a process description language called the Interconnect Technology Format (ITF).

Note: Using nxtgrd files aligns the synthesis and signoff tools and no longer requires you to maintain a separate TLUPlus file. However, you must generate an nxtgrd file using the grdgenxo tool version N-2017.12 or later.

To specify an nxtgrd file for RC estimation, use the `set_tlu_plus_files` command. As shown in the following example, the command sets the NXTGRD_file_max.nxtgrd file for the maximum condition calculation of the resistance and capacitance:

```
de_shell> set_tlu_plus_files -max_tluplus NXTGRD_file_max.nxtgrd \
   -tech2itf_map design.map
```

When you use the `set_tlu_plus_files` command to read nxtgrd files for 5-nm and smaller technology nodes, you must have the DC-Synth-AdvGeo license. If the license is not available, the tool issues an error message.

Support for Black Boxes

DC Explorer 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, as described in Creating Quick Timing Models for Black Boxes.

- Physical black boxes

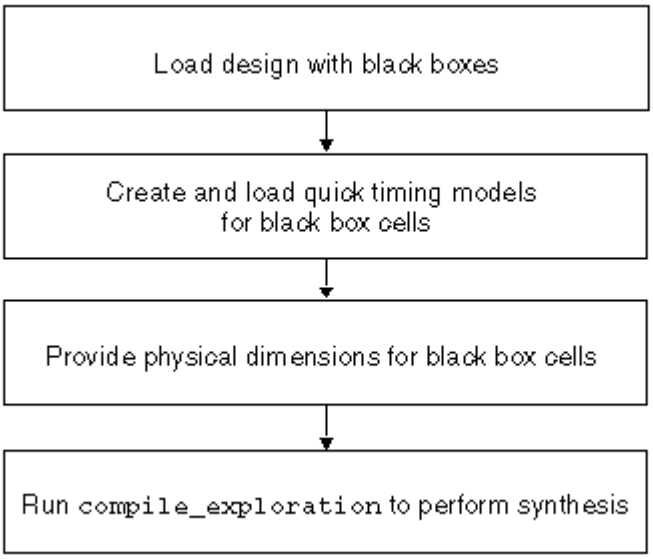
  These are cells that do not have physical representation. Examples include the following types of cells:

  - Empty hierarchy cells or black-boxed modules
  - Unlinked or unresolved cells

  You can define the physical dimensions for physical black box cells. If the physical representation of a cell is not available in your physical libraries, DC Explorer defines it automatically, as described in Defining Physical Dimensions.

Figure 3-1 shows the flow for defining the timing and physical dimensions for black box cells and synthesizing the design.

Figure 3-1: 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.

See Also

- Commands for Defining Timing in Black Boxes
- Automatic Creation of Physical Library Cells
- Defining Physical Dimensions
- Creating Quick Timing Models for Black Boxes
- Design Vision Help

Automatic Creation of Physical Library Cells

If the physical representation of a cell is not available with in your physical libraries, DC Explorer defines it automatically. The tool can create physical library cells for the following cells:

- 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)
```

Defining Physical Dimensions

DC Explorer 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:

  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:

  ```
  de_shell> estimate_fp_black_boxes \
  -sm_size {100 100} -sm_util 0.7 \
  [get_cells "is_physical_black_box==true" alu1]
  ```

- To create a rectilinear black box, use the `-polygon` option:

  The following example estimates a black box named alu1 and specifies it as a rectilinear soft macro:

  ```
  de_shell> 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]
  ```

- To create a hard macro black box, use the `-hard_macros` option:

  In the following example, the size of the black box named U1 is estimated from the size of the hard macro ram16x128:

  ```
  de_shell> estimate_fp_black_boxes -hard_macros ram16x128U1
  ```

Determining the Gate Equivalent Area

DC Explorer 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:

```
de_shell> 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:

```
de_shell> set_fp_base_gate -area 10
```

  In this example, the base gate area is set to 10 square microns.

Creating Quick Timing Models for Black Boxes

DC Explorer 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. DC Explorer 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, follow these steps:

1. Create a new model by using the `create_qtm_model` command.

```
de_shell> create_qtm_model BB
```

   BB is the model name.

2. Specify the logic information, such as the name of the logic library, the maximum transition time, the maximum capacitance, and the wire load information.

```
de_shell> set_qtm_technology -library library_name
de_shell> set_qtm_technology -max_transition trans_value
de_shell> set_qtm_technology -max_capacitance cap_value
```

3. Specify global parameters, such as setup and hold characteristics.

```
de_shell> set_qtm_global_parameter -param setup -value setup_value
de_shell> set_qtm_global_parameter -param hold -value hold_value
de_shell> set_qtm_global_parameter -param clk_to_output \
-value cto_value
```

4. Specify the ports by using the `create_qtm_port` command.

```
de_shell> create_qtm_port -type input A
de_shell> create_qtm_port -type input B
de_shell> create_qtm_port -type output OP
```

5. Specify delay arcs by using the `create_qtm_delay_arc` command.

```
de_shell> create_qtm_delay_arc -name A_OP_R -from A \
-from_edge rise -to OP -value 0.10 -to_edge rise
de_shell> create_qtm_delay_arc -name A_OP_F -from A \
-from_edge fall -to OP -value 0.11 -to_edge fall
de_shell> create_qtm_delay_arc -name B_OP_R -from B \
-from_edge rise -to OP -value 0.15 -to_edge rise
de_shell> 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.

```
de_shell> report_qtm_model
```

7. Save the quick timing model using the `save_qtm_model` command:

```
de_shell> save_qtm_model
```

8. Write out the .db file for the quick timing model using the `write_qtm_model` command:

```
de_shell> write_qtm_model -out_dir QTM
```

9. Add the .db file to the `link_library` variable to load the quick timing model:

```
de_shell> lappend link_library "QTM/BB.db"
```

See Also

- Commands for Defining Timing in Black Boxes

Commands for Defining Timing in Black Boxes

Table 3-3 lists the quick timing model commands you can use to define timing for black box cells.

Table 3-3: 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`

See Also

- Creating Quick Timing Models for Black Boxes

Identifying Black Boxes

DC Explorer sets the following attributes on black boxes:

- `is_black_box`

  When you specify the `is_black_box` attribute with the `get_cells` command, DC Explorer identifies functionally unknown black boxes:

  ```
  de_shell> get_cells -hierarchical-filter "is_black_box==true"
  ```

- `is_logical_black_box`

  When you specify the `is_logical_black_box` attribute with the `get_cells` command, DC Explorer identifies the logical black boxes in the design:

  ```
  de_shell> get_cells -hierarchical-filter "is_logical_black_
  box==true"
  ```

- `is_physical_black_box`

  When you specify the `is_physical_black_box` attribute with the `get_cells` command, DC Explorer identifies the physical black boxes in the design:

  ```
  de_shell> get_cells -hierarchical \
  -filter "is_physical_black_box==true"
  ```

4 Tolerance for Incomplete or Mismatched Data

DC Explorer can continue the link process despite incomplete or mismatched data between the RTL netlist and libraries, between logic libraries and physical libraries, or between top-level RTL instantiations and interface pin definitions of RTL subblocks. The tool provides a summary list and detailed reports of linking error and warning messages issued for these data inconsistencies. When the tool encounters infeasible paths, it infers path assumptions and continues to optimize other critical paths.

Tolerance for incomplete or mismatched data is enabled by default. You can change some of the settings that control specific tolerance, such as bit and bit-blasted naming styles, before DC Explorer reads in a design and links the design.

To learn more about tolerance for incomplete or mismatched data, see

- Tolerance Categories
- Default Tolerance Setup
- Adjusting the Tolerance Setting for Bus Pin Naming Styles
- Resolving Bus and Bit-Blasted Naming Styles
- Allowing Pin Name Synonyms
- Allowing Unconnected Interface Pins
- Including Anchor Cells in Netlists
- Creating Reports and Missing Constraints

Tolerance Categories

Table 4-1 lists the categories of tolerance for incomplete or mismatched data that DC Explorer supports. A check mark (X) indicates that the tolerance category is applicable and supported.

Table 4-1: Tolerance Categories for Incomplete or Mismatched Data

Tolerance category	Library definition versus instantiation	Top-level RTL versus RTL blocks	Logic library versus physical library
Mismatched bus pin naming styles	X	X	X
Bus versus bit-blasted	X	X	X
Case mismatches between pin names	X	X	X
Missing cells in logic libraries	X		
Missing cells in physical libraries			X
Missing pins in logic libraries	X		X
Missing pins in physical libraries			X
Missing pins in RTL modules		X	
Missing RTL modules		X	
Pin direction mismatches			X
Pin name synonyms (pin mapping)	X	X	X
Port width mismatches	X	X	
Unconnected interface pins	X	X	

See Also

- Default Tolerance Setup
- Adjusting the Tolerance Setting for Bus Pin Naming Styles

- Resolving Bus and Bit-Blasted Naming Styles
- Allowing Pin Name Synonyms
- Allowing Unconnected Interface Pins
- Reporting Incomplete or Mismatched Data

Limitations

DC Explorer supports all the tolerance categories for mismatches in Verilog files, but not for mismatches between subblock instantiations and top-level component instantiations in VHDL files. For example, use the tolerance category of missing pins in RTL modules for the following VHDL code. The code shows a mismatch between the two instantiations, SUB1 and TOP, where the SUB1 instantiation is missing the MSGPIN pin. The tool issues an error message for this type of mismatch when analyzing the design.

```
entity SUB1 is
  port (AA, BB: in bit; CC: out bit);
end entity SUB1;

architecture RTL of SUB1 is
begin
  CC <= AA and BB;
end architecture RTL;

library WORK;
use WORK.all;

entity TOP is
  port (a, b, e: in bit; c: out bit);
end entity TOP;

architecture RTL of TOP is
begin
  U1 : entity WORK.SUB1(RTL)
  port map (AA=>a, BB =>b, MSGPIN=>e, CC=>c);
end architecture RTL;
```

See Also

- Tolerance Categories

Default Tolerance Setup

By default, DC Explorer is set up for the following tolerance categories, and you cannot configure the settings except the tolerance for bus pin naming styles. To learn more about the default behaviors of DC Explorer, see each description of tolerance for

- Bus Pin Naming Styles
- Case Mismatches Between Pin Names
- Missing Cells in Logic Libraries
- Missing Cells in Physical Libraries
- Missing Pins in Logic Libraries
- Missing Pins in FRAM Views
- Pin Direction Mismatches Between Logic and Physical Libraries
- Port Width Mismatches

Bus Pin Naming Styles

By default, DC Explorer can match RTL blocks to their corresponding blocks in the logic library based on the naming conventions defined by the `bit_blasted_bus_linking_naming_styles` variable. To adjust the default setting of this variable for different bus naming styles, see Adjusting the Tolerance Setting for Bus Pin Naming Styles. Table 4-2 shows matching examples based on the default setting of this variable.

Table 4-2: Default Bus Name Matching Examples

RTL	Logic library file
OPRNDA_0_	OPRNDA[0]
OPRNDA_1_	OPRNDA[1]
OPRNDB(0)	OPRNDB_0_
OPRNDB(1)	OPRNDB_1_
OPRNDC[0]	OPRNDC(0)
OPRNDC[1]	OPRNDC(1)

See Also

- Adjusting the Tolerance Setting for Bus Pin Naming Styles

Case Mismatches Between Pin Names

By default, DC Explorer allows uppercase or lowercase mismatches of pin names between

- Top-level instantiations and RTL blocks
- Logic libraries and RTL instantiations

- Logic libraries and physical libraries (FRAM views)

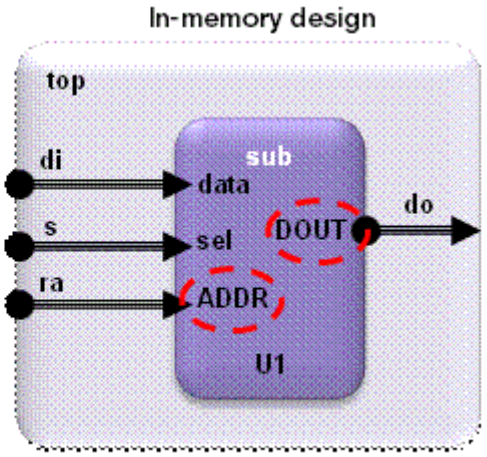
Figure 4-1 shows an example of case mismatches in pin names between top-level instantiations and an RTL block. The design loaded in memory shows that the mismatches have been resolved.

Figure 4-1: Example of Case Mismatches Between Pin Names

RTL

```
module sub (data, sel, ADDR, DOUT);
  input [3:0] data, sel, ADDR;
  output [3:0] DOUT;
....

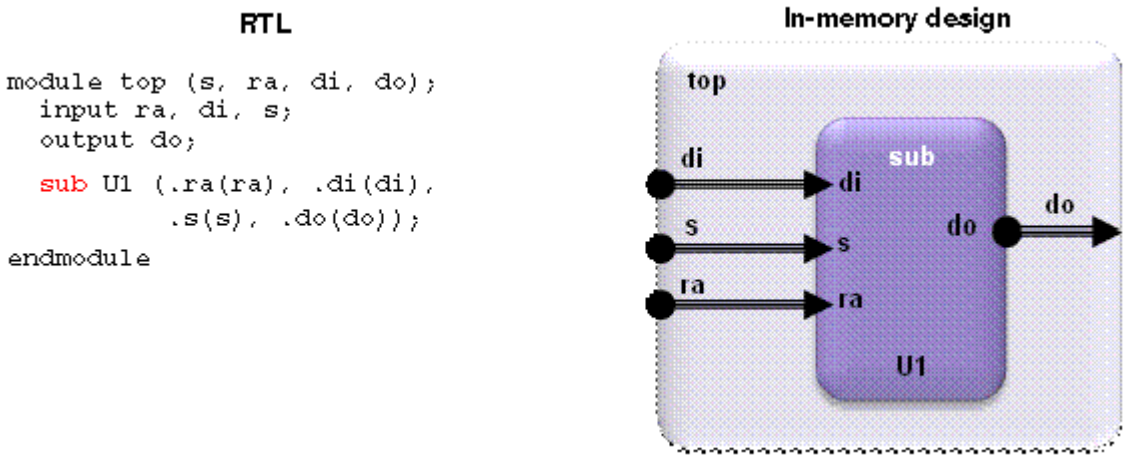
endmodule
module top (s, ra, di, do);
  input [3:0] di, s, ra;
  output [3:0] do;
  sub U1 (.data(di), .sel(s),
          .addr(ra), .DOut(do));
endmodule
```



Missing Cells in Logic Libraries

During the link process, DC Explorer can link cells with missing references in logic libraries, as shown in Figure 4-2. In such cases, DC Explorer issues no error messages, but the resolved library cells, macros, or modules contain no timing arcs.

Figure 4-2: Tolerance for Missing Cells in Logic Libraries



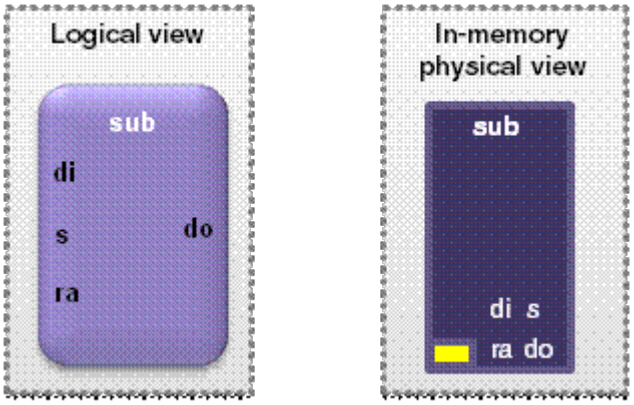
Missing Cells in Physical Libraries

When a physical library is a missing cell, DC Explorer performs the following tasks to resolve the mismatch:

- Creates the FRAM view for the missing cell in the local Milkyway design library, using a one-unit-tile bounding box
- Derives the pin directions of the missing cell from the logic library
- Adds physical pins at the origin of the FRAM view

For large missing macros in physical libraries, you should follow the steps described in the black box flow to resolve the mismatch. For more information about the black box flow, see the IC Compiler Design Planning User Guide. Figure 4-3 shows the FRAM view created for the sub logic cell.

Figure 4-3: Example of Missing Cells in Physical Libraries



See Also

- IC Compiler Design Planning User Guide

Missing Pins in Logic Libraries

When a logic library cell or RTL reference has missing pins, DC Explore allows the mismatches by doing the following:

- Creates an anchor cell to preserve the connection and to prevent the removal of each missing pin
- Ignores the timing constraints on the missing pins
- Automatically removes the anchor cells and restores the missing pin connections in the ASCII netlist when you write out the design

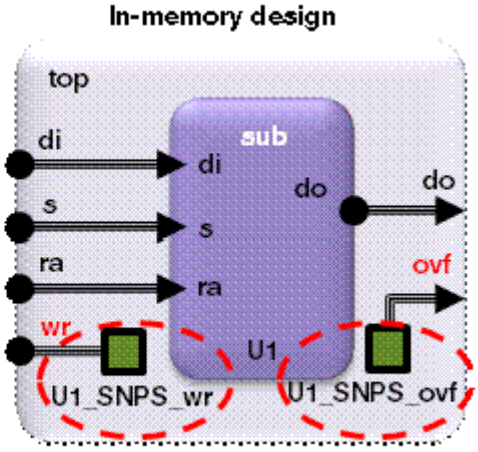
As shown in Figure 4-4, two anchor cells are created for the wr and ovf missing pins.

Figure 4-4: Example of Missing Pins in Logic Libraries

RTL

```
module sub (ra, di, s, do);
  input [3:0] ra, di, s;
  output [3:0] do;
....
endmodule
module top (s, ra, di, wr, do, ovf);
  input [3:0] ra, di, s;
  input wr;
  output [3:0] do;
  output ovf;
  sub U1 (.ra(ra), .di(di), .s(s),
          .wr(wr), .do(do), .ovf(ovf));
endmodule
```

In-memory design



See Also

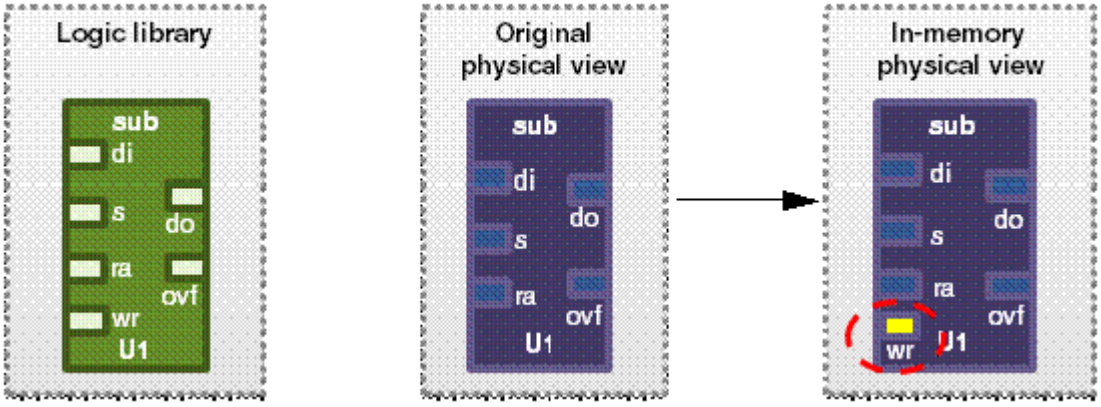
- Including Anchor Cells in Netlists

Missing Pins in FRAM Views

When a logic library cell has more pins than the FRAM view, DC Explorer creates a new physical pin for each additional pin. The new physical pin is added to the lower-left corner of the cell bounding box and placed on the lowest allowed routing layer.

As shown in Figure 4-5, a dummy physical pin is created and placed at the lower-left corner of the sub block for the missing wr pin in the FRAM view.

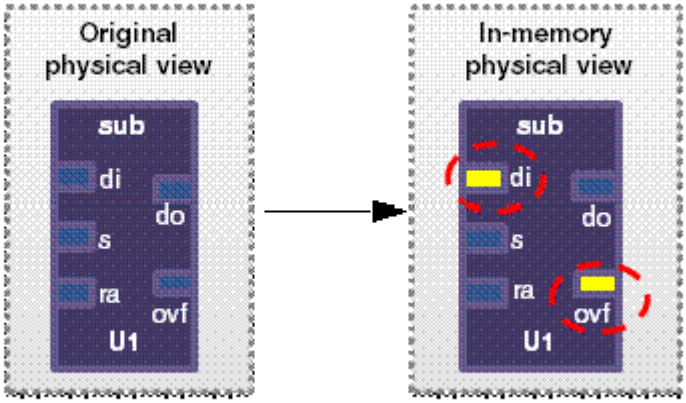
Figure 4-5: Example of a Missing Pin in the FRAM View



Pin Direction Mismatches Between Logic and Physical Libraries

DC Explorer allows port direction mismatches. For example, when a pin direction specified in the logic library is different from that in the FRAM view, DC Explorer creates a physical link for the logic library cell by using the direction specified in the logic library. Figure 4-6 shows how DC Explorer automatically resolves the di and ovf pin direction mismatches.

Figure 4-6: Example of Pin Direction Mismatches



Port Width Mismatches

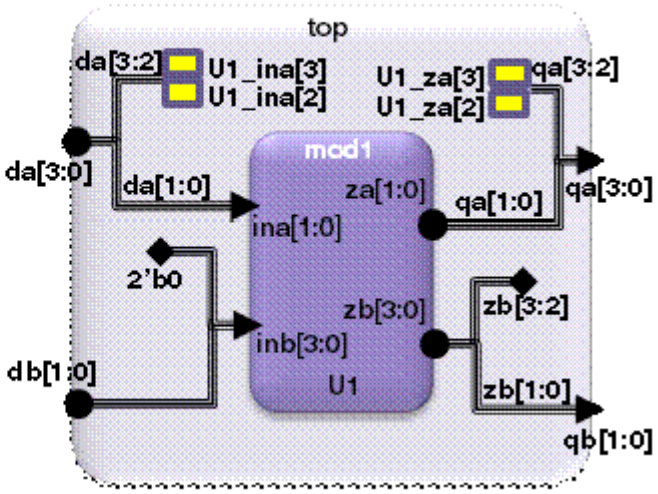
DC Explorer allows port width mismatches. For example, when a top-level design instantiates an RTL block that contains mismatched port widths, DC Explorer truncates or zero-extends the bus and adds anchor cells to preserve the extra signal connections at the top level, as shown in Figure 4-7.

Figure 4-7: Example of Port Width Mismatches

Mismatched RTL

```
module mod1 (ina, inb, za, zb);
  input [1:0] ina;
  input [3:0] inb;
  output [1:0] za;
  output [3:0] zb;
. . . .
endmodule
module top (da, db, qa, qb);
  input [3:0] da;
  input [1:0] db;
  output [3:0] qa;
  output [1:0] qb;
  mod1 U1 (.ina(da), .inb(db),
           .za(qa), .zb(qb));
endmodule
```

In-memory design



Adjusting the Tolerance Setting for Bus Pin Naming Styles

DC Explorer can match certain bus pin naming styles by default. You can enable DC Explorer to match other bus pin naming styles by setting the `bit_blasted_bus_linking_naming_styles` variable. The default of this variable is `"%s\[%d\] %s(%d) %s_%d_"`. To allow DC Explorer to match the bus pin naming styles shown in Table 4-3, enter

```
de_shell> set_app_var enable_bit_blasted_bus_linking true
de_shell> set_app_var bit_blasted_bus_linking_naming_styles \
"%s_%d %s\[%d\] %s<%d>"
```

In this example, %s_%d represents a string ending with an underscore and a number. A name in this form matches any name consisting of the same string followed by the same number enclosed in braces (%s\[%d\]) or the same string followed by the same number in angle brackets (%s<%d>). A pair of names in any of the three specified forms are considered equivalent names for matching purposes. Table 4-3 shows some examples of matching names for this example.

Table 4-3: Matching Bus Pin Naming Styles Between an RTL and Logic Library File

RTL	Logic library file
OPRNDA_0	OPRNDA[0]
OPRNDA_1	OPRNDA[1]
OPRNDB<0>	OPRNDB_0
OPRNDB<1>	OPRNDB_1
OPRNDC<0>	OPRNDC[0]
OPRNDC<1>	OPRNDC[1]

See Also

- Bus Pin Naming Styles

Resolving Bus and Bit-Blasted Naming Styles

To enable DC Explorer to match different bus and bit-blasted naming styles,

1. Set the `enable_bit_blasted_bus_linking` variable to `true`, the default.
2. Set the `bit_blasted_bus_linking_naming_styles` variable to specific naming styles.

As shown in Figure 4-8, the \sel[1] and \sel[0] bit-blasted signals of the sub RTL block are mismatched. You can set the following variables to resolve the mismatches:

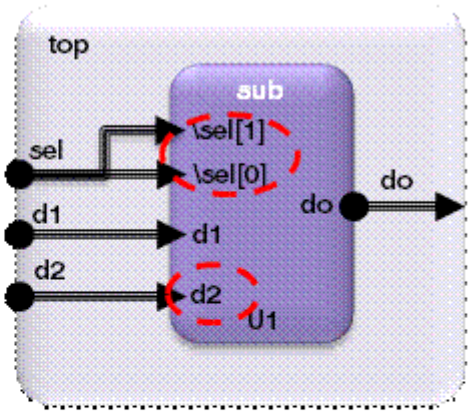
```
de_shell> set_app_var enable_bit_blasted_bus_linking true
de_shell> set_app_var bit_blasted_bus_linking_naming_styles \
"%s\[%d\] %s(%d) %s_%d_"
```

Figure 4-8: Example of Bus and Bus-Blasted Naming Styles

Mismatched RTL

```
module sub (do, d1, d2, \sel[1], \sel[0]);
  output do;
  input [3:0] d1, d2;
  input \sel[1], \sel[0];
....
endmodule
module top (do, d1, d2, sel);
  output do;
  input [3:0] d1, d2;
  input [1:0] sel;
  sub U1 (.sel(sel), .do(do), .d1(d1),
          .\d2[3](d2[3]), .\d2[2](d2[2]),
          .\d2[1](d2[1]), .\d2[0](d2[0]));
endmodule
```

Resolved in-memory design



Mapped Netlist

```
module sub (do, d1, d2, \sel[1], \sel[0]);
....
endmodule
module top (do, d1, d2, sel);
....
  sub U1 (.\sel[1](sel[1]), .\sel[0](sel[0]),
          .do(do), .d1(d1), .d2(d2));
endmodule
```

Using the same settings of the two variables in the previous example, DC Explorer can match multidimensional buses, such as the a bus and b bus of the sub1 RTL block shown in the following RTL example:

```
module sub1 (input \a[1][1], \a[1][0], \a[0][1],\a[0][0],
             input [1:0][1:0]b,
             output [1:0][1:0] c);

module top (input [1:0][1:0] ain,
            input [1:0][1:0] bin,
            output [1:0][1:0] cout);

sub1 U1 (.a(ain),
         .\b[1][1](bin[1][1]), .\b[1][0]),
         .\b[0][1](bin[0][1]), .\b[0][0]),
         .c(cout));
endmodule
```

Allowing Pin Name Synonyms

You can enable the pin mapping capability based on specified synonyms during the link process. The capability resolves mismatches for each of the following cases:

- Top-level instantiations and RTL blocks
- Logic libraries and RTL instantiations
- Logic libraries and physical libraries (FRAM view)

To enable pin name synonym matching,

1. Make sure that the `link_allow_pin_name_synonym` variable is set to `true` (the default).
2. Use the `set_pin_name_synonym` command to specify pin name synonyms for RTL, logic library, and physical library pin names.

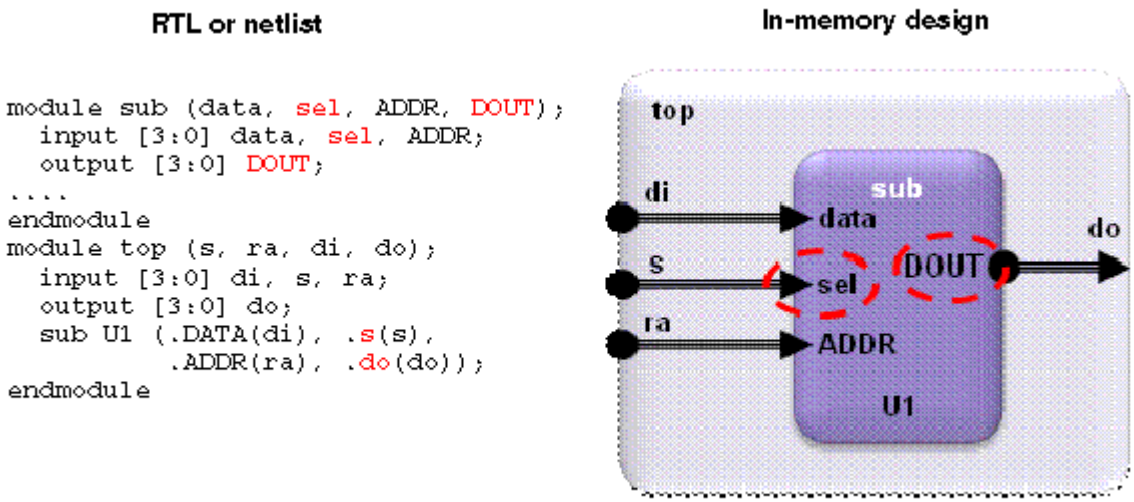
To remove pin name synonyms, use the `remove_pin_name_synonym` command. You should remove all pin name synonyms after the link process and then reapply them before loading the SDC file. To report pin name synonyms, use the `report_pin_name_synonym` command.

Example 1

The following example enables pin mapping between the top top-level instantiation and the sub RTL block. The `set_pin_name_synonym` command allows the s synonym for the sel pin and the do synonym for the DOUT pin.

```
de_shell> set_app_var link_allow_pin_name_synonym true
de_shell> set_pin_name_synonym s sel
de_shell> set_pin_name_synonym do DOUT
```

Figure 4-9: Pin Mapping Between Top-Level Instantiations and RTL Blocks



Example 2

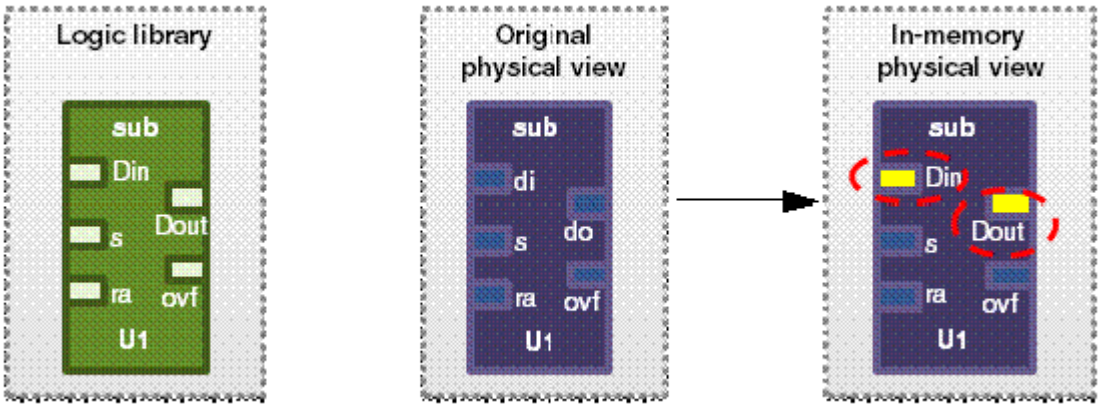
To enable pin mapping between the logic library and physical library, as shown in the following example, the `set_pin_name_synonym` command

1. Creates a FRAM view, which is saved in the Milkyway design library.
2. Automatically corrects the pin names of the FRAM view to match the logic library pin names based on the specified pin name synonyms.

The example uses the Dout synonym for the do pin and the Din synonym for the di pin.

```
de_shell> set link_allow_pin_name_synonym true
de_shell> set_pin_name_synonym Dout do
de_shell> set_pin_name_synonym Din di
```

Figure 4-10: Example of Pin Mapping Between Logic Library and Physical Library



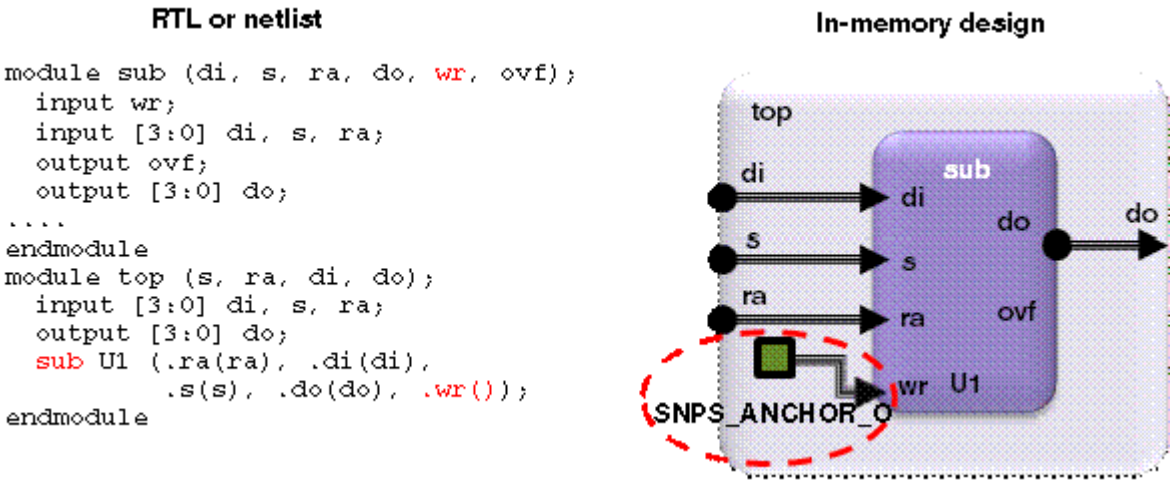
Allowing Unconnected Interface Pins

To prevent unconnected interface pins from

- Connecting to constant high or constant low
- Propagating constant values across unconnected interface pins

You can set the `link_preserve_dangling_pins` variable to `true`, changing it from its default of `false`. Setting this variable to `true` allows DC Explorer to add an anchor cell to each dangling interface pin. As shown in Figure 4-11, DC Explorer adds the SNPS_ANCHOR_O anchor cell to the unconnected wr signal.

Figure 4-11: Example of Unconnected Interface Pins



Including Anchor Cells in Netlists

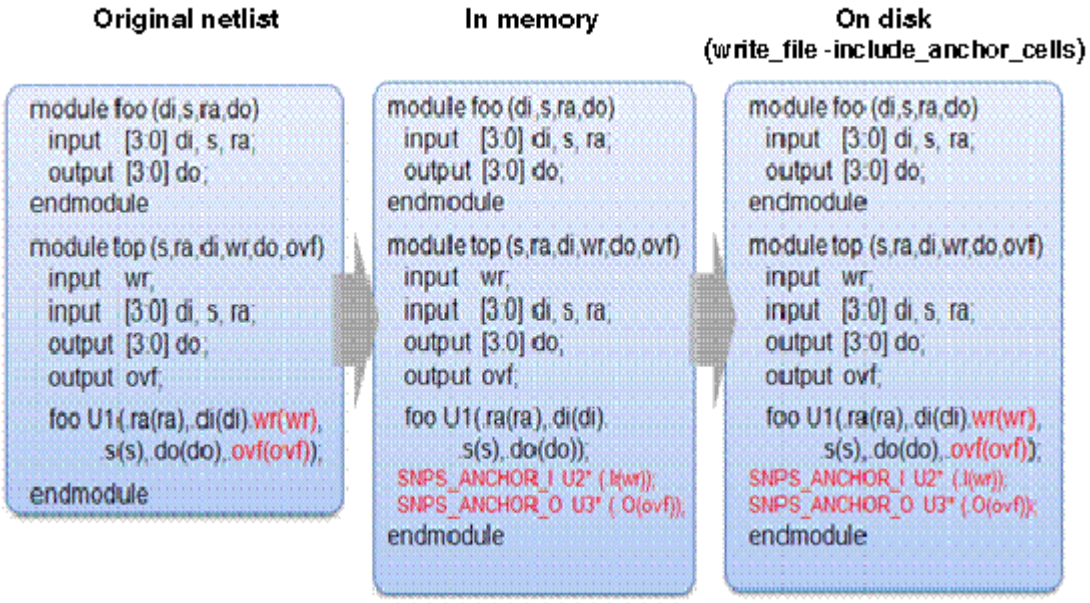
By default, DC Explorer creates an anchor cell to prevent the removal of each missing pin in logic libraries or RTL and to preserve the connection. When you save the design, the anchor cells are automatically removed from the netlist. To include the anchor cells in the netlist in Verilog format, use the `-include_anchor_cells` option with the `write_file` command.

For example, the following command saves the hierarchical design with anchor cells in Verilog format:

```
de_shell> write_file -hierarchy -format verilog \
-output mapped.vg -include_anchor_cells
```

Figure 4-12 shows how the netlist includes the anchor cells.

Figure 4-12: Anchor Cells Support in Netlist



Creating Reports and Missing Constraints

Your design might consist of incomplete or mismatched data, such as case mismatches between pin names, missing cells in logic libraries, missing RTL modules, and missing constraints. Incomplete or missing timing constraints can cause infeasible paths. To report incomplete data and infeasible paths and to create timing exceptions and missing constraints, see

- Reporting Incomplete or Mismatched Data
- Reporting Missing Constraints
- Reporting Infeasible Paths
- Creating Constraints Using Categorized Timing Reports
- Creating Exceptions for Infeasible Paths

See Also

- Tolerance Categories

Reporting Incomplete or Mismatched Data

To report incomplete or mismatched data, choose one of the following methods:

- Use the `link` command to show a list of warnings of mismatches found and resolved.

  For example,

```
Warning: Automatically reconstructed 'sel' bus with width of 2 from
{sel[0] ... sel[1]} bit-blasted bus pins in 't1' reference.
(LINK-912)

Warning: Automatically reconstructed 'd2' bus with width of 4 from
{ d2[0] d2[1] ... d2[3] } bit-blasted bus pins in 'U1' cell.
(LINK-912)

Warning: 'wr' pin on 'inst1' cell in 'top' design is missing from
the
'SliceInit' reference. (LINK-902)

Warning: Unable to connect 'wr' net to the 'wr' pin of 'inst1' cell
in 'top' design. (LINK-903) . . . .

Warning: Design mismatches were detected by the linker and resolved
to link the current design. You can use report_design_mismatch
and check_design commands explicitly to see warning messages about
the design mismatches. (DESH-017)
```

- Use the `check_design` command to show a summary of all design mismatches that the tool encounters and resolves. For example,

```
Design Mismatches                                 Logical Physical
------------------------------------------------------------------
Missing pin on logical library part (LINK-902)      1
Can't connect net to missing pin (LINK-903)         2
Missing physical pin (LINK-905)                               1
Presence of linker anchor cells (LINK-911)          2
Reconstructed bus (LINK-912)                        1
------------------------------------------------------------------
Total netlist elements                              6        1
```

- Use the `report_design_mismatch` command to show the details of design mismatches. For example,

```
****************************************
Report:      design mismatches
Design:      mydesign
Version:     H-2013.03
Date:        Fri Jan 25 17:58:57 2013
```

```
*************************************
Number of missing pins on logical library cells : 1

Design Object  Type    Mismatch
---------------------------------------------------------------
top    inst1   cell    'wr' pin on 'inst1' cell in 'top' design
                       is missing from the 'SliceInit' reference.

Number of unconnectable nets due to missing pins : 1

Design Object  Type    Mismatch
---------------------------------------------------------------
top    wr      net     Unable to connect 'wr' net to the
                       '*cell*6/wr' pin of 'inst1' cell in 'top'
                       design.

Number of linker anchor cells : 1

Design Object  Type    Mismatch
---------------------------------------------------------------
top    DCLINKER cell   Linker anchor cell inserted for missing
                       pin on 'inst1' cell.

Number of reconstructed bus : 1

Design Object  Type    Mismatch
---------------------------------------------------------------
top    U1      cell    Automatically reconstructed 'sel' bus
                       with width of 2 from { sel[0] ... sel[1]}
bit-blasted bus pins in 'my_design' reference.
```

See Also

- Reporting Missing Constraints
- Reporting Infeasible Paths
- Creating Constraints Using Categorized Timing Reports

Reporting Missing Constraints

To report missing constraints, such as missing clocks, I/O latencies, driving cells, and output loads in the current design, use the `report_missing_constraints` command. For example,

```
de_shell> report_missing_constraints
Information: Checking out the license 'DesignWare'. (SEC-104)
*********************************************
Report : Missing Constraints
Design : des_top
Version: H-2013.03
```

```
Date   : Fri Jan 25 17:58:57 2013
************************************************
Missing Clock
--------------------------------------------------------------------------------
        clk_hi_1
--------------------------------------------------------------------------------
Total Missing Clock Definition found: 2.

Missing Input Delay
---------------------------------------
        combi_hi[3]
        comb1_hi[2]
---------------------------------------
The total number of missing input delay found is 2.

Missing Output Delay
---------------------------------------
        out_filter_i1
---------------------------------------
The total number of missing output delay found is 1.
```

By default, DC Explorer uses zero I/O delays and selects inverters from the logic library for missing drivers and output loads. You can specify values for the missing constraints instead of letting the tool assume the constraint values. To create missing constraints in the current design, use the `create_missing_constraints` command with the appropriate options.

For example,

```
de_shell> create_missing_constraints -period 0.75 \
-input_delay 0.02 -output_delay 0.1 -output new1.sdc
```

See Also

- Reporting Infeasible Paths
- Creating Constraints Using Categorized Timing Reports
- Creating Exceptions for Infeasible Paths

Reporting Infeasible Paths

Infeasible timing paths can exist for the following reasons:

- Missing boundary conditions
- Unrealistic timing goals for input delays
- Unrealistic timing goals for output delays

During optimization, the `compile_exploration` command automatically detects infeasible paths in the design. To control the focus of optimization effort on other paths, the command

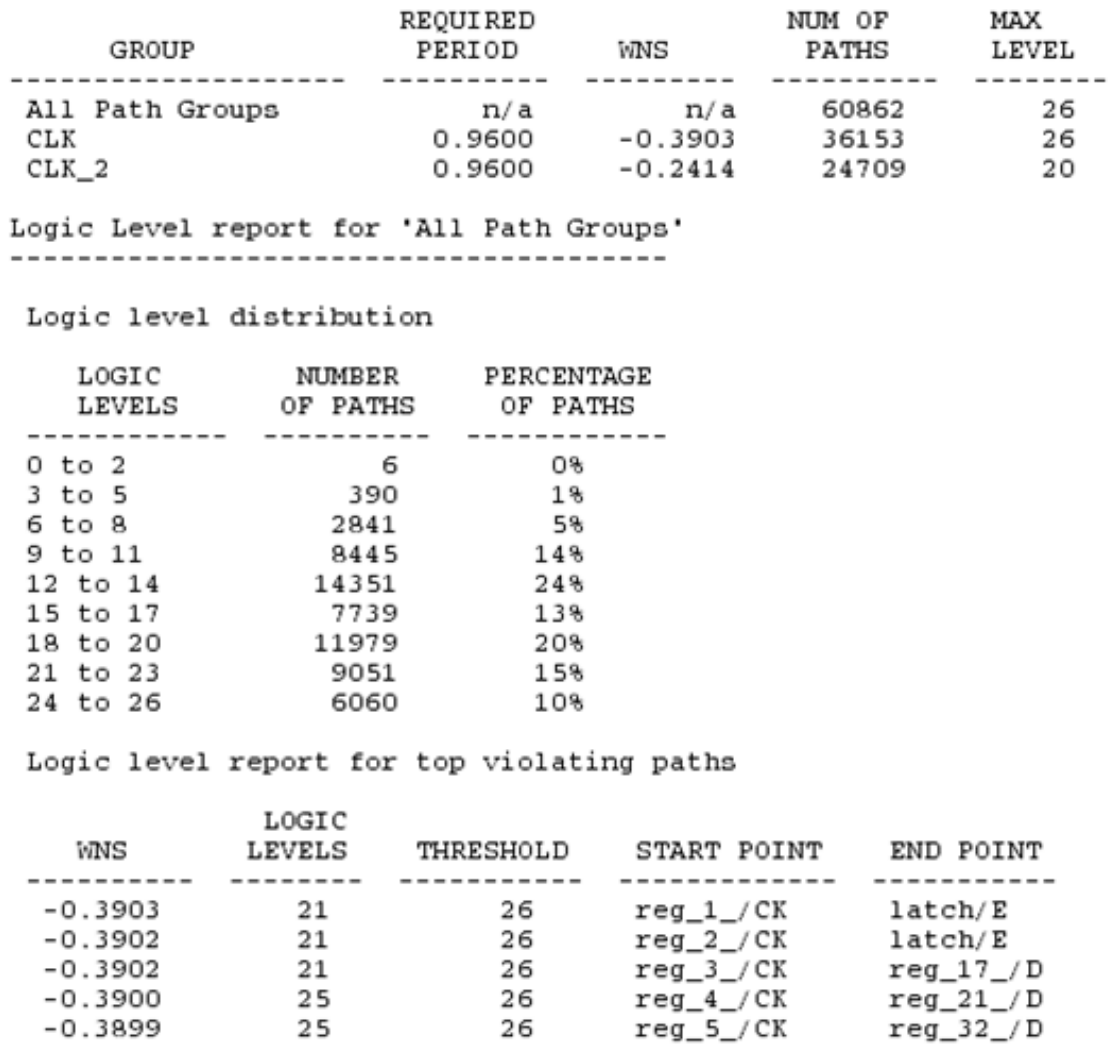
temporarily sets these infeasible paths as false paths. At the end of optimization, these false path settings are removed.

After running the `compile_exploration` command, you can report the infeasible paths that are temporarily set as false paths. To do this, specify the `-infeasible_paths` option with the `create_qor_snapshot` and `query_qor_snapshot` commands, and then generate a categorized timing report in HTML format, as shown in the following command sequence. To set the maximum number of infeasible paths to be reported, specify the `-max_paths` option with the `create_qor_snapshot` command.

```
de_shell> compile_exploration
de_shell> create_qor_snapshot -max_paths 30 -infeasible_paths -name snap1
de_shell> query_qor_snapshot -name snap1 -infeasible_paths \
-columns {path_group infeasible_paths startpoint endpoint wns zero_path}
\
-output_file snap1.htm
de_shell> sh firefox snap1.htm
```

Figure 4-13 shows the categorized timing report in HTML format generated by the command sequence. Each infeasible path detected during optimization is indicated with a YES under the Infeasible Path column.

Figure 4-13: Categorized Timing Report in HTML Format



GROUP	REQUIRED PERIOD	WNS	NUM OF PATHS	MAX LEVEL
All Path Groups	n/a	n/a	60862	26
CLK	0.9600	-0.3903	36153	26
CLK_2	0.9600	-0.2414	24709	20

Logic Level report for 'All Path Groups'

Logic level distribution

LOGIC LEVELS	NUMBER OF PATHS	PERCENTAGE OF PATHS
0 to 2	6	0%
3 to 5	390	1%
6 to 8	2841	5%
9 to 11	8445	14%
12 to 14	14351	24%
15 to 17	7739	13%
18 to 20	11979	20%
21 to 23	9051	15%
24 to 26	6060	10%

Logic level report for top violating paths

WNS	LOGIC LEVELS	THRESHOLD	START POINT	END POINT
-0.3903	21	26	reg_1_/CK	latch/E
-0.3902	21	26	reg_2_/CK	latch/E
-0.3902	21	26	reg_3_/CK	reg_17_/D
-0.3900	25	26	reg_4_/CK	reg_21_/D
-0.3899	25	26	reg_5_/CK	reg_32_/D

Alternatively, you can view the infeasible paths by generating a timing report using the `-attributes` option with the `report_timing` command. The report shows each infeasible path whose startpoint and endport are marked with the `inf` attribute.

See Also

- Creating Exceptions for Infeasible Paths
- Creating Constraints Using Categorized Timing Reports
- Reporting Quality of Results

Creating Constraints Using Categorized Timing Reports

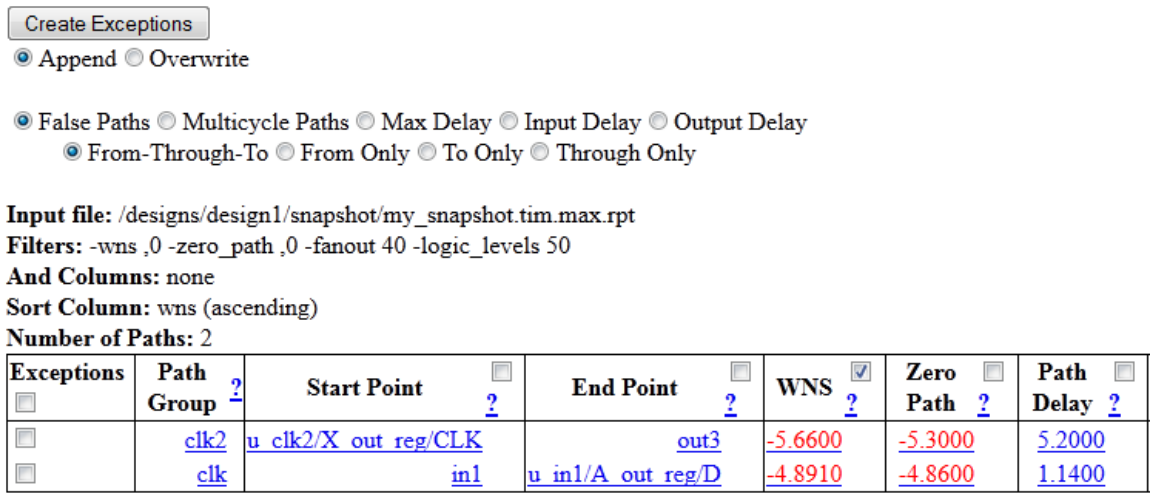
You can use a categorized timing report in HTML format to create the following constraints:

- False paths
- Multicycle paths
- Maximum delays
- Input delays
- Output delays

Note: You must first generate a categorized timing report by using a command sequence similar to the following:

```
de_shell> compile_exploration -scan
de_shell> create_qor_snapshot -nworst 3 -name my_snapshot
de_shell> query_qor_snapshot -name my_snapshot -output_file qor_report
de_shell> sh firefox qor_report.html
```

Figure 4-14: Categorized Timing Report in HTML Format



From the categorized timing HTML report, as shown in Figure 4-14, you can create the missing constraints and exceptions by following these steps:

1. Click the Append button.
2. Click the corresponding parameter button, such as Multicycle Paths.
3. Enter a value for the parameter.
4. Click the Create Exceptions button.

A new browser window that contains a set of commands based on the constraints you specified appears.

5. Update your SDC file with these commands.

See Also

- Measuring Quality of Results
- Reporting Infeasible Paths

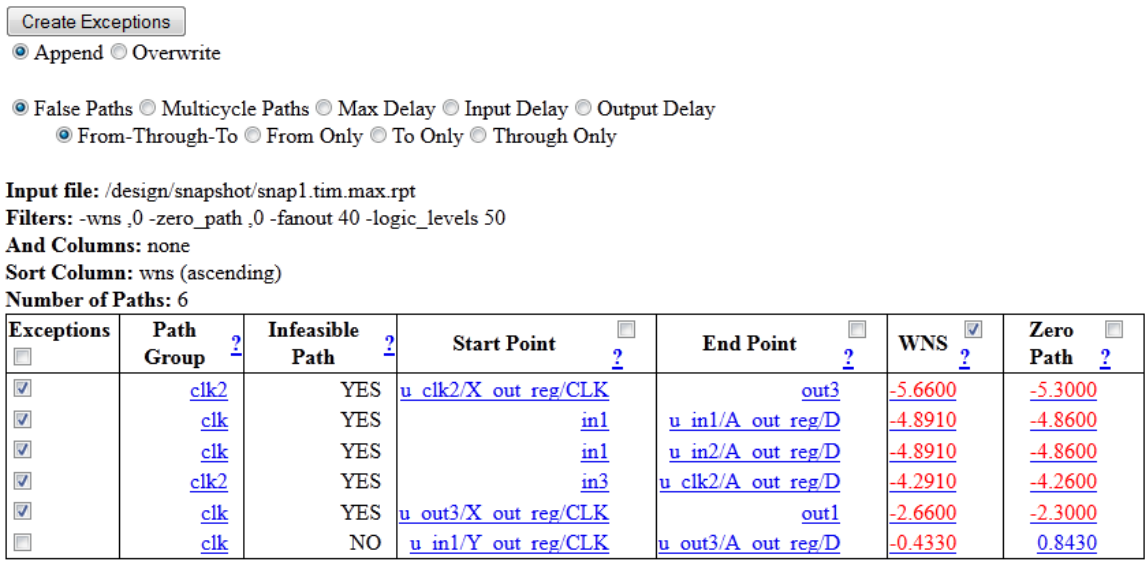
Creating Exceptions for Infeasible Paths

You can use a categorized timing report in HTML format to create exceptions for infeasible paths. You must first generate a categorized timing report that includes the infeasible paths in your design by using a command sequence similar to the following:

```
de_shell> compile_exploration
de_shell> create_qor_snapshot -max_paths 30 -infeasible_paths -name snap1
de_shell> query_qor_snapshot -infeasible_paths -name snap1 \
-columns {path_group infeasible_paths startpoint endpoint wns zero_path}\
-output_file snap1.html
de_shell> sh firefox snap1.html
```

Figure 4-15 shows the categorized timing report in HTML format generated by the command sequence.

Figure 4-15: Categorized Timing Report in HTML Format



Exceptions	Path Group ?	Infeasible Path ?	Start Point ?	End Point ?	WNS ?	Zero Path ?
☑	clk2	YES	u_clk2/X_out_reg/CLK	out3	-5.6600	-5.3000
☑	clk	YES	in1	u_in1/A_out_reg/D	-4.8910	-4.8600
☑	clk	YES	in1	u_in2/A_out_reg/D	-4.8910	-4.8600
☑	clk2	YES	in3	u_clk2/A_out_reg/D	-4.2910	-4.2600
☑	clk	YES	u_out3/X_out_reg/CLK	out1	-2.6600	-2.3000
☐	clk	NO	u_in1/Y_out_reg/CLK	u_out3/A_out_reg/D	-0.4330	0.8430

From the categorized timing HTML report, you can create the missing constraints and exceptions by following these steps:

1. Click the Append button.
2. Click the False Paths button.
3. Select the check box under the Exceptions column of each infeasible path for which you want to create an exception.
4. Click the Create Exceptions button.

   A new browser window that contains a set of commands based on the exceptions you specified appears. For example, when you select the first five infeasible paths, the following commands are displayed in the pop-up window:

   ```
   set_false_path -from u_clk2/X_out_reg/CLK -to out3;
   set_false_path -from in1 -to u_in1/A_out_reg/D;
   set_false_path -from in1 -to u_in2/A_out_reg/D;
   set_false_path -from in3 -to u_clk2/A_out_reg/D;
   set_false_path -from u_out3/X_out_reg/CLK -to out1;
   ```

5. Update your SDC file with these commands.

See Also

- Reporting Infeasible Paths
- Reporting Quality of Results

5 Working With Designs in Memory

DC Explorer 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 about how to open and modify a design, see

- Design Terminology
- Opening Designs
- Setting the Current Design
- Linking Designs
- Editing Designs
- Changing the Design Hierarchy
- Using Design Object Attributes
- Creating Designs
- Copying Designs
- Renaming Designs
- Saving Designs
- Removing Designs
- Design Database and Netlist Naming Consistency

Design Terminology

Different companies use different terminology for designs and their components. This section describes the terminology used in Synopsys synthesis tools and the relationships between design instances and references.

- Designs
- Design Objects
- Relationships Between Designs, Instances, and References

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 be compiled independently of one another, or they can be used as subdesigns in larger designs.

Designs can be flat or hierarchical:

- Flat designs

  A flat design contain no subdesigns and has only one structural level. It contains only library cells.

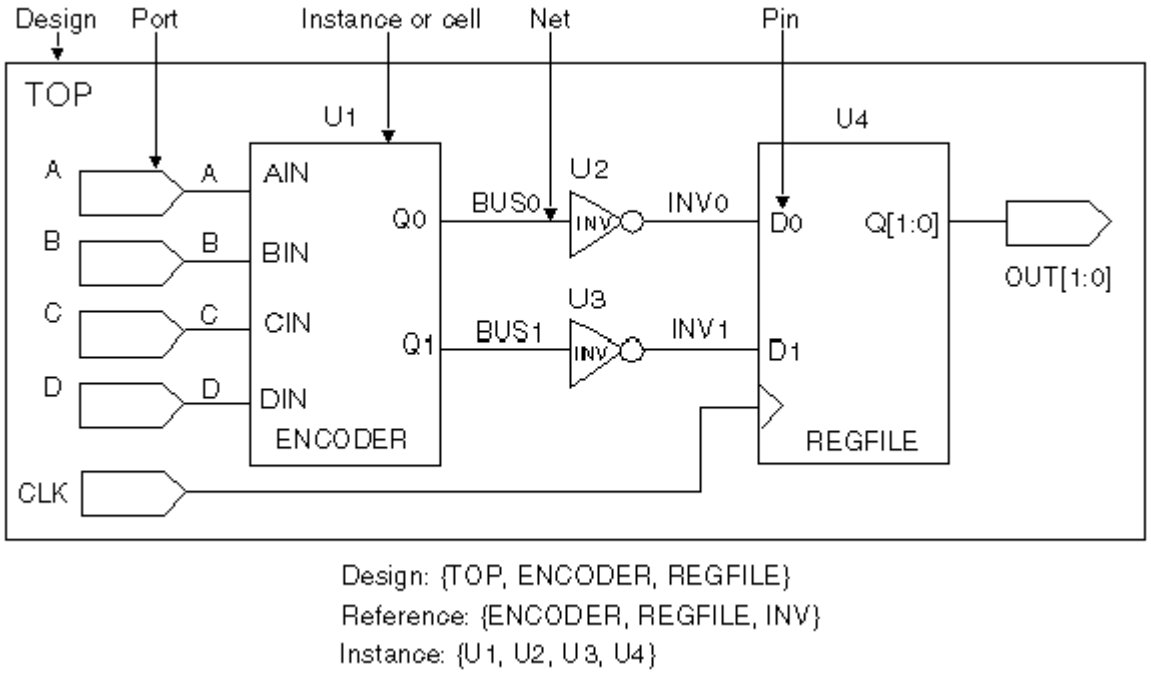
- Hierarchical designs

  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

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

Figure 5-1: Design Objects



Design: {TOP, ENCODER, REGFILE}
Reference: {ENCODER, REGFILE, INV}
Instance: {U1, U2, U3, U4}

Designs

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.

References

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.

Instances or Cells

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. Multiple

instances can point to the same reference, but each instance 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.

Port

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

Pins

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

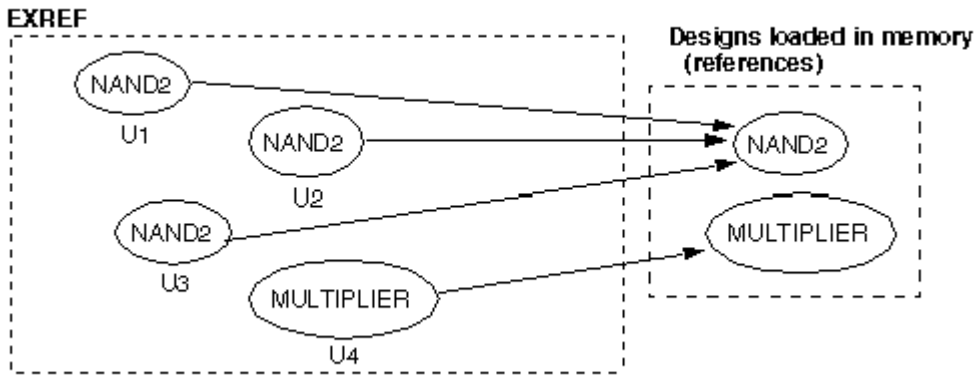
Nets

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

Relationships Between Designs, Instances, and References

Figure 5-2 shows the relationships between 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 once.

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.

See Also

- The Process of Resolving References

Opening Designs

Before you open a design, ensure that your design is in one of the following formats listed in Table 5-1:

Table 5-1: Supported Input Formats

Format	Description
.ddc	Synopsys internal database format (recommended)
SystemVerilog	IEEE standard SystemVerilog (see the HDL Compiler documentation)
Verilog	IEEE standard Verilog (see the HDL Compiler documentation)
VHDL	IEEE standard VHDL (see the HDL Compiler documentation)

To open a design, use either of the following two methods:

- The analyze and elaborate Commands

  For example,

  ```
  de_shell> analyze -format vhdl my_design
  de_shell> elaborate my_design
  ```

  If the `analyze` command reports errors, fix them in the HDL source file and run `analyze` again. After you modify an analyzed design, you must reanalyze it.

- The read_file Command

  For example,

  ```
  de_shell> read_file -rtl vhdl my_file
  ```

To learn the differences between the two methods and how to open designs in different formats, see

- Differences Between the Two Read Methods
- Opening HDL and .ddc Files

The analyze and elaborate Commands

The `analyze` command performs the following tasks:

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

The `elaborate` command performs the following tasks:

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

See Also

- *HDL Compiler for Verilog User Guide*
- *HDL Compiler for VHDL User Guide*

The read_file Command

The `read_file` command

- 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
- Does not create any intermediate files for Verilog

  To create intermediate files, set the `hdlin_auto_save_templates` variable to `true`.

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

If you do not specify the design format, the `read_file` command infers the format based on the file extension. If no known extension is used, DC Explorer uses 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 already compressed internally when it is written. To enable DC Explorer to automatically infer the file format for compressed files, use the following file naming structure: filename.format.gz.

The following formats are supported:

- ddc: the .ddc extension
- db: the .db, .sldb, .sdb, .db.gz, .sldb.gz, and .sdb.gz extensions
- Verilog: the .v, .verilog, .v.gz, and .verilog.gz extensions
- SystemVerilog: the .sv, .sverilog, .sv.gz, and .sverilog.gz extensions
- VHDL: the .vhd, .vhdl, vhd.gz, and .vhdl.gz extensions

See Also

- Linking Designs

Differences Between the Two Read Methods

Table 5-2 summarizes the differences between using the `analyze` and `elaborate` commands and using the `read_file` command.

Table 5-2: The analyze and elaborate Commands Versus the read_file Command

Comparison	The analyze and elaborate commands	The read_file command
Input formats	Support VHDL and Verilog formats.	Supports all formats.
When to use	Synthesize VHDL or Verilog files.	Synthesizes netlists, precompiled designs, and so forth.
Parameters	Allow you to set parameter values on the `elaborate` command line. For parameterized designs, you can use the `analyze` and `elaborate` commands to build a new design with nondefault values.	Cannot pass parameters. You must use directives in HDL.
Architecture	Allow you to specify the architecture to be elaborated.	Cannot specify the architecture to be elaborated.
Link process	The `elaborate` command executes the `link` command automatically to resolve references.	You must use the `link` command to resolve references.

See Also

- *HDL Compiler for Verilog User Guide*
- *HDL Compiler for VHDL User Guide*

Opening HDL and .ddc Files

You can open designs in HDL and .ddc formats:

- Opening HDL Files
- Opening .ddc Files

Opening HDL Files

To open HDL files, use one of the following methods:

- 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,

```
de_shell> analyze -format vhdl -library -work RISCTYPES.vhd
de_shell> analyze -format vhdl \
-library -work {ALU.vhd STACK_TOP.vhd  STACK_MEM.vhd ...}
de_shell> elaborate RISC_CORE \
-architecture STRUCT -library WORK -update
```

- The `read_file` command

  For example,

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

- The `read_verilog` or the `read_vhdl` command

  For example,

```
de_shell> read_verilog RISC_CORE.v
```

  Alternatively, you can use the `read_file -format VHDL` or the `read_file -format verilog` command.

Opening .ddc Files

To open .ddc files, use the `read_ddc` command or the `read_file -format ddc` command. For example,

```
de_shell> read_ddc RISC_CORE.ddc
```

The .ddc format is backward compatible but not forward compatible; that is, you can read .ddc files created with older versions of DC Explorer but not with later versions than the one you are using.

Setting the Current Design

To set the design that you are working on as the current design, use one of the following commands:

- The `read_file` command

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

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

- The `elaborate` command
- The `current_design` command

  Use this command to set any design in de_shell memory as the current design. For example,

  ```
  de_shell> current_design MY_DESIGN
  Current design is 'MY_DESIGN'. {MY_DESIGN}
  ```

To display the name of the current design, enter

```
de_shell> printvar current_design
current_design = "test"
```

To prevent excessive runtime usage, avoid writing scripts with a large number of `current_design` commands, such as in a loop.

Linking Designs

For a design to be complete, all cell instances in the design must be linked to the library components and designs that are referenced. To link designs, use the `link` command. DC

Explorer resolves the references using the link libraries set by the `link_library` variable, the `search_path` variable, and the `local_link_library` attribute.

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

In addition to linking designs, you can also perform the following tasks:

- Changing Design References
- Querying Design References
- Locating Designs by Using a Search Path

See Also

- The Process of Resolving References

Changing Design References

To change the component or design to which a cell or reference is linked, use the `change_link` command. When you specify a cell instance as the object, the link from that cell is changed. When you specify a reference as the object, the links from all cells having that reference are changed. DC Explorer changes the link only when you specify a reference component or design that has the same number of ports with the same size and direction as the original reference.

Running the `change_link` command copies all link information 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 `link` command on the design.

You can use the `-all_instances` option to specify an instance as an object that is at a lower level in the hierarchy and its parent design 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.

Example 1

The following commands change the link for the U1 and U2 cells from the current design to the MY_ADDER design:

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

Example 2

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

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

Example 3

The following commands change the link for all instances of the inv1 cell when its parent design, bot, is instantiated twice: mid1/bot1 and mid1/bot2.

```
de_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)

de_shell> get_cells -hierarchical -filter "ref_name == AN3"
{mid1/bot1/inv1 mid1/bot2/inv1}
```

Example 4

You can direct the tool to resolve a cell reference that is equivalent to the original cell reference but has different pin names. To map the pin names between the original and new cell references, use the `-pin_map {{old_pin1 new_pin1} {old_pin2 new_pin2}...}` option with the `change_link` command.

For example, the following command links the U1 cell in the current design to the AN21 cell mapping the old pin names (A1, A2, and Z) to the new pin names (A, B, and Y):

```
de_shell> change_link [get_cells U1] AN21 \
-pin_map {{A1 A} {A2 B} {Z Y}}
```

Locating Designs by Using a Search Path

To find a design file location, specify the file name with the `which` command. DC Explorer uses the search path defined in the `search_path` variable to look for the design file, starting with the leftmost directory specified in this variable, and reports the first design file it locates. By default, the search path includes the current working directory and $SYNOPSYS/libraries/syn, where $SYNOPSYS is the path to the installation directory. For example,

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

To specify other directories in addition to the default search path, enter a command similar to the following

```
de_shell> lappend search_path project
```

Querying Design References

You can query design references by using the following methods:

- To report information about all references in the current instance or current design, use the `report_reference` command.

- To return a collection of instances that you specify, use the `get_references` command.

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

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

- To see the reference names, use the `report_cell` command.

  For example,

```
de_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
```

The Process of Resolving References

DC Explorer performs the following steps to resolve references:

1. It determines which library components and subdesigns are referenced in the current design and the hierarchy.
2. It searches the link libraries to locate the references.
   1. DC Explorer first searches the libraries and design files defined in the `local_link_library` attribute of the current design.

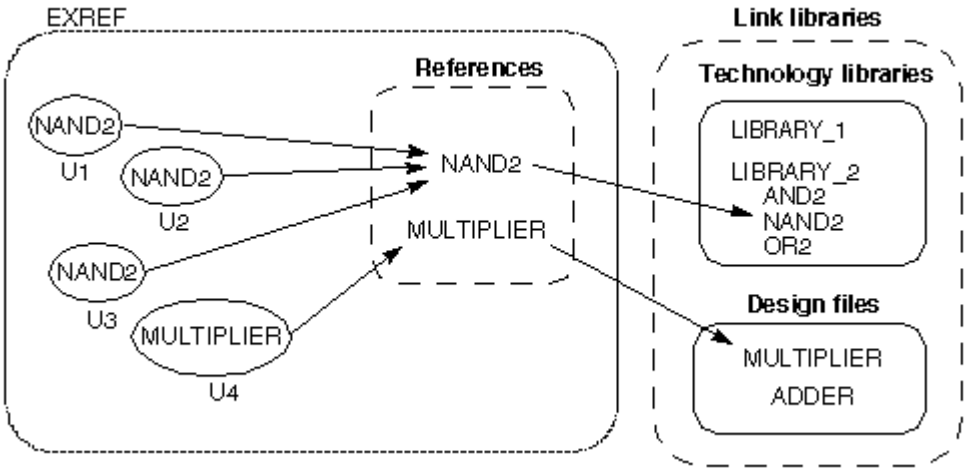
      In a hierarchical design, DC Explorer considers only the local link libraries of the top-level design and ignores the local link libraries associated with the subdesigns.

   2. If the `link_library` variable is set to an asterisk (*), DC Explorer searches in memory for the reference.
   3. DC Explorer 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.
4. It links (connects) the located references to the design.

DC Explorer uses the first reference it locates. If it locates additional references with the same name, it generates a warning message identifying the ignored references. If DC Explorer does not find the reference, it issues a warning.

The arrows in Figure 5-3 show the connections that the link process creates between the instances, references, and link libraries. In this example, DC Explorer finds the NAND2 component in the LIBRARY_2 logic library and the MULTIPLIER subdesign in a design file.

Figure 5-3: Resolving References



See Also

- Locating Designs by Using a Search Path

Editing Designs

You use the commands described in Design Editing Tasks and Commands to edit designs. For a unique design, these netlist editing commands accept instance objects, that is, cells at a lower level of hierarchy. You can operate from any level in a hierarchical design without using the `current_design` command. For example, you can enter the following command to create a cell named my_cell in the design mid1:

```
de_shell> create_cell mid1/my_cell 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, the following sequence of commands replaces the reference for the U8 cell with a high-power inverter.

```
de_shell> get_pins U8/*
{"U8/A", "U8/Z"}
de_shell> all_connected U8/A
{"n66"}
de_shell> all_connected U8/Z
{"OUTBUS[10]"}
de_shell> remove_cell U8
Removing cell 'U8' in design 'top'.
```

```
1
de_shell> create_cell U8 IVP
Creating cell 'U8' in design 'top'.
1
de_shell> connect_net n66 [get_pins U8/A]
Connecting net 'n66' to pin 'U8/A'.
1
de_shell> connect_net OUTBUS[10] [get_pins U8/Z]
Connecting net 'OUTBUS[10]' to pin 'U8/Z'.
```

Alternatively, you can achieve the same result by using the `change_link` command. For example,

```
de_shell> change_link U8 IVP
```

Additional netlist editing commands include the `size_cell`, `insert_buffer`, and `remove_buffer` commands.

To query or specify particular designs and objects, see

- Querying the Design and Objects
- Specifying Design Objects

See Also

- Design Editing Tasks and Commands

Design Editing Tasks and Commands

You can incrementally edit a design in memory or change a netlist by using the de_shell commands listed in Table 5-3.

Table 5-3: Design Editing Tasks and Commands

Object	Task	Command
Cells	Create a cell Delete a cell	`create_cell` `remove_cell`

Nets	Create a net
	Connect a net
	Disconnect a net
	Delete a net
		`create_net`
		`connect_net`
		`disconnect_net`
		`remove_net`

Ports	Create a port
	Delete a port
		`create_port`
		`remove_port`
		`remove_unconnected_ports`

Pins	Connect pins	`connect_pin`

Buses	Create a bus Delete a bus	`create_bus` `remove_bus`

See Also

- Editing Designs

Querying the Design and Objects

To find out what designs are in memory, use the `list_designs` command. Use the `-show_file` option if you also want to find out the corresponding design file names. For example,

```
de_shell> list_designs
A (*)    B    C

de_shell> list_designs -show_file

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

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

The asterisk (*) marks the current design. The B and C designs are both contained in the B.ddc file.

You can list various objects in the current design by using the commands listed in Table 5-3.

Table 5-4 lists the commands and the tasks they perform.

Table 5-4: Commands to Query Design Objects

Object	Command	Task
Instance	`list_instances` `report_cell`	Lists instances and their references. Displays information about instance.
Reference	`report_reference`	Displays information about references.

Port	`report_port`
	`report_bus`
	`all_inputs`
	`all_outputs`
		Displays information about ports.
		Displays information about bused ports.
		Returns all input ports.
		Returns all output ports.

Net	`report_net` `report_bus`	Displays information about nets. Displays information about bused nets.

Clock	`report_clock` `all_clocks`	Displays information about clocks. Returns all clocks.
Register	`all_registers`	Returns all registers.
Collection	`get_cells`, `get_designs`	Returns a collection of cells, designs, libraries and library cells, library cell pins, nets, pins, or ports.

Specifying Design Objects

You can specify design objects by

- Using a Relative Path
- Using an Absolute Path

Using a Relative Path

When you use a relative path to specify a design object, the object must be in the current design and the path is 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. To change the current instance, use the `current_instance` command.

For example, to mark the U1/U15 hierarchical cell in the Count_16 design with the `dont_touch` attribute, use either of the following command sequences:

- The frame of reference remains at the top level of the Count_16 design

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

- The frame of reference changes to the U1 instance

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

  DC Explorer interprets all future object specifications relative to the U1 instance. The `current_instance` command points to the current instance.

To display the current instance, enter

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

To reset the current instance to the top level of the current design, use the `current_instance` command. For example

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

Using an Absolute Path

When you use an absolute path to specify an object, the object can be in any design in de_shell memory. The syntax for specifying an object is

```
[file:]design/object
```

where

- *file* is the path name of the design file in memory followed by a colon (:)

  Use this argument when multiple designs in memory have the same name.

- *design* is the name of the design in de_shell memory
- *object* is 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, DC Explorer looks for the object by using the types allowed by the command. To specify an object type, use the `get_*` command.

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

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

See Also

- Querying the Design and Objects

Changing the Design Hierarchy

Ideally, the hierarchy of the design in DC Explorer should accurately reflect the design partitioning in the HDL description. However, you can change the hierarchy in DC Explorer to experiment with alternative hierarchical partitioning without modifying the original HDL description.

Before making any design hierarchy changes, use the `report_hierarchy` command to show the current design hierarchy. You can use the following methods to change the hierarchy without modifying the HDL description.

- Adding Levels of Hierarchy
- Removing Levels of Hierarchy

- Merging Cells From Different Subdesigns

After making any design hierarchy changes, use the `report_hierarchy` command to verify the hierarchy changes.

Adding Levels of Hierarchy

To create a level of hierarchy by grouping cells or related components into a subdesign, use the `group` command. The grouped cells are replaced by a new instance 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. Grouping cells might not preserve all the attributes and constraints of the original cells.

The following examples group cells into subdesigns:

To group two cells into a new design named my_design with the U instance name, enter

```
de_shell> group {u1 u2} -design_name my_design -cell_name U
```

To group all cells that have the alu prefix into a new design named uP with the UCELL instance name, enter

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

To group the bot1, bot2, and j cells that have the unique mid1 parent design into a new subdesign named my_design with the U1 instance name, enter

```
de_shell> group {mid1/bot1 mid1/bot2 mid1/j} \
-cell_name U1 -design_name my_design
```

Alternatively, you can use the following two commands:

```
de_shell> current_design mid1
de_shell> group {bot1 bot2 j} -cell_name U1 -design_name my_design
```

The following examples group related components into subdesigns:

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

```
de_shell> group -hdl_block ftj/abc -design_name new_block
```

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

```
de_shell> group -hdl_block ftj -hdl_bussed
```

Removing Levels of Hierarchy

DC Explorer does not optimize logic across hierarchical boundaries; therefore, you might need to remove levels of hierarchy in certain designs to improve the timing. Removing a level of

hierarchy is called ungrouping, which merges subdesigns of a hierarchical level into the parent design. Designs, subdesigns, and cells marked with the `dont_touch` attribute cannot be ungrouped.

To ungroup hierarchies, you can use following commands:

- The `ungroup` command before optimization

  See the Ungrouping Hierarchies Before Optimization topic.

- The `set_ungroup` command explicitly during optimization

  See the Ungrouping Hierarchies Explicitly During Optimization topic.

- The `compile_exploration` command implicitly during optimization

  See the Ungrouping Hierarchies Automatically During Optimization topic.

See Also

- Timing Constraint Preservation During Ungrouping

Timing Constraint Preservation During Ungrouping

Hierarchical pins are removed when cells are ungrouped. DC Explorer handles timing constraints placed on hierarchical pins in different ways based on whether you are ungrouping a hierarchy before or after optimization.

Table 5-5 summarizes the effects that ungrouping has on timing constraints within different compile flows.

Table 5-5: Preserving Hierarchical Pin Timing Constraints

Compile flow	Effect on hierarchical pin timing constraints
Ungrouping a hierarchy before optimization by using the `ungroup` command	Timing constraints placed on hierarchical pins are preserved.
Ungrouping a hierarchy during optimization by using the `set_ungroup` command followed by the `compile_exploration` command	Timing constraints placed on hierarchical pins are not preserved by default. To preserve timing constraints, set the `auto_ungroup_preserve_constraints` variable to `true`.
Automatically ungrouping a hierarchy during optimization by using the `compile_exploration` command	To ungroup the hierarchy and preserve timing constraints, set the `auto_ungroup_preserve_constraints` variable to `true`.

When preserving timing constraints, DC Explorer reassigns the timing constraints to appropriate adjacent pins that remain on the same net 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.

When DC Explorer moves the constraints to a leaf cell, it preserves the constraints during compilation by marking the cell with the `size_only` attribute. DC Explorer chooses the direction that minimizes the number of leaf cells to be marked with the `size_only` attributes when both the forward and backward directions are possible.

When you ungroup an unmapped design, DC Explorer moves the constraints on a hierarchical pin to a leaf cell and marks the cell with the `size_only` attribute. The constraints are preserved through the compile process only if a one-to-one matching exists between an 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.

Ungrouping Hierarchies Automatically During Optimization

To automatically ungroup hierarchies during optimization, you can use the `compile_exploration` command. DC Explorer provides two strategies to automatically ungroup hierarchies: area-based auto ungrouping and delay-based auto ungrouping.

By default, the `compile_exploration` command performs delay-based auto ungrouping, which ungroups hierarchies along the critical path and is used essentially for timing optimization. In addition, the command performs area-based auto ungrouping before initial mapping. DC Explorer estimates the area for unmapped hierarchies and removes small subdesigns to improve area and timing quality of results.

To report the hierarchies that were ungrouped during the `compile_exploration` process, use the `report_auto_ungroup` command. This report gives instance names, cell names, and the number of instances for each ungrouped hierarchy.

See Also

- Timing Constraint Preservation During Ungrouping
- The *Design Compiler User Guide*

Ungrouping Hierarchies Before Optimization

Use the `ungroup` command to ungroup one or more designs before optimization. You can use this command from any level in a hierarchical design without using the `current_design` command.

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

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

To ungroup a list of cells, enter

```
de_shell> ungroup {high_decoder_cell low_decoder_cell}
```

To ungroup the U1 cell and specify the U1: prefix for the new cells, enter

```
de_shell> ungroup U1 -prefix "U1:"
```

To completely collapse the hierarchy of the current design, enter

```
de_shell> ungroup -all -flatten
```

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

```
de_shell> ungroup -start_level 3 CELL_X
```

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

```
de_shell> ungroup -start_level 3 {U1 U2}
```

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

```
de_shell> ungroup -start_level 3 -all
```

Instance objects are cells at a lower level of a hierarchy. The following example illustrates how the `ungroup` command accepts instance objects when the parent design is unique. In the example, MID1/BOT1 is a unique instantiation of design BOT. The command ungroups the MID1/BOT1/DES1 and MID1/BOT1/DES2 cells.

```
de_shell> ungroup {MID1/BOT1/DES1 MID1/BOT1/DES2}
```

Alternatively, you can use the following two commands:

```
de_shell> current_instance MID1/BOT1

de_shell> ungroup {DES1 DES2}
```

See Also

- Timing Constraint Preservation During Ungrouping

Ungrouping Hierarchies Explicitly During Optimization

You can control which designs to be ungrouped during optimization by using the `set_ungroup` command followed by the `compile_exploration` command. You specify the cells or designs to be ungrouped, and the `set_ungroup` 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 the U1 cell during optimization, enter

```
de_shell> set_ungroup U1
de_shell> compile_exploration
```

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

```
de_shell> get_attribute object ungroup
```

To remove an `ungroup` attribute, use the `remove_attribute` command, or set the `ungroup` attribute to `false` by using the `set_ungroup` command. For example,

```
de_shell> set_ungroup object false
```

See Also

- Timing Constraint Preservation During Ungrouping

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 the u_add and u_mult subdesigns.

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

Using Design Object 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, although modified attribute settings are not automatically saved.

DC Explorer 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
- Logic 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 read-only, and DC Explorer sets these attribute values. You cannot change the values of read-only attributes, but you can change the values of read/write attributes.

To learn how to query, change, set attributes on design objects, see

- Querying Attribute Values
- Setting Attributes on Objects
- Changing Object Attributes
- Commands to Get Attribute Descriptions
- Object Search Order

Querying Attribute Values

To query the values of design object attributes, use the `report_attribute` command, the `get_attribute` command, or the GUI.

All Attributes

Use the `report_attribute` command to see all attributes. For example,

```
de_shell> report_attribute object_list
```

Specific Attributes

Use the `get_attribute` command to see a specific attribute. The following example reports the `max_fanout` attribute on the OUT7 port:

```
de_shell> get_attribute OUT7 max_fanout
```

```
Performing get_attribute on port 'OUT7'.
{3.000000}
```

You can query the bounding box information for pins and ports. For example, the following commands report the `bbox` attribute on the Q pin of register reg_1 and the test_mode port of the current design:

```
de_shell> get_attribute [get_pins reg_1/Q] bbox
de_shell> get_attribute [get_ports test_mode] bbox
```

Note: To query the `bbox` attribute for pins and ports, you need to enable the physical flow capability by setting the `de_enable_physical_flow` variable to `true`.

If an attribute applies to more than one object type, DC Explorer searches the database for the specified object.

Using the GUI

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.

See Also

- Setting Attributes on Objects
- Changing Object Attributes
- Commands to Get Attribute Descriptions
- Object Search Order

Setting Attributes on Objects

To set attributes on design objects, use one of the following two methods:

- Using an attribute-specific command

  Use an attribute-specific command to set the command's associated attribute on an object. The following example marks the U1 cell with the `dont_touch` attribute.

  ```
  de_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. This command enforces the predefined attribute type, such as string or integer. If you set an attribute with a value of an incorrect type, the command issues an error message. To find the predefined type for an attribute, use the `list_attributes` command.

  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.

  The following example sets the `dont_touch` attribute on the lsi_10k/FJK3 library cell:

```
de_shell> set_attribute lsi_10K/FJK3 dont_touch true
```

See Also

- Querying Attribute Values
- Changing Object Attributes
- Commands to Get Attribute Descriptions
- Object Search Order

Changing Object Attributes

You can change, remove, or save attributes set on objects:

- Remove a specific attribute on an object by using the `remove_attribute` command

  You cannot use this 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 inherits the attribute.

  The following example removes the `max_fanout` attribute from the OUT7 port:

```
de_shell> remove_attribute OUT7 max_fanout
```

- Remove all attributes from the current design by using the `reset_design` command

  The `reset_design` command removes all design information, including clocks, input and output delays, path groups, operating conditions, timing ranges, and wire load models. Running this command is often equivalent to restarting the design.

- Change the values of attributes

  To change the value of an attribute, remove the attribute and then re-create it to set the type.

- Save the attributes by using the `write_script` command

  DC Explorer does not automatically save attribute values when you exit de_shell. To re-create the attributes, use the `write_script` command to generate a de_shell script. By default, the `write_script` command reports the attribute-setting commands to the screen. Use the redirection operator (>) to redirect the output to a file. For example,

```
de_shell> write_script > attr.scr
```

Note: The `write_script` command supports only application (system-defined) attributes, not user-defined attributes.

See Also

- Querying Attribute Values
- Setting Attributes on Objects
- Commands to Get Attribute Descriptions
- Object Search Order

Commands to Get Attribute Descriptions

Most attributes apply to only a limited selection of object types; 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. Some attributes are predefined and are recognized by DC Explorer; other attributes are user-defined. You can get detailed information about the predefined attributes by using the commands listed in Table 5-6.

Table 5-6: 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`

See Also

- Querying Attribute Values
- Setting Attributes on Objects
- Changing Object Attributes
- Object Search Order

Object Search Order

Commands that can set an attribute on more than one type of object use a particular search order to find the object to which the attribute applies. For example, the `set_dont_touch` command operates on cells, nets, references, and library cells. If you specify an object named X with the `set_dont_touch` command and two objects (such as the design and a cell) are named X, DC Explorer applies the attribute to the first object type found. In this case, the attribute is set on the design, not on the cell.

DC Explorer searches until it finds a matching object, or it displays an error message if it does not find a matching object.

You can search for an object of a specified type by using a `get_*` command to specify the object. For example, assume that the current design contains a cell and a net both named my_object. The following command sets the `dont_touch` attribute on the cell because of the default search order:

```
de_shell> set_dont_touch my_object
```

To set the `dont_touch` attribute on the net instead, enter

```
de_shell> set_dont_touch [get_nets my_object]
```

See Also

- Querying Attribute Values
- Setting Attributes on Objects
- Changing Object Attributes
- Commands to Get Attribute Descriptions

Creating Designs

To create new designs, use the `create_design` command. You use this command to create an empty design in de_shell memory. The following example creates a new design named my_ design. The memory file name is my_design.db, and the path is the current working directory.

```
de_shell> create_design my_design
de_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
```

Designs created with the `create_design` command contain no design objects. To add design objects to the new design, use commands such as `create_clock`, `create_cell`, `create_bus`, `create_port`, and `connect_net`.

Copying Designs

You use the `copy_design` command to copy a design in memory to a new design The following example copies the test design in memory and names the new design test_new:

```
de_shell> copy_design test test_new
Information: Copying design /designs/test.ddc:to
designs/test.ddc:test_new

de_shell> list_designs -show_file
/designs/test.ddc
test (*) test_new
```

To manually create new subdesigns that you can edit and to link instances to those new subdesigns, use the `copy_design` and `change_link` commands. For example, assume that a design has two identical cells, U1 and U2, both linked to COMP. Enter the following command sequence to create new, separate linked subdesigns:

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

de_shell> change_link U1 COMP1
Performing change_link on cell 'U1'.

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

de_shell> change_link U2 COMP2
```

```
Performing change_link on cell 'U2'.
de_shell> create_design my_design
de_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
```

Designs created with the `create_design` command contain no design objects. To add design objects to the new design, use commands such as `create_clock`, `create_cell`, `create_bus`, `create_port`, and `connect_net`.

See Also

- Editing Designs

Renaming Designs

You can assign a new name to a design or move a list of designs to a file. To rename a design in memory, use the `rename_design` command. To save the renamed file, use the `write_file` command. You can use the `list_designs` command to show the design before and after running the `rename_design` command. For example,

```
de_shell> list_designs -show_file
/designs/test.ddc
test(*) test_new

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

de_shell> list_designs -show_file
/designs/test.ddc
test (*) test_new_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 command sequence adds the NEW_ prefix to the D design and updates the links for its instance cells:

```
de_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}

de_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)

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

```
de_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}
```

In this example, the first step reports the cells that instantiate the D design. After you run the second step, the instances are linked to the renamed reference design: NEW_D.

Saving Designs

You can save (write to disk) the design and subdesigns of the design hierarchy at any time, using different names or formats. After you modify a design, you should manually save it because DC Explorer does not automatically save designs.

DC Explorer supports the design file formats listed in Table 5-7.

Table 5-7: Supported Output Formats

Format	Description
.ddc	Synopsys internal database format
Verilog	IEEE Standard Verilog (see the HDL Compiler documentation)
svsim	SystemVerilog netlist wrapper. Note: The `write_file -format svsim` command writes only the netlist wrapper, not the gate-level design under test (DUT) itself. To write the gate-level DUT, you must use the existing `write_file -format verilog` command.
VHDL	IEEE Standard VHDL (see the HDL Compiler documentation)
Milkyway	Format for writing a Milkyway database within DC Explorer

You can use the following methods to save design files:

- Use the `write_file` command to convert designs in memory to a specified format and save that representation to disk

  By default, the `write_file` command saves only the top-level design. To save the entire design, specify the `-hierarchy` option. The following example writes all designs in the hierarchy of the top design in a .ddc file by using the -hierarchy and `-format` options:

  ```
  de_shell> write_file -hierarchy -format ddc top
  Writing ddc file 'top.ddc' Writing ddc file 'A.ddc' Writing ddc file
  'B.ddc'
  ```

  The following example writes multiple designs to the test.ddc file by using the `-output` option:

  ```
  de_shell> write_file -format ddc -output test.ddc {ADDer MULT16}
  Writing ddc file 'test.ddc'
  ```

- Use the `write_milkyway` command to write to a Milkyway database

  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.

See Also

- Design Database and Netlist Naming Consistency
- Removing Designs
- Using a Milkyway Database

Removing Designs

After completing a compilation session and saving the optimized design, you can delete the design from memory before opening another one. To remove a design from memory, use the `remove_design` command. By default, the `remove_design` command removes only the specified designs from de_shell memory.

If you define variables that reference design objects, DC Explorer removes these references when you remove the design from memory. This prevents future commands from operating on nonexistent design objects. For example,

```
de_shell> set_app_var PORTS [all_inputs]
{"A0", "A1", "A2", "A3"}
de_shell> query_objects $PORTS
PORTS = {"A0", "A1", "A2", "A3"}
de_shell> remove_design
Removing design 'top'
1
de_shell> query_objects $PORTS
Error: No such collection '_sel2' (SEL-001)
```

Design Database and Netlist Naming Consistency

Before writing a netlist from within de_shell, ensure that all net and port names conform to the naming conventions for your layout tool and 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.

Topics in this section

Naming Rules in the .synopsys_dc.setup File

You use the `define_name_rules` command to define a set of rules for naming design objects. The .synopsys_dc.setup file shows the following naming rules provided by a specific layout tool vendor. These naming rules

- Limit object names to alphanumeric characters
- Change DesignWare cell names to valid names (changes “*cell*” to “U” and "*-return” to “RET”)

```
define_name_rules simple_names -allowed "A-Za-z0-9_" \
-last_restricted "_" \
-first_restricted "_" \
-map { {{"\*cell\*","U"}, {"*-return","RET"}} }
```

For example, you can specify the `-map` option to remove trailing underscores from cell names:

```
de_shell> define_name_rules naming_convention \
 -map {{{_$, ""}} } -type cell
```

To list available name rules, use the `report_name_rules` command.

Note: Your vendor might use different naming conventions. Check with your vendor to determine the naming conventions that you need to follow.

Resolving Naming Problems

You might encounter conflicts in naming conventions in 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 that have different syntax definitions. Using tool sets from several vendors can introduce additional naming problems.

Correct naming helps eliminate mismatch errors and the need for name escaping in your design. To resolve naming issues, you use the `change_names` command to resolve naming issues and make the name changes in the design database file before you write any files. Your initial flow is

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 optimize your design again 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 switch before you write the design.

   Note: 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,

```
de_shell> change_names -rules verilog -hierarchy
```

4. Write files to disk by using 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 files matches the design database file. Enter the following commands and compare the output.

```
de_shell> write -format verilog -hierarchy -output "consistent.v"
de_shell> write -format ddc -hierarchy -output "consistent.ddc"
```

6. Write the files for third-party tools.

To show effects of the `change_names` command without actually making the changes, use the `report_names` command.

Avoiding Bit-Blasted Ports in SystemVerilog and VHDL Structures

You can use the `change_names` command to avoid bit-blasted ports in SystemVerilog structs and VHDL records (packed or unpacked). To enable this capability, specify the `-preserve_struct_ports` option with the `define_name_rules` command. You can add this option to the existing Verilog naming rules as follows:

```
de_shell> define_name_rules verilog -preserve_struct_ports
de_shell> change_names -hierarchy -rules verilog
```

Note that using either of the following options of the `define_name_rules` command overrides the `-preserve_struct_ports` option:

- `-remove_port_bus`
- `-dont_change_bus_members`

As shown in the following examples, the Verilog netlist 2 preserves the struct ports using the naming rules specified by the preceding two commands:

- RTL

```
typedef struct {logic blue; logic red;} color;
module top (
   input  color i_color,
   output color o_color
);
assign o_color = i_color;

endmodule
```

- Netlist 1 with bit-blasted struct ports

```
module top ( i_color_blue_, i_color_red_, o_color_blue_, o_color_
red_ );
  input  i_color_blue_, i_color_red_;
  output o_color_blue_, o_color_red_;
  wire   o_color_blue_, o_color_red_;
  assign o_color_blue_ = i_color_blue_;
  assign o_color_red_  = i_color_red_;
endmodule
```

- Netlist 2 preserving the struct ports

```
module top ( i_color, o_color );
  input  [1:0] i_color;
  output [1:0] o_color;
assign o_color[1] = i_color[1];
assign o_color[0] = i_color[0];
endmodule
```

6 Defining the Design Environment

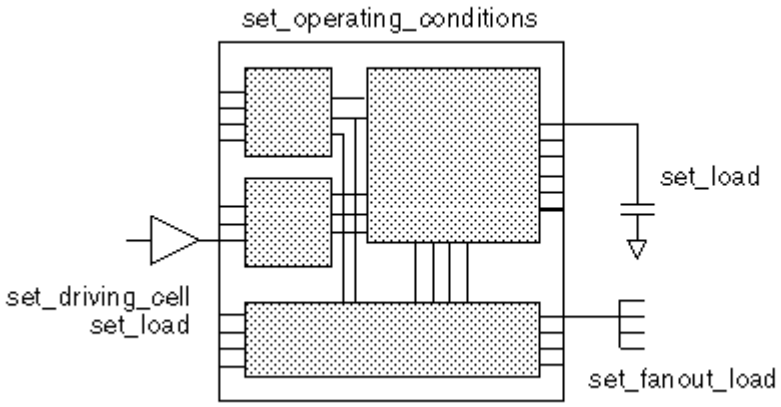
Before optimizing your design, you must define the design environment in which the design is expected to operate by specifying operating conditions and system interface characteristics. Operating conditions include temperature, voltage, and process variations. System interface characteristics consist of input drivers, input and output loads, and fanout load. The design environment model directly affects design synthesis results.

To learn how to define the design environment, see

- Operating Conditions
- Defining the Operating Conditions
- Modeling the System Interface
- Setting Logic Constraints on Ports
- Multivoltage Designs Using UPF

Figure 6-1 illustrates the commands used to define the design environment.

Figure 6-1: Commands Used to Define the Design Environment



Operating Conditions

In most technologies, variations in operating temperature, supply voltage, and manufacturing process parameters can strongly affect circuit performance. 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 characterization.

- Supply voltage variation

  The design's supply voltage can vary from the established ideal value during 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, DC Explorer must consider the worst-case and best-case scenarios for the expected variations in process, temperature, and voltage.

See Also

- Defining the Operating Conditions

Defining the Operating Conditions

Most logic libraries have predefined sets of operating conditions. Before you set the operating conditions, you can

- List the operating conditions defined in a logic library by using the `report_lib` command

  The logic library must be loaded in memory before you can run the `report_lib` command. To find out which libraries loaded in memory, use the `list_libs` command. The following example generates a report for the my_lib library that is stored in my_lib.db:

  ```
  de_shell> read_file my_lib.db
  de_shell> report_lib my_lib
  ```

- List the operating conditions defined for the current design by using the `current_design` command

If the logic library contains operating condition specifications, you can use them as the default. To specify explicit operating conditions, which supersede the default, use the `set_operating_conditions` command. The following example sets the operating conditions for the current design to worst-case commercial:

```
de_shell> set_operating_conditions WCCOM -library my_lib
```

Multicorner-Multimode Designs

Designs are often required to operate under multiple modes and multiple operating conditions, also known as corners. Such designs are referred to as multicorner-multimode designs. The tool can analyze and optimize across multiple modes and corners concurrently. The multicorner-multimode feature provides compatibility between flows in the DC Explorer and IC Compiler tools.

To define the modes and corners, use the `create_scenario` command. A scenario definition includes commands that specify the TLUPlus libraries, operating conditions, and constraints. To learn how to define scenarios, see Defining Scenarios.

See Also

- Operating Conditions
- Optimizing Multicorner-Multimode Designs

Modeling the System Interface

To model the design's interaction with the external system, you perform the following tasks:

- 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

To determine the delay and transition time characteristics of incoming signals, DC Explorer needs information about the external drive strength and the loading at each input port. Drive strength is the reciprocal of the output drive resistance, and the transition delay at an input port is the product of the drive resistance and the capacitance load of the input port.

By default, DC Explorer assumes zero drive resistance on input ports, meaning infinite drive strength. To set a realistic drive strength, use one of the following commands:

- The `set_driving_cell` command sets drive characteristics on ports driven by cells in the logic library.

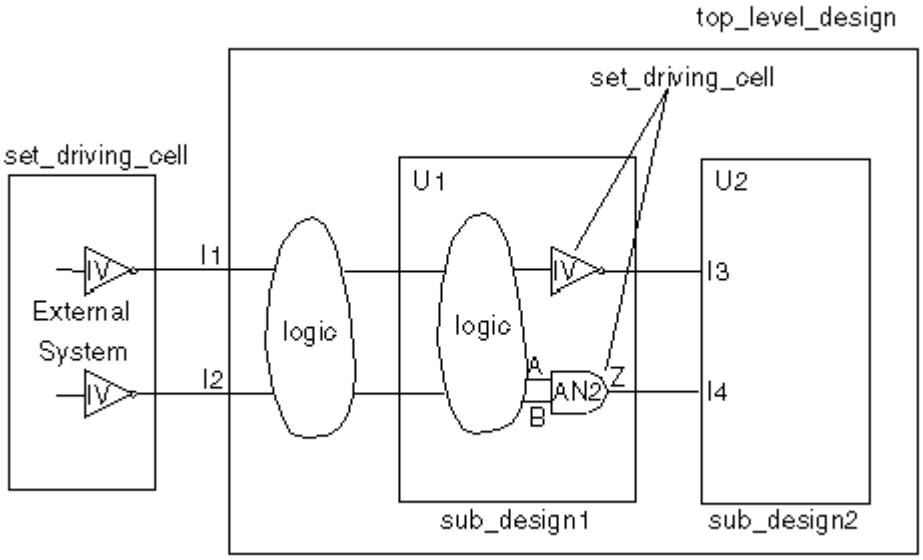
The command associates a library pin with an input port so that the delay calculator can accurately model the drive capability of an external driver. To remove driving cell attributes on ports, use the `remove_driving_cell` command.

- The `set_input_transition` command sets the signal transition time on the top-level ports when the input drive capability cannot be characterized by a cell in the logic library.

Both the `set_driving_cell` and `set_input_transition` commands affect the port transition delay, but they do not set design rule constraints on input ports, such as the maximum fanout and maximum transition time. However, the `set_driving_cell` command does place design rule constraints on input ports if the specified driving cell has design rule constraints.

Figure 6-2 shows a hierarchical design. The top-level design has two subdesigns, U1 and U2. The I1 and I2 ports of the top-level design are driven by the external system and have a drive resistance of 1.5.

Figure 6-2: Drive Characteristics



To set the drive characteristics for this example, follow these steps:

1. Use the `set_driving_cell` command to define the drive resistance.

```
de_shell> current_design top_level_design
de_shell> set_driving_cell -lib_cell IV {I1 I2}
```

2. Change the current design to sub_design2 to describe the drive capability for the ports on the sub_design2 design.

```
de_shell> current_design sub_design2
```

3. Associate the drive capability of the IV cell with the I3 port by using the `set_driving_`

cell command.

```
de_shell> set_driving_cell -lib_cell IV {I3}
```

4. Different arcs of the AN2 cell have different transition times. To check for setup violations, specify the slowest AN2 arc, assuming the slowest arc is from B to Z.

```
de_shell> set_driving_cell -lib_cell AN2 -pin Z -from_pin B {I4}
```

For checking setup violations, the worst-case arc is the slowest arc. For checking hold violations, the worst-case arc is the fastest arc.

Note: For heavily loaded driving ports, such as clock nets, keep the drive strength setting at zero so that DC Explorer does not buffer the nets. Buffering of clock nets should be carried out separately during clock tree synthesis.

Defining Loads on Input and Output Ports

By default, DC Explorer applies zero capacitive load to input and output ports. To set a capacitive load value on input and output ports, use the set_load command. DC Explorer uses this value to select an appropriate drive strength of an output pad and to model the transition time on an input pad.

The following example sets a capacitance load on the out1 pin:

```
de_shell> set_load 30 {out1}
```

You should specify the load value in units 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. To list the library units, use the report_lib command.

Defining Fanout Loads on Output Ports

To model the external fanout effects of one or more ports, use the set_fanout_load command to specify the expected fanout load values on output ports. For example,

```
de_shell> set_fanout_load 4 {out1}
```

DC Explorer 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.

Fanout load is a value that has no unit and represents a numerical contribution to the total fanout, expressed as the number of standard-cell inputs connected to the net. It is not a capacitance value.

See Also

- Defining Maximum Fanout

Setting Logic Constraints on Ports

To improve optimization results, you can set logic constraints on ports to eliminate redundant ports or inverters by using the following methods:

- Setting logic equivalence

  Some input ports are driven by logically related signals. To define that two input ports are logically equivalent or logically opposite, use the `set_equal` or the `set_opposite` command respectively.

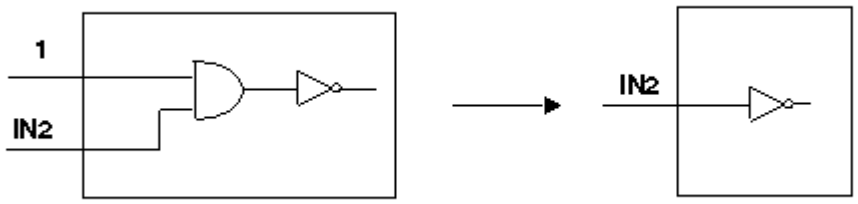
  The following example specifies that the IN_X and IN_Y ports are logically equal:

```
de_shell> set_equal IN_X IN_Y
```

- Assigning constant values to input ports

  Setting a constant value to an input enables DC Explorer to simplify the surrounding logic function during optimization and to create a smaller design. To assign a don't care, logic 1, or logic 0 value to inputs in the current design, use the `set_logic_dc`, `set_logic_one`, or `set_logic_zero` command respectively. Figure 6-3 shows an example of simplified input port logic.

  Figure 6-3: Simplified Input Port Logic



  The following example sets the A and B inputs to don't care and the IN input to logic 1:

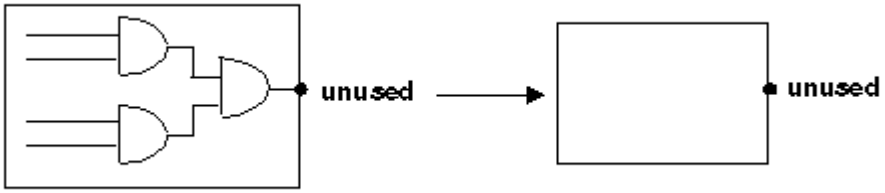
```
de_shell> set_logic_dc {A B}
de_shell> set_logic_one IN
```

  To reset the values set by these commands, use the `remove_attribute` command.

- Specifying unconnected output ports

  If an output is not used; that is, it is unconnected, the logic driving the output can be minimized or eliminated during optimization, as shown in Figure 6-4.

Figure 6-4: Minimizing Logic Driving an Unconnected Output Port



To specify outputs to be unconnected to outside logic, use the `set_unconnected` command. To reset the attribute set by this command, use the `remove_attribute` command.

Multivoltage Designs Using UPF

Using IEEE™ 1801 Unified Power Format (UPF) Standard commands, you can specify low-power intent for multivoltage designs and verification strategies for each power domain, including the domain's isolation cells, retention cells, and level shifters. During the compile, the tool automatically inserts these power management cells according to the specified strategies.

To use the UPF commands in DC Explorer, you must set the `de_enable_upf_exploration` variable to `false`, changing it from its default of `true`, and a Power Compiler license is required.

In multivoltage designs, the subdesign instances (blocks) operate at different voltages. To reduce power consumption, you specify power domains using the UPF commands. The blocks in a power domain can be powered up and down independent of the power states 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 shifters and isolation cells can be used 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 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

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 need to align the hierarchies to the voltage areas.

In addition to power domains, you specify power information, such as supply network, power strategies, power state tables, and operating voltages, for the power intent using the UPF commands.

To implement multivoltage designs using UPF, see

- The *Power Compiler User Guide*
- The *Synopsys Multivoltage Flow User Guide*

To run the RTL UPF design exploration using minimal UPF, see

- UPF Exploration

7 Defining Design Constraints

Constraints are declarations that define the design's goals in measurable circuit characteristics, such as timing, area, and capacitance. DC Explorer needs these constraints to effectively optimize the design.

To learn the concepts and tasks necessary for defining design constraints, see

- DC Explorer Constraint Types
- Defining Design Rule Constraints
- Disabling Design Rule Fixing on Special Nets
- Propagating Constraints in Hierarchical Designs

DC Explorer Constraint Types

DC Explorer optimizes designs by using the following two types of constraints:

- Design Rule Constraints

  The logic library defines these implicit constraints. These constraints are requirements for a design to function correctly, and they apply to any design that uses the library. By default, design rule constraints have higher priority than optimization constraints.

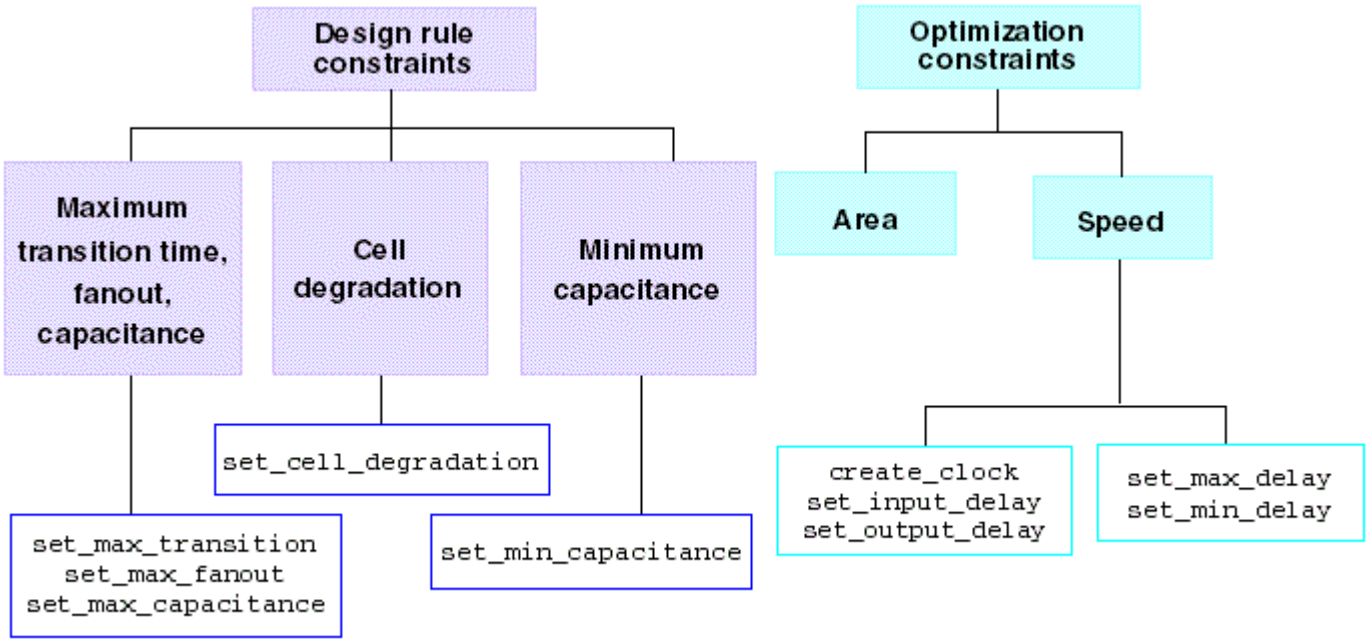
- Optimization Constraints

  You define these explicit constraints. Optimization constraints apply to the design on which you are working for the duration of the de_shell session and represent the design's goals.

You specify constraints interactively on the command line or in a constraint file. DC Explorer tries to meet both design rule constraints and optimization constraints, but design rule constraints take precedence over optimization constraints.

Figure 7-1 shows the major design rule constraints and optimization constraints for DC Explorer and the de_shell interface commands to set the constraints.

Figure 7-1: Major DC Explorer Constraints



See Also

- Constraint Priorities
- Precedence of Design Rule Constraints
- Generating Constraint Reports

Design Rule Constraints

Design rule constraints reflect technology-specific requirements that your design must meet to function as intended. Design rules constrain the nets of a design but are maintained as attributes associated with the pins of cells from the logic library. Most logic libraries specify default design rules.

These are the design rule constraint types:

- Maximum Transition Time
- Maximum Fanout
- Maximum Capacitance
- Minimum Capacitance
- Cell Degradation

You cannot remove the `max_transition`, `max_fanout`, `max_capacitance`, and `min_capacitance` attributes set in a logic library because they are the requirements for the logic library, but you can set stricter values that override the default values.

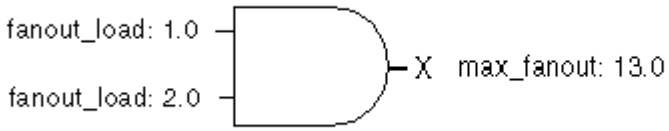
Maximum Transition Time

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 logic libraries contain restrictions on the maximum transition times for pins, creating implicit transition time constraints for designs using the libraries. If your design uses multiple logic libraries that contain a different default for the `max_transition` attribute, DC Explorer uses the strictest value globally.

Maximum Fanout

Maximum fanout is a design rule constraint. Many logic libraries contain fanout restrictions on driving pins, creating implicit fanout constraints for the driving pins in designs using these libraries. DC Explorer models fanout restrictions by associating the `fanout_load` attribute with each input pin and the `max_fanout` attribute with each output (driving) pin of a cell, as shown in Figure 7-2.

Figure 7-2: The fanout_load and max_fanout Attributes



To evaluate the fanout for a driving pin, such as the X pin in Figure 7-2, DC Explorer

1. Calculates the sum of all the `fanout_load` attributes for inputs driven by the X pin.

   Fanout load is a dimensionless number, not a capacitance. It represents a numerical contribution to the total effective fanout.

2. Compares the specified value of the `max_fanout` attribute with the sum.

   If the sum is less than the specified value, the fanout constraint is met. Otherwise, DC Explorer attempts to meet the constraint by choosing a higher-drive component.

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 on the pin.

DC Explorer 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, DC Explorer compares the calculated capacitance value with the value of the `max_capacitance` attribute of the pin driving the net.

Minimum Capacitance

Minimum capacitance is a design rule constraint. Some logic libraries specify minimum capacitance. The minimum capacitance constraint specifies the minimum load a cell can drive. During optimization, DC Explorer ensures that the load driven by a cell meets the minimum capacitance constraint for that cell. When a violation occurs, DC Explorer fixes the violation by sizing the driver. 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 violations by recompiling the module driving the ports.

Cell Degradation

Cell degradation is a design rule constraint. Some logic libraries contain cell degradation tables. The tables list the maximum capacitance that a cell can drive as a function of the transition times at the inputs of the cell. During compilation, if cell degradation tables are specified in the logic library, DC Explorer tries to ensure that the capacitance value for a net is less than the specified value. If cell degradation tables are not specified in the logic library, you can set the `cell_`

`degradation` attribute explicitly on the input ports by using the `set_cell_degradation` command. By default, a port has no `cell_degradation` attribute.

See Also

- Constraint Priorities
- Precedence of Design Rule Constraints
- Defining Design Rule Constraints

Optimization Constraints

Optimization constraints represent timing and area goals and restrictions that you want but that might not be crucial to the operation of a design. Timing constraints have higher priority than area constraints. By default, optimization constraints are secondary to design rule constraints.

The optimization constraints consist of

- Input and output delays (timing constraints)
- Maximum and minimum delays (timing constraints)
- Maximum area

Defining Timing Constraints

When defining timing constraints, you should consider that your design might contain both synchronous paths and asynchronous paths.

- To constrain synchronous paths, use the `create_clock` command to specify a clock in the design. After specifying the clocks, you should also specify the input and output port timing specifications by using the `set_input_delay` and `set_output_delay` commands.
- To constrain asynchronous paths, use the `set_max_delay` and `set_min_delay` commands to specify the point-to-point minimum and maximum delay values.

Maximum Delay

Maximum delay is an optimization constraint. DC Explorer contains a built-in static timing analyzer for evaluating timing constraints. The static timing analyzer calculates path delays from local gates and interconnect delays based on the timing values and operating conditions specified in the logic library, but it does not simulate the design. The DC Explorer 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.

DC Explorer determines the maximum delay target values for each timing path in the design after considering clock waveforms and skew, library setup times, external delays, multicycle or false path specifications, and the values set by the `set_max_delay` command. Load, drive, operating conditions, and other factors are also taken into account.

Minimum Delay

Although minimum delay is an optimization constraint, DC Explorer fixes the minimum delay constraint violations when it fixes design rule violations. Minimum delay constraints are set explicitly with the `set_min_delay` command or set implicitly because of hold time requirements. The minimum delay to a pin or port must be greater than the target delay.

Maximum Area

Maximum area is an optimization constraint. Maximum area represents the number of gates in the design, not the physical area that the design occupies. Usually the area requirements for the design are stated as the smallest design that meets the performance goal. By default, DC Explorer optimizes the design automatically for area after timing optimization is complete. You cannot define the maximum area constraint for the design.

DC Explorer ignores the following components when it calculates circuit area:

- Unknown components
- Components with unknown areas
- Technology-independent generic cells

The area of a cell is technology-dependent and obtained from the logic library.

See Also

- *Synopsys Timing Constraints and Optimization User Guide*

Constraint Priorities

DC Explorer attempts to minimize the total constraint cost. Each type of constraint has a relative cost so that more effort is applied to meeting the more important constraints, possibly at the expense of other constraints. Table 7-1 shows the default order of priorities. By default, design rule constraints have higher priority than optimization constraints.

Table 7-1: Default Order of Constraint Priority

Priority (descending order)	Notes
connection classes	
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
area	Optimization Constraint
cell count	

When multiple design rule violations occur, DC Explorer fixes the violations, starting with the highest priority first. When possible, it also evaluates and selects alternatives that reduce violations of other design rules. DC Explorer uses the same approach for the optimization delay violations. DC Explorer can violate optimization constraints if necessary to avoid violating design rule constraints.

See Also

- Precedence of Design Rule Constraints
- Optimization Constraints
- Defining Design Rule Constraints
- Generating Constraint Reports

Precedence of Design Rule Constraints

DC Explorer resolves conflicts among design rule constraints by following this descending order of precedence.

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.
- DC Explorer calculates transition time for a net in two ways, depending on the library.

  - For libraries using the CMOS delay model, DC Explorer 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, DC Explorer 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.
- Depending on your logic library, the `set_driving_cell` command behaves differently.
  - For libraries using the CMOS delay model, drive resistance is a constant.
  - 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_driving_cell` command affects the port transition time. The command places the design rule constraint on the affected port.

The `set_load` command places a load on a port or net. The units of this load must be consistent with your logic library. This value is used for timing optimization, not for maximum fanout optimization.

See Also

- Constraint Priorities
- Defining Design Rule Constraints

Generating Constraint Reports

To check design rules and optimization goals, use the `report_constraint` command to generate the constraint report for the current design.

For each constraint in the current design, the constraint report lists:

- Whether the constraint is met or violated, and by how much
- The design object that is the worst violator

The constraint report summarizes the constraints in the order of priority. Constraints not present in your design are not included in the report.

Generating Maximum Capacitance Reports

To report maximum capacitance on violating nets only, use the `report_constraint` command as follows:

```
de_shell> report_constraint -all_violators -max_capacitance
...
                Required       Actual
Net            Capacitance   Capacitance      Slack
-------------------------------------------------------------
n_6            0.0014226     0.0128121     -0.0113895 (VIOLATED)
```

```
n_16            0.0014226     0.0115580     -0.0101354 (VIOLATED)
n_121           0.0014226     0.0114569     -0.0100343 (VIOLATED)
n_21            0.0014226     0.0113311     -0.0099085 (VIOLATED)
div_18_17/n_2   0.0014226     0.0044171     -0.0029945 (VIOLATED)
div_18_17/n_4   0.0014226     0.0042607     -0.0028381 (VIOLATED)
...
```

To report maximum capacitance on all nets, use the `report_net` command as follows:

```
de_shell> report_net -max_capacitance

...
Net:    Required Capacitance  Actual Capacitance   Slack
-------------------------------------------------------------
CMD[0]          1.25                  0.25           1.0(Met)
CMD[1]          2.50                  4.50          -2.0(Violated)
...
```

These commands report the following details about nets:

- Required capacitance defined by users or derived using the libraries
- Actual capacitance calculated by the tool
- Slack (the required capacitance minus the actual capacitance)

See Also

- Constraint Priorities

Defining Design Rule Constraints

When defining design rule constraints, you should consider the following typical design rule scenarios:

- The `set_max_fanout` and `set_max_transition` commands
- The `set_max_fanout` and `set_max_capacitance` commands

Typically, a logic library specifies a default for either the `max_transition` or `max_capacitance` attribute, but not both. To achieve the best result, do not assign both `max_transition` and `max_capacitance` attributes to a design.

To learn how to set up design rule constraints, see

- Defining Maximum Transition Time
- Defining Maximum Fanout
- Defining Maximum Capacitance
- Defining Minimum Capacitance

- Defining Cell Degradation
- Design Rule Constraints
- Constraint Priorities
- Defining Connection Class

Defining Maximum Transition Time

To specify the maximum transition time on clock groups, ports, or designs, use the `set_max_transition` command to set the `max_transition` attribute. During compilation, DC Explorer ensures that the transition time for a net is less than the specified value, for example, by buffering the output of a driving gate.

For example, to set a maximum transition time of 3.2 units for the adder design, enter

```
de_shell> set_max_transition 3.2 [get_designs adder]
```

To reset the value set by the `set_max_transition` command, use the `remove_attribute` command:

```
de_shell> remove_attribute [get_designs adder] max_transition
```

If you set the `max_transition` attribute that is also defined in the library, DC Explorer tries to meet the stricter value.

Specifying Clock-Based Maximum Transition Time

The value of the `max_transition` attribute can vary with the operating frequency of a cell. This frequency is defined as the highest clock frequency of the registers driving a cone of logic. To set the `max_transition` attribute on pins in a specific clock group for a design containing multiple clock frequencies, use the `set_max_transition` command.

For example, the following command sets the `max_transition` attribute to a value of 5 units on all pins in the Clk clock group:

```
de_shell> set_max_transition 5 [get_clocks Clk]
```

DC Explorer follows these rules in determining the value of the `max_transition` attribute:

- When the `max_transition` attribute is set on a design or port and a clock group, the strictest constraint is honored.
- If multiple clocks launch the same path, the strictest constraint is honored.
- If the `max_transition` attributes are already specified in a logic library, DC Explorer tries to meet these constraints during compilation.

See Also

- Design Rule Constraints
- Constraint Priorities

Defining Maximum Fanout

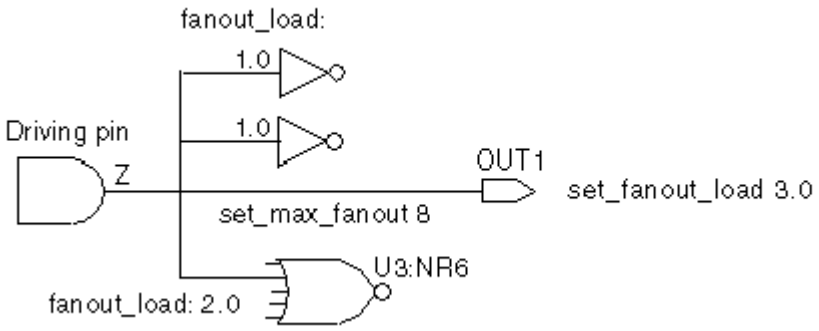
To specify the maximum fanout load for one or more input ports, use the `set_max_fanout` command to set the `max_fanout` attribute. You can set a more conservative fanout constraint on the entire library or define fanout constraints for specific pins of an individual cell in the library. If you specify a `max_fanout` attribute that is also defined in the library, DC Explorer tries to meet the stricter value.

For example, to set a maximum fanout constraint on every driving pin and input port in a design called ADDER, enter

```
de_shell> set_max_fanout 8 [get_designs ADDER]
```

Figure 7-3 shows a maximum fanout constraint is applied to a design.

Figure 7-3: Calculation of Maximum Fanout



To check whether the Z driving pin meets the maximum fanout constraint, DC Explorer compares the specified value of the `max_fanout` attribute with the fanout load. In this example, the total fanout load is 1.0 + 1.0 + 3.0 + 2.0 = 7.0, which is less than 8; therefore, this net meets the constraint.

The fanout load imposed by a driven cell (U3) is not necessarily 1.0. Libraries can define higher fanout loads to model internal cell fanout effects. You can also set a fanout load on an output port (OUT1) to model external fanout effects.

To reset a maximum fanout value set on an input port or design, use the `remove_attribute` command. For example,

```
de_shell> remove_attribute [get_ports port_name] max_fanout
de_shell> remove_attribute [get_designs design_name] max_fanout
```

Defining Expected Fanout for Output Ports

To specify the external fanout load on an output port, use the `set_fanout_load` command to set the `fanout_load` attribute. DC Explorer adds the fanout load value to all other loads on the pin driving each port specified and tries to make the total load less than the maximum fanout load of the pin.

For example, the following command sets a fanout load of 2 units to all outputs:

```
de_shell> set_fanout_load 2 [all_outputs]
```

To reset the value set by the `set_fanout_load` command on an output port, use the `remove_attribute` command. For example,

```
de_shell> remove_attribute port_name fanout_load
```

To determine the fanout load, use the `get_attribute` command. The following example finds the fanout load on the input pin of the AND2 library cell in the libA library:

```
de_shell> get_attribute "libA/AND2/i" fanout_load
```

To find the default fanout load set on the logic library libA, enter

```
de_shell> get_attribute libA default_fanout_load
```

See Also

- Design Rule Constraints
- Constraint Priorities

Defining Maximum Capacitance

To specify a maximum capacitance for the nets attached to named ports or for all nets, use the `set_max_capacitance` command to set the `max_capacitance` attribute. You can 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 value of the `max_capacitance` attribute of the pin driving the net.

For example, to set a maximum capacitance of 3 units for the adder design, enter

```
de_shell> set_max_capacitance 3 [get_designs adder]
```

To reset the value set by the `set_max_capacitance` command, use the `remove_attribute` command:

```
de_shell> remove_attribute [get_designs adder] max_capacitance
```

See Also

- Design Rule Constraints
- Constraint Priorities

Defining Minimum Capacitance

To specify a minimum capacitance for nets attached to input or bidirectional ports, use the `set_min_capacitance` command to set the `min_capacitance` attribute. If a library `min_capacitance` attribute and an input port `min_capacitance` attribute both exist, DC Explorer tries to meet the stricter value. By default, the minimum capacitance constraint has higher priority than the maximum transition time, maximum fanout, and maximum capacitance constraints.

The following example sets a minimum capacitance value of 12.0 units on the high_drive port:

```
de_shell> set_min_capacitance 12.0 high_drive
```

To report only minimum capacitance constraint information, use the `-min_capacitance` option with the `report_constraint` command. To get information about the current port settings, use the `report_port` command. To reset the settings of the `set_min_capacitance` command, use the `remove_attribute` command.

See Also

- Design Rule Constraints
- Constraint Priorities

Defining Cell Degradation

To set the `cell_degradation` attribute explicitly on input ports, use the `set_cell_degradation` command. The `cell_degradation` attribute specifies that the capacitance value for a net is less than the cell degradation value.

You can use the cell degradation constraint with other constraints, but the maximum capacitance constraint has higher priority than the cell degradation constraint.

Note: Use of the `set_cell_degradation` command requires a DC Ultra license.

The following example sets a maximum capacitance value of 2.0 units on the late_riser port:

```
de_shell> set_cell_degradation 2.0 late_riser
```

To report only cell degradation constraint information, use the `-cell_degradation` option with the `report_constraint` command. To remove the `cell_degradation` attribute, use the `remove_attribute` command.

See Also

- Design Rule Constraints
- Constraint Priorities

Defining 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. To set a connection class constraint on a port, use the `set_connection_class` command.

For example, the following example sets the connection class called “internal” on the xyz port in the current design:

```
de_shell> set_connection_class internal xyz
```

See Also

- Design Rule Constraints
- Constraint Priorities

Disabling Design Rule Fixing on Special Nets

You can enable or disable design rule fixing on clock, constant, or scan nets by using the `set_auto_disable_drc_nets` command. You can do this for all nets of a given type in the current design, such as all clock nets, all constant nets, all scan nets, or any combination of these three net types. Nets that are set for disabled design rule fixing are marked with the `auto_disable_drc_nets` attribute. By default, the tool disables design rule fixing on clock and constant nets, but not on scan nets.

Note: 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 change 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 the `ideal_networks` attribute. This command never overrides the settings specified by the `set_ideal_net` or `set_ideal_network` command.

See Also

- Design Rule Constraints
- Defining Design Rule Constraints

Propagating Constraints in Hierarchical Designs

Hierarchical designs consist of subdesigns. You can propagate constraints up or down the hierarchy using the following methods:

- Characterizing Subdesigns

  The characterizing method 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

  The modeling method creates a characterized design as a library cell.

- Propagating Constraints Up the Hierarchy

  This method 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 by using the `set_driving_cell`, `set_input_delay`, `set_output_delay`, and `set_load` commands.

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`), characterization removes the annotations and cannot overwrite the existing annotations made on the subdesign. In this case, you must explicitly remove annotations from the subdesign by using the `reset_design` command before characterizing the design again.

Characterizing Port Signal Interfaces of Subdesigns

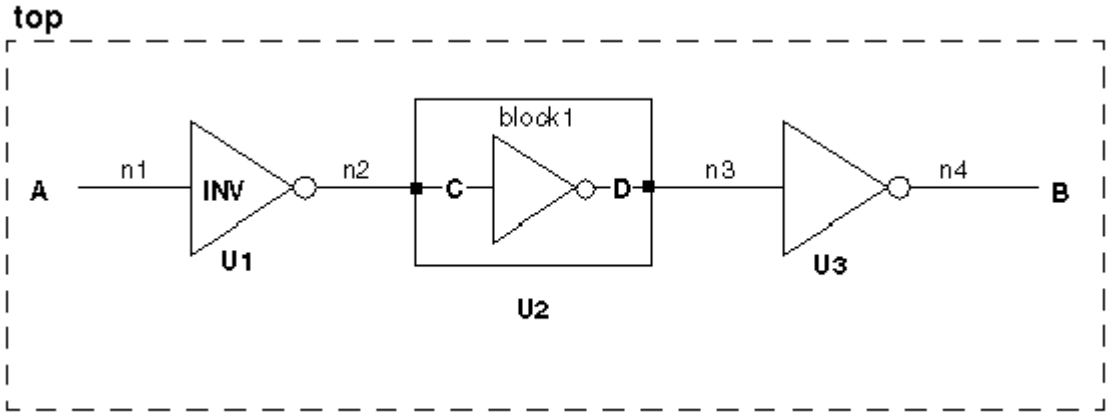
You can manually set port signal information by using the following commands:

```
create_clock
set_clock_latency
set_clock_uncertainty
set_driving_cell
set_input_delay
set_load
set_max_delay
set_min_delay
set_output_delay
set_propagated_clock
```

Combinational Design Example

Figure 7-4 shows a subdesign block in a combinational design called top.

Figure 7-4: Characterizing Drive, Timing, and Load Values of a Combinational Design



The following script shows an example of how to set the port interface attributes manually:

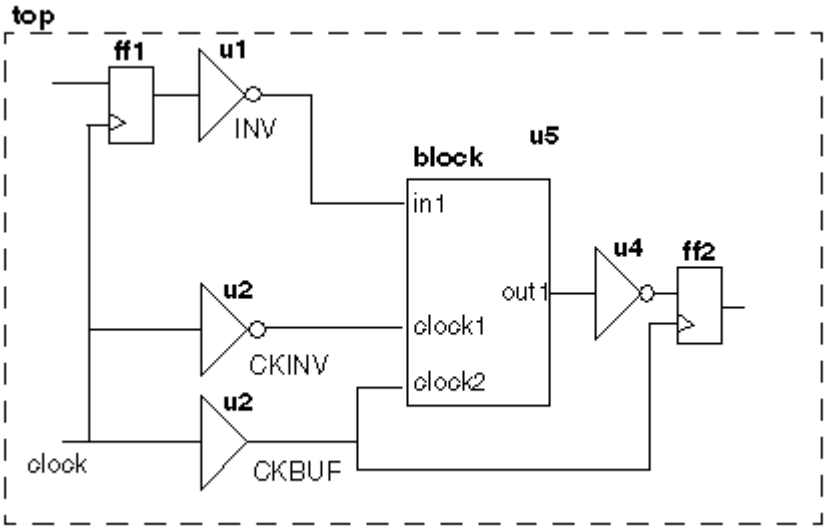
```
current_design top
set_input_delay 0 A
set_max_delay 10 -to B
current_design block1
set_driving_cell -lib_cell INV {C}
set_input_delay 3.3 C
set_load 1.3 D
set_max_delay 9.2 -to D
current_design top
```

The script sets the arrival time of the n2 net to 3.3 units, the load of U3 to 1.3 units, and the inherited maximum delay to 10 - 0.8 = 9.2 units (0.4 for each inverter).

Sequential Design Example

Figure 7-5 shows a subdesign block in a sequential design named top.

Figure 7-5: Characterizing Sequential Design Drive, Timing, and Load Values



The following script shows an example of how to set the port interface information manually:

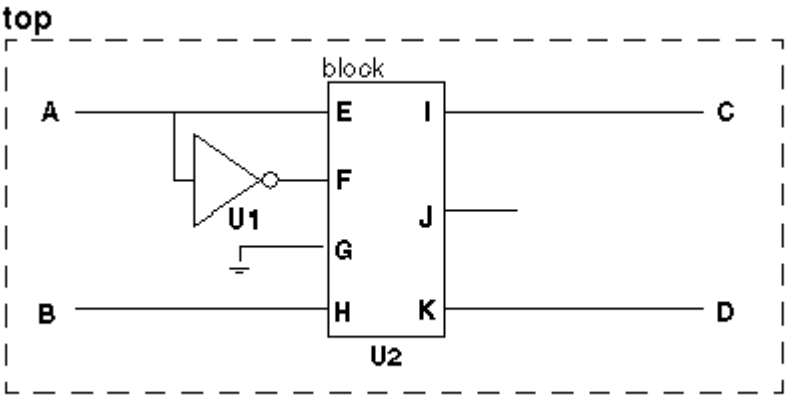
```
current_design top
create_clock -period 10 -waveform {0 5} clock
current_design block
create_clock -name clock -period 10 -waveform {0 5} clock1
create_clock -name clock_bar -period 10 \
-waveform {5 10} clock2
set_input_delay -clock clock 1.8 in1
set_output_delay -clock clock 1.2 out1
set_driving_cell -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
```

This script sets the input delay from ff1/CP to u5/in1 to 1.8 units, the output delay through u4 plus the setup time of ff2 to 1.2 units, and the outside load on the out1 net to 0.85 units.

Example of Characterizing Logical Port Connections of Subdesigns

Figure 7-6 shows a subdesign block in a design called top.

Figure 7-6: Characterizing Port Connection Attributes



You can set logical port connections manually by using the following script:

```
current_design block
set_opposite    { E F }
set_logic_zero  { G }
set_unconnected { J }
```

Propagating Constraints Up the Hierarchy

For hierarchical designs, if you compile the lower-level blocks and then the higher-level blocks (bottom-up compilation), you can propagate clocks, timing exceptions, and disabled timing arcs from the lower-level .ddc files to the current design by using the `propagate_constraints` command. The command propagates constraints from lower-level blocks to the current design. If you do not use this command, you cannot propagate constraints set on a lower-level block to a higher-level block in which it is instantiated.

Note: Using the `propagate_constraints` command might increase memory usage.

The following script shows the methodology to propagate constraints upward. Assume that A is the top-level block and B is the lower-level block.

```
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 are propagated up, use the `-verbose` and `-dont_apply` options and redirect the output to a file. For example,

```
de_shell> propagate_constraints \
-design name -verbose -dont_apply -output report.cons
```

Use the `-format ddc` option to save the .ddc file with the propagated constraints, so you do not need to repeat the propagating step when restarting a new de_shell session.

Conflicts Between Designs

The following rules apply when conflicts between lower-level and top-level blocks occur:

- Clock name conflicts

  When a lower-level clock has the same name as the clock of the current design or another block, the clock is not propagated. DC Explorer issues a warning.

- Clock source conflicts

  When a clock source of a lower-level block is already defined as a clock source of a higher-level block, the lower-level block is not propagated. DC Explorer issues a warning.

- Exceptions from or to a clock that is not propagated

  This can be either a virtual clock or a clock that is not propagated from that block because of a conflict.

- Exceptions

  A lower-level exception overrides a higher-level exception that is defined on the same path.

8 UPF Exploration

DC Explorer accepts minimal UPF and automatically generates isolation strategies and level shifter strategies based on the power state table and domain supply voltages. You can use the RTL UPF generated by DC Explorer for floorplanning, synthesis, and physical implementation.

To run the UPF exploration flow, ensure that the `de_enable_upf_exploration` variable is set to `true` (the default).

The following topics describe the UPF exploration flow:

- UPF Exploration Flow Overview
- Minimal UPF File
- Exploration Synthesis With UPF
- Reporting Power Exploration Results

For information about UPF multivoltage design implementation, see the *Power Compiler User Guide*.

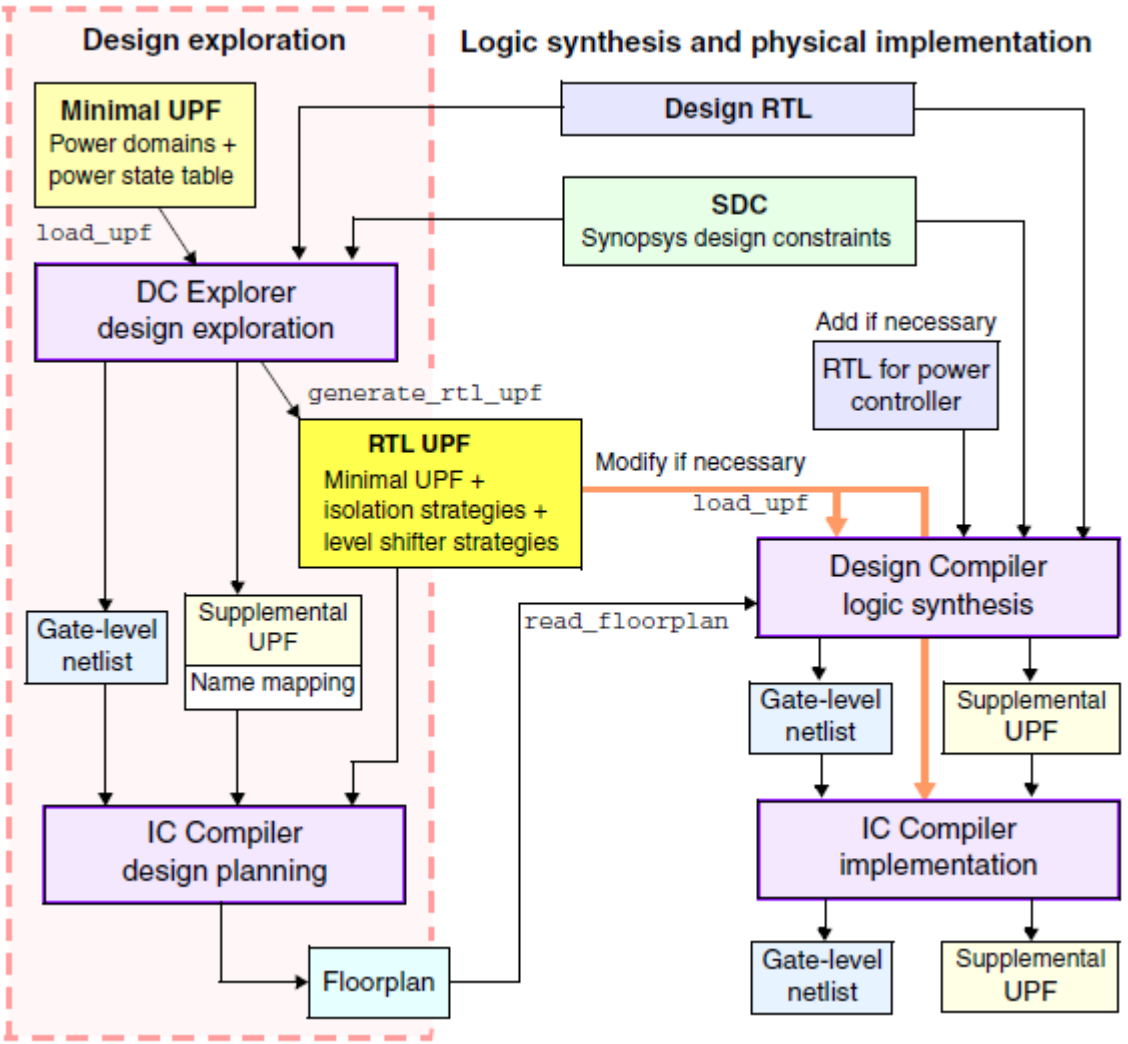
See Also

- SolvNet article 1755326, "UPF in DC Explorer Application Note"
- Multivoltage Designs Using UPF
- The *Synopsys Multivoltage Flow User Guide*

UPF Exploration Flow Overview

Using a minimal UPF file containing only power domain and power state definitions, DC Explorer generates a complete RTL UPF file that contains isolation and level shifter strategies, using RTL names in the added strategies, as well as a floorplan. You can use this file as the golden UPF file for synthesis and implementation in the Design Compiler and IC Compiler tools. You can also modify this file to change the default isolation and level shifter strategies created by the tool or to add retention strategies. The flow is summarized in Figure 8-1.

Figure 8-1: Design Exploration, Synthesis, and Implementation Flow With UPF



To learn how to run the UPF exploration flow, see

- Design Exploration With UPF
- Running Design Exploration With UPF
- Isolation and Level Shifter Insertion

Design Exploration With UPF

Specifying the UPF power intent for the DC Explorer tool is different from specifying the UPF power intent for the Design Compiler or IC Compiler tool. For design exploration, you need to specify only the power domains using the `create_power_domain` command and the power states using the `add_power_state`, `create_pst`, and `add_pst_state` commands. This basic subset of the complete power intent of a design is called minimal UPF.

Using the minimal UPF, DC Explorer automatically generates isolation strategies and level shifter strategies required in the design based on the presence of power-down domains and supply voltage differences between domains. The tool writes out the modified UPF in a complete form that can be used for specifying the power intent of the design for the Design Compiler and IC Compiler tools.

Input Data Used by DC Explorer

These are the types of input data used by DC Explorer in the UPF flow:

- Design RTL – The RTL for the exploration flow should not contain any power management cells (isolation, level shifter, and retention cells) or control signals for these cells. If any such cells are present, they are treated as ordinary standard cells and their power management functions are ignored.
- Synopsys design constraints – The SDC file must contain the `set_voltage` commands to set the voltage of each supply net for the power domains and the `set_operating_conditions` command to set the top-level operating condition.
- Minimal UPF – The minimal UPF file must contain commands to define the power domains and power states but should not contain isolation, level shifter, or retention commands (which are ignored if present).

Output Data for Synthesis and Physical Implementation

The design exploration flow with UPF provides the following data for the synthesis and physical implementation:

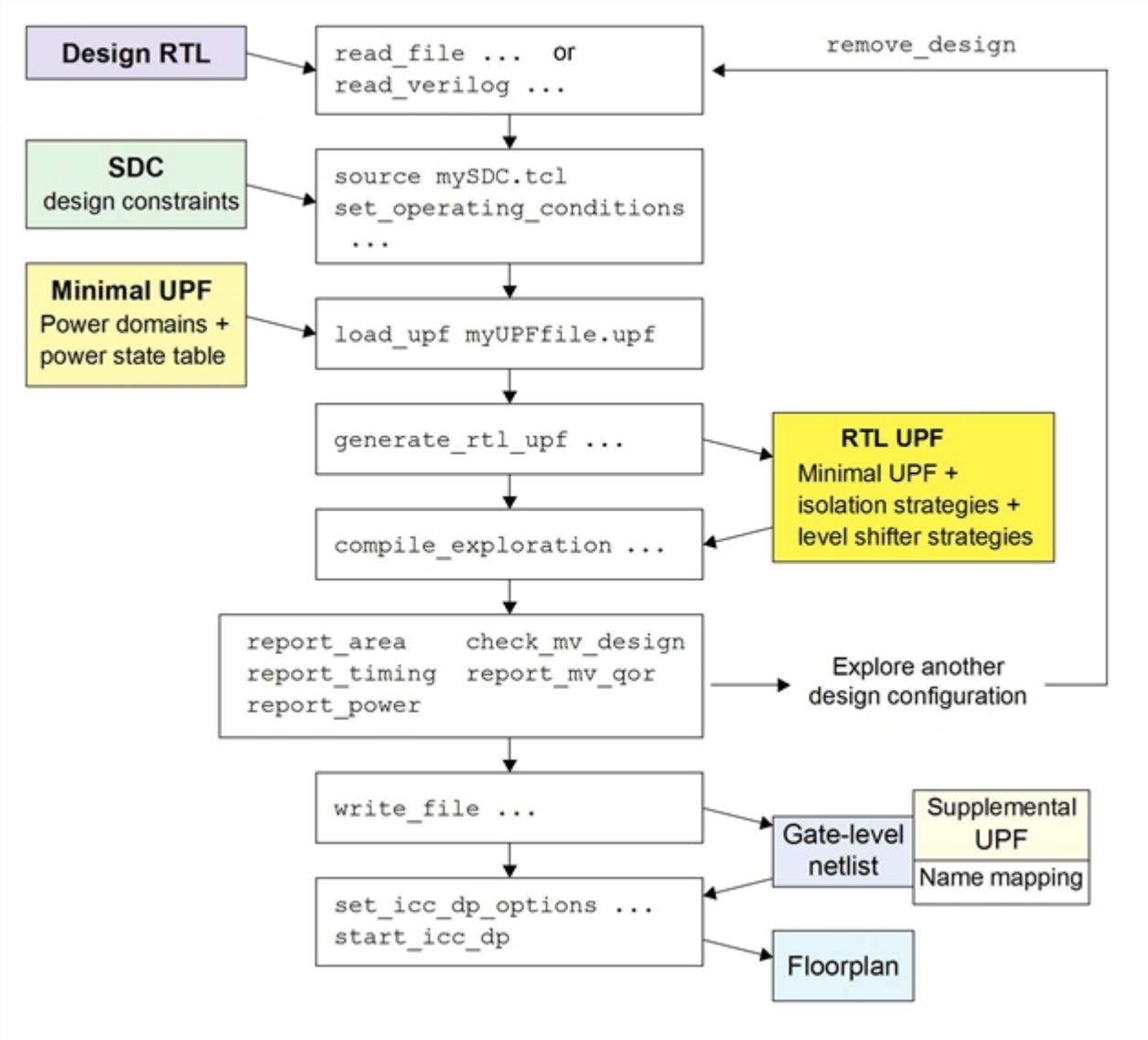
- RTL UPF – You can use the RTL UPF as the golden UPF for the synthesis and physical implementation flow. You can modify the RTL UPF as needed. For example, the exploration flow uses “location self” for isolation strategies. You can change this to “location parent” if that is what you prefer.

- Floorplan – You can use the floorplan to guide synthesis in Design Compiler topological mode and physical implementation in the IC Compiler tool.

The exploration flow automatically creates the isolation control signals and optionally creates a dummy controller block (black box) to drive these signals. To start logic synthesis, you need to provide the real controller logic in RTL format to replace the dummy controller block. The controller logic enables and disables the isolation cells according to the current power-down states of the domains in the design. You can edit the control signals in the RTL UPF.

The general exploration flow with UPF is summarized in Figure 8-2.

Figure 8-2: Design Exploration Flow With UPF



See Also

- Running Design Exploration With UPF
- Isolation and Level Shifter Insertion

Running Design Exploration With UPF

Follow these steps to run the general design exploration flow with UPF.

1. Read in the RTL description of the design that should not include any power management cells, control signals for such cells, or power controller circuit.
2. Read in the Synopsys design constraints (SDC file), including the `set_voltage` command, and set the top-level operating condition.
3. Read in the minimal UPF file, which should not contain any isolation, level shifter, or retention strategy, by using the `load_upf` command.
4. Write out the full UPF design intent (RTL UPF) by using the `generate_rtl_upf` command.

   DC Explorer automatically generates isolation strategies based on the presence of power-down domains with outputs that feed into active domains. It also automatically generates level-shifting strategies based on adjacent power domains having different supply voltages. (The tool does not generate or implement retention strategies.)

   If basic logic gates with the proper operating voltage are not available for linking, the tool links to cells having a different operating voltage.
5. Compile the design by using the `compile_exploration` command.

   DC Explorer inserts isolation, level shifter, and always-on buffer cells according to the full power intent of the RTL UPF file.
6. Report the area, timing, and power of the design.

   You can repeat steps 1 through 6 using a new minimal UPF file to explore other possible power supply infrastructures.
7. Write out the gate-level netlist, including PG connections, by using the `write_file` command.
8. Write out a supplemental UPF file by using the `save_upf -supplemental` command. The supplemental UPF file contains objects that were renamed during design optimization.
9. Perform floorplan exploration and generate a floorplan for the design as described in Performing Floorplan Exploration.

Note: Power management cells can be inserted only by the `compile_`

`exploration` command. The `insert_mv_cells` command is not supported.

Although DC Explorer does not support incremental compile, you can read a compiled design in .ddc format, including the power management cells, back into the tool for analysis.

Upon completion of the exploration flow, you can use the generated RTL UPF and floorplan for logic synthesis and physical implementation.

See Also

- Design Exploration With UPF
- Isolation and Level Shifter Insertion

Isolation and Level Shifter Insertion

The following example shows how DC Explorer inserts the power management cells in the exploration flow.

In the minimal UPF file, you specify only the power domains and power state table information as follows:

```
create_power_domain PD1 -elements U1
create_power_domain PD2 -elements U2
create_power_domain PD3 -elements U3

add_power_state PD1.primary -state HIGH12 \
  {-supply_expr {power == `{FULL_ON, 1.2}}}
add_power_state PD1.primary -state OFFX \
  {-supply_expr {power == `{OFF}}}
add_power_state PD1.ground -state GND \
  {-supply_expr {ground == `{FULL_ON, 0.00}}}

add_power_state PD2.primary -state HIGH12 \
  {-supply_expr {power == `{FULL_ON, 1.2}}}
add_power_state PD2.primary -state LOW08 \
  {-supply_expr {power == `{FULL_ON, 0.8}}}
add_power_state PD2.primary -state OFFX \
  {-supply_expr {power == `{OFF}}}
add_power_state PD2.ground -state GND \
  {-supply_expr {ground == `{FULL_ON, 0.00}}}

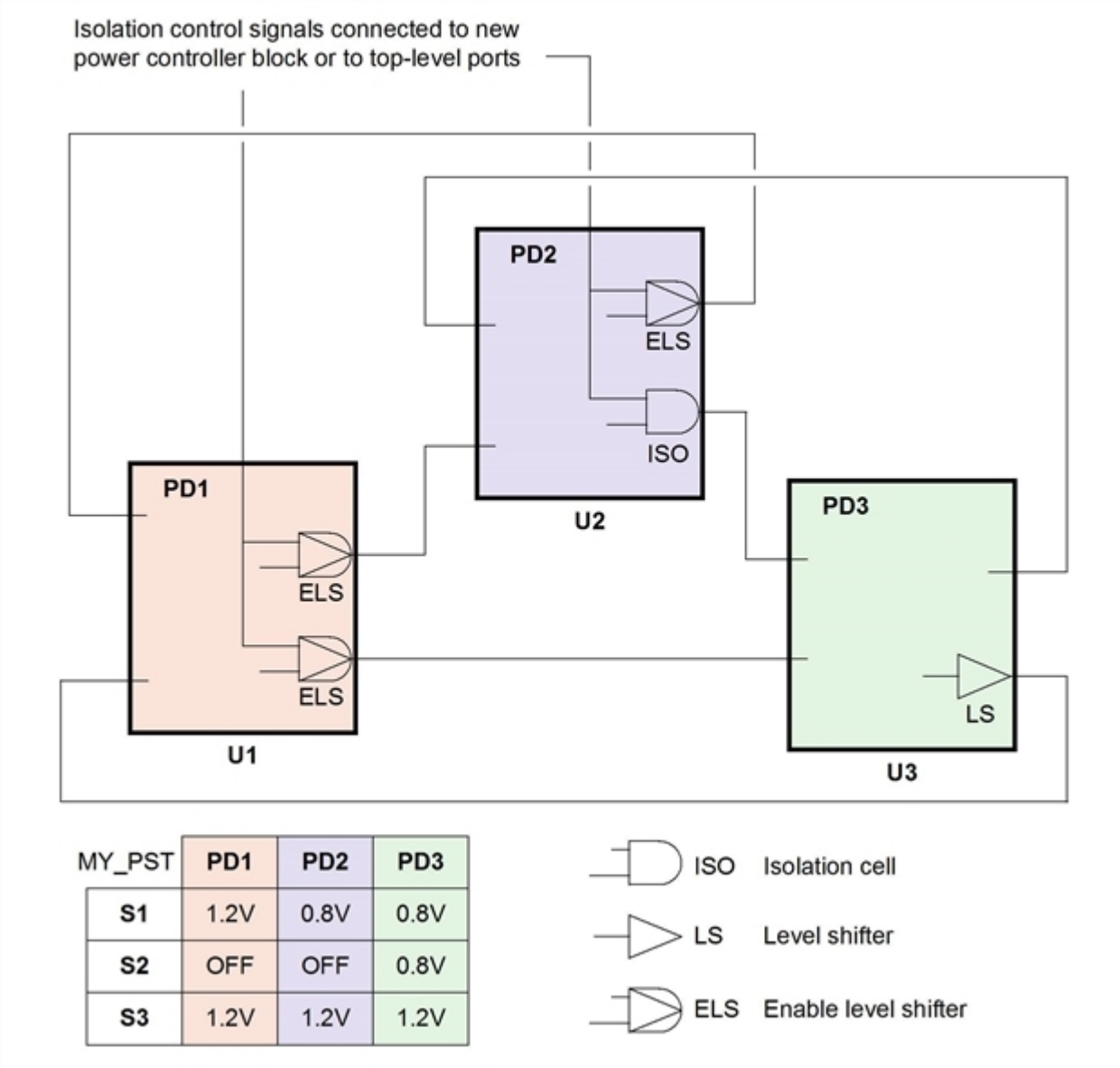
add_power_state PD3.primary -state HIGH12 \
  {-supply_expr {power == `{FULL_ON, 1.2}}}
add_power_state PD3.primary -state LOW08 \
  {-supply_expr {power == `{FULL_ON, 0.8}}
add_power_state PD3.ground -state GND \
  {-supply_expr {ground == `{FULL_ON, 0.00}}}
```

```
create_pst MY_PST -supplies \
  {PD1.primary.power PD2.primary.power PD3.primary.power}
add_pst_state S1 -pst MY_PST -state {HIGH12 LOW08  LOW08 }
add_pst_state S2 -pst MY_PST -state {OFFX   OFFX   LOW08 }
add_pst_state S3 -pst MY_PST -state {HIGH12 HIGH12 HIGH12}
```

After you read in the RTL design, the SDC constraints, and the minimal UPF file, use the `generate_rtl_upf` command to automatically write out the full RTL UPF for the design, including isolation and level shifter strategies.

When you use the `compile_exploration` command, DC Explorer follows the generated strategies and inserts the power management cells shown in Figure 8-3.

Figure 8-3: Isolation and Level Shifter Cells Inserted by DC Explorer



MY_PST	PD1	PD2	PD3
S1	1.2V	0.8V	0.8V
S2	OFF	OFF	0.8V
S3	1.2V	1.2V	1.2V

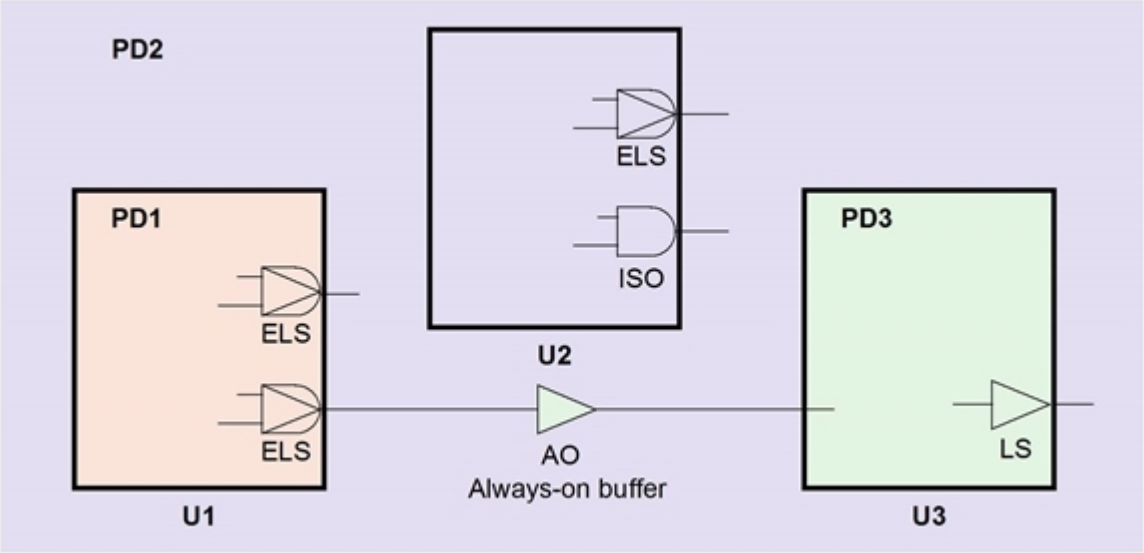
The tool inserts an isolation cell at each output from a domain that can be shut down and a level shifter where an output of the driver domain can operate a different voltage from the receiver domain. If an output requires both isolation and level shifter functions, the tool inserts an enable level shifter.

In certain situations, the insertion of isolation cells on all outputs of shutdown domains can result in isolation cells that are not actually needed. For example, the isolation function of the ELS cell inserted in the U2 block in Figure 8-3 is not needed because the driver domain PD2 is shut down only when the receiver domain PD1 is also shut down.

If a signal must pass from one power domain to another through a third power domain and a buffer is needed on the connection, the tool inserts an always-on buffer on the net, as shown in Figure 8-4. In this example, the always-on buffer uses the power supply from domain PD3, the domain of the signal receiver.

```
create_power_domain PD1 -elements U1
create_power_domain PD2 -include_scope
create_power_domain PD3 -elements U3
```

Figure 8-4: Always-On Buffer Inserted by the DC Explorer Tool



See Also

- Running Design Exploration With UPF
- Design Exploration With UPF

Minimal UPF File

By specifying only the minimal UPF in the design exploration flow, you can quickly understand, analyze, and compare the impact of various power infrastructure decisions, such as whether to power down a region or use a different supply voltage. You can get early estimates of the area, power, and leakage for each domain and the whole system.

The minimal UPF file might contain the following items:

- Required Commands
- Optional Commands
- Ignored Commands
- Ignored Domain Dependency of Supply Nets

Required Commands

The minimal UPF requires the following items:

- The power domain definitions specified by the `create_power_domain` command
- The power states for the primary supplies of all power domains specified by the `add_power_state` or `add_port_state` command
- A power state table that defines all possible power states specified by the `create_pst` and `add_pst_state` commands

In addition to providing the UPF commands that specify the power domains, power states, and power state table, you also need to use the `set_voltage` command to specify the voltages of the primary supplies and the `set_operating_conditions` command to specify the top-level operating condition.

Optional Commands

The minimal UPF file might contain commands that specify supply nets, supply sets, supply ports, supply connections, and power switches. DC Explorer supports the following UPF commands, but using these commands is optional for design exploration:

```
associate_supply_set
connect_supply_net
create_power_switch
create_supply_net
create_supply_port
create_supply_set
load_upf
name_format
set_port_attributes
set_related_supply_net
set_scope
```

The `create_power_domain` command automatically creates a default isolation supply set handle, `default_isolation`, for the specified domain. DC Explorer uses this supply set handle for the isolation strategy in that domain, and it automatically generates the `set_voltage` command to set the voltages for this supply set. (In other tools, you can explicitly specify other supply sets or supply nets by using the `set_isolation` command.)

Specifying power states for the `default_isolation` handles is optional. If you do not specify these power states, the tool automatically generates them based on the power state table for the primary supplies.

The `-repeater_supply` option of the `set_port_attributes` command is ignored.

Ignored Commands

The minimal UPF file should not contain isolation, level shifter, or retention strategy commands. DC Explorer ignores the following UPF commands:

```
map_isolation_cell
map_level_shifter_cell
map_power_switch
map_retention_cell
set_isolation
set_isolation_control
set_level_shifter
set_retention
set_retention_control
```

The tool reads and accepts these commands but does not use them. When you generate the RTL UPF file using the `generate_rtl_upf` command, the tool writes out these commands as comment lines. For example, if the minimal UPF file contains this line:

```
set_isolation ISO1 -domain PD1
```

the tool writes out the following comment line in the RTL UPF file:

```
# set_isolation ISO1 -domain PD1
```

Ignored Domain Dependency of Supply Nets

In DC Explorer, automatic inference of isolation cells depends on supply set handles and implicit supply sets that are available in all power domains. The tool does not support domain-dependent supply sets, which are created with the `-domain` option of the `create_supply_net` command in the Design Compiler and IC Compiler tools.

For example, the minimal UPF file contains the following command that uses the `-domain` option to restrict the usage of supply net VDD1 to power domain PD1.

```
create_supply_net VDD1 -domain PD1
```

When you run the `generate_rtl_upf` command, DC Explorer writes out the `create_supply_net` command with the `-domain` option in a comment line followed by the modified command without the option setting in the RTL UPF file. For example,

```
#create_supply_net VDD1 -domain PD1
create_supply_net VDD1
```

Exploration Synthesis With UPF

In DC Explorer, the `generate_rtl_upf` command writes out a UPF file containing the complete design intent of the design, including isolation and level shifter strategies. This is called the RTL UPF file because it represents the power intent of the design using RTL names for automatically generated isolation and level shifter strategies.

The `compile_exploration` command performs logic synthesis of the design, including the following power management tasks: isolation insertion, level shifter insertion, and always-on buffer insertion. For more information about exploration synthesis with UPF, see the *UPF in DC Explorer Application Note*.

Note:

The design exploration flow does not implement retention strategies. Upon completion of the design exploration flow, you can add retention strategies to the RTL UPF that can be used in the Design Compiler and IC Compiler tools.

The well bias feature is not supported. This is the feature in the Design Compiler tool that implements well bias supplies and the primary supply, a feature enabled by setting the `enable_bias` design attribute to `true`.

In a bottom-up flow, when each block is being optimized, you need to define the boundary conditions for the block. For UPF, you specify the boundary conditions by using the `set_related_supply_net`, `set_port_attributes -driver_supply`, and `set_port_attributes -receiver_supply` commands. These commands specify the power supplies of the external drivers at the inputs and the external loads at the outputs of the block.

See Also

- SolvNet article 1755326, "UPF in DC Explorer Application Note"

Reporting Power Exploration Results

You can check the power supply infrastructure at any time by using the `check_mv_design` command. After you compile a proposed design with the `compile_exploration` command, you can get an estimate of the area, timing, and power by using the `report_area`, `report_timing`, and `report_mv_qor` commands.

To get the leakage power of the isolation cells or level shifters separately, use the `report_mv_qor` command. By default, the `report_mv_qor` command reports the total leakage power, area, and number of cells in all the power domains. When you specify the `-verbose` option, the command separately reports the leakage power for isolation cells, level shifters, always-on cells, macros, I/O pads, and standard cells.

Design Vision GUI

You can use Design Vision GUI (opened by the `gui_start` command) to view the UPF intent of the design and analyze the design problems. To select a GUI analysis tool, use the Power pull-down menu.

For more information about the GUI analysis features, see the *Design Vision User Guide*.

9 Using Floorplan Physical Constraints

The principal reason for using floorplan physical constraints is to improve timing correlation with the post-layout tools, such as IC Compiler, by considering these constraints during optimizations. You provide floorplan physical constraints to DC Explorer by using one of the following methods:

- Export the floorplan information in DEF files or a Tcl script from IC Compiler and import this information into DC Explorer
- Create these constraints manually

To learn how to create, import, reset, save, and report floorplan physical constraints, see

- Importing Floorplan Information
- Physical Constraints Overview
- Saving Physical Constraints
- Saving in ASCII Format for IC Compiler II
- Including Physical-Only Cells
- Relative Placement

Importing Floorplan Information

You can import floorplan information to DC Explorer by using one of the following methods:

- Using DEF files

  Export the floorplan information from IC Compiler by using the `write_def` command and import this information into DC Explorer by using the `extract_physical_constraints` command.

  - Using the write_def Command in IC Compiler
  - Reading DEF Information in DC Explorer

- Using a Tcl script

  Export the floorplan information from IC Compiler by using the `write_floorplan` command and import this information into DC Explorer by using the `extract_physical_constraints` command.

  - Saving the Floorplan Information in IC Compiler
  - Reading the Floorplan Script in DC Explorer

See Also

- Imported DEF Information
- Imported Physical Constraints

Using the write_def Command in IC Compiler

To improve timing, area, and power correlation between DC Explorer and IC Compiler, you can read your mapped DC Explorer netlist into IC Compiler, create a basic floorplan in IC Compiler, export this floorplan from IC Compiler, and read the floorplan back into DC Explorer.

You use the `write_def` command in IC Compiler to export floorplan information to a Design Exchange Format (DEF) file which you can read into DC Explorer. The following example uses the `write_def` command to write floorplan information to the my_physical_data.def file:

```
icc_shell> write_def -version 5.7 -rows_tracks_gcells -macro -pins \
-blockages -specialnets -vias -regions_groups -verbose \
-output my_physical_data.def
```

Reading DEF Information in DC Explorer

To import floorplan information from a DEF file, use the `extract_physical_constraints` command. This command imports floorplan information from the specified DEF file and applies this information to the design. The applied floorplan information is saved in the .ddc file and does not need to be reapplied when you read in the .ddc file in subsequent de_shell sessions.

The following command shows how to import physical constraints from multiple DEF files:

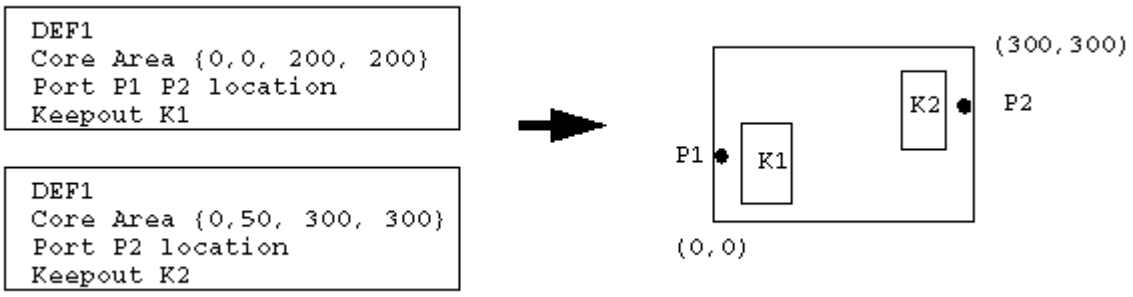
```
de_shell> extract_physical_constraints {des_1.def des2.def … des_N.def}
```

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 files. When incremental mode is disabled, the placement area is imported based on the site rows in the DEF files. 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 9-1 shows an example of incremental extraction performed by the `extract_physical_constraints` command.

Figure 9-1: Incremental Extraction with the extract_physical_constraints command




```
de_shell> extract_physical_constraints DEF1.def DEF2.def
```

To disable incremental mode, specify the `-no_incremental` option with the `extract_physical_constraints` command.

To learn about the imported DEF information and port and macro name matching considerations, see

- Imported DEF Information
- Name Matching of extract_physical_constraints
- Resolving Site Names Mismatches

Imported DEF Information

DC Explorer supports high-level floorplan physical constraints, such as die area, core area and shape, port location, macro location and orientation, keepout margins, placement blockages, preroutes, bounds, vias, tracks, voltage areas, and wiring keepouts:

- 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 `extract_physical_constraints` command supports automatic die area extraction from DEF files. DEF files are generated by floorplanning tools and used by DC Explorer to import physical constraints. The tool can extract both rectangular and rectilinear die areas that are defined in the DEF files. 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.

Example 9-1 shows a die area definition in a DEF file:

Example 9-1:

```
DEF
   UNITS DISTANCE MICRONS 1000 ;
   DIEAREA ( 0 0 ) ( 0 60000 ) ( 39680 60000 ) ( 39680 40000 ) \
           ( 59360 40000 ) ( 59360 0 ) ;
```

Example 9-2 shows how DC Explorer translates the DEF definition into Tcl when you write out your physical constraints using the `write_floorplan -all` command:

Example 9-2:

```
de_shell> 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, DC Explorer sets the location on the corresponding cell in the design. Example 9-3 shows DEF macro location and orientation information, where the letters E and W denote east rotation and west rotation respectively. The equivalent Tcl commands are shown in Example 9-4.

Example 9-3:

```
COMPONENTS 2 ;
    - macro_cell_abx2 + FIXED ( 4350720 8160 ) E ;
    - macro_cell_cdy1 + FIXED ( 4800 8160 ) W ;
END COMPONENTS
```

Example 9-4:

```
de_shell> set_cell_location macro_cell_abx2 \
-coordinates { 4350.720 8.160 } -orientation E -fixed
de_shell> 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.

Note: DEF versions before version 5.7 did not support partial blockages. In addition, 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 9-8.

Example 9-5 shows DEF hard placement blockage information, and Example 9-6 shows the equivalent Tcl command.

Example 9-5:

```
BLOCKAGES 50 ;
...
    - PLACEMENT RECT ( 970460 7500 ) ( 3247660 129940 )
...
END BLOCKAGES
```

Example 9-6:

```
de_shell> 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 9-7 (DEF version 5.7) or Example 9-8 (DEF version 5.6), Example 9-9 shows the equivalent Tcl command.

Example 9-7: DEF Version 5.7 Soft Placement Blockage Information

```
BLOCKAGES 50 ;
...
- PLACEMENT + SOFT RECT ( 970460 7500 ) ( 3247660 129940 ) ;
...
END BLOCKAGES
```

Example 9-8: DEF Version 5.6 Soft Placement Blockage Information

```
BLOCKAGES 50 ;
...
- PLACEMENT RECT ( 970460 7500 ) ( 3247660 129940 ) ; #SNPS_SOFT_BLOCKAGE

...
END BLOCKAGES
```

Example 9-9:

```
de_shell> create_placement_blockage -name def_obstruction_23 \
-bbox { 970.460 7.500 3247.660 129.940 } -type soft
```

For a partial placement blockage, if the extracted DEF information is as shown in Example 9-10, Example 9-11 shows the equivalent Tcl command.

Example 9-10:

```
BLOCKAGES 50 ;
...
- PLACEMENT + PARTIAL 80 RECT ( 970460 7500 ) ( 3247660 129940 ) ;
...
END BLOCKAGES
```

Example 9-11:

```
de_shell> 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 file, DC Explorer creates wiring keepouts on the design.

Example 9-12 shows DEF wiring keepout information.

Example 9-12:

```
BLOCKAGES 30 ;
...
    - LAYER METAL6 RECT ( 0 495720 ) ( 4050 1419120 );
...
END BLOCKAGES
```

Placement Bounds

If REGIONS defining bounds exist in the DEF file, the `extract_physical_constraints` command imports placement bounds. Also, if any cells in the related GROUP are attached to the

region, fuzzy cell matching occurs between these cells and 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 9-13 shows imported placement bounds information, and Example 9-14 shows the equivalent Tcl command.

Example 9-13:

```
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
```

Example 9-14:

```
de_shell> 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 file, DC Explorer sets the location on the corresponding port in the design.

Example 9-15 shows imported port location information, and Example 9-16 shows the equivalent Tcl commands.

Example 9-15:

```
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
```

Example 9-16:

```
de_shell> set_port_location Out1 -coordinate {80.875 0.000} \
-layer_name M3 -layer_area {0.000 0.000 4.200 0.200}
de_shell> 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 9-17 shows DEF information for such a case, and Example 9-18 shows the equivalent Tcl commands.

Example 9-17:

```
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
```

Example 9-18:

```
de_shell> set_port_location sys_addr[23] \
-coordinate { 1587.825 2744.180 } \
-layer_name METAL3 -layer_area {0.000 0.000 0.820 5.820}
de_shell> 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 9-19 shows DEF information for such a case, and Example 9-20 shows the equivalent Tcl command.

Example 9-19:

```
PINS 1;
    - OUT + NET OUT + DIRECTION INPUT + USE SIGNAL
        + LAYER m4 ( -120 0 ) ( 120 240 )
        + FIXED ( 4557120 1726080 ) S ;
END PINS
```

Example 9-20:

```
de_shell> set_port_location OUT -coordinate { 4557.120 1726.080 } \
-layer_name m4 -layer_area {-0.120 -0.240 0.120 0.000}
```

Preroutes

DC Explorer extracts preroutes that are defined in the DEF file.

Example 9-21 shows imported preroute information, and Example 9-22 shows the equivalent Tcl command.

Example 9-21:

```
SPECIALNETS 2 ;
- vdd
 + ROUTED METAL3 10000 + SHAPE STRIPE ( 10000 150000 ) ( 50000 * )
 + USE POWER ;
...
END SPECIALNETS
```

Example 9-22:

```
de_shell> create_net_shape -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

DC Explorer imports site array information that is defined in the DEF file. Site arrays in the DEF file define the placement area.

Example 9-23 shows imported site array information, and Example 9-24 shows the equivalent Tcl command.

Example 9-23:

```
ROW ROW_0 core 0 0 N DO 838 BY 1 STEP 560 0;
```

Example 9-24:

```
de_shell> 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 9-25 shows how DC Explorer translates the DEF definition into Tcl when you write out your physical constraints using the `write_floorplan -all` command.

Example 9-25:

```
de_shell> create_via -type via -net VDD -master VIA67 \
-route_type pg_strap -at {746.61 2279} -orient N
de_shell> create_via -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
```

Routing Tracks

The `extract_physical_constraints` command extracts any track information that is defined in the DEF file. Tracks define the routing grid for designs based on standard cells. They can be used during routing, and track support can enhance congestion evaluation and reporting in DC Explorer 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 types and locations, 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 9-26 shows how DC Explorer translates the DEF definition into Tcl when you write out your physical constraints using the `write_floorplan -all` command.

Example 9-26:

```
de_shell> create_track -layer metal1 -dir X -coord 0.100 -space 0.200 \
-count 11577 -bounding_box {{0.000 0.000} {2315.400 2315.200}}
de_shell> create_track -layer metal1 -dir Y -coord 0.200 -space 0.200 \
-count 11575 -bounding_box {{0.000 0.000} {2315.400 2315.200}}
```

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

- Reading DEF Information in DC Explorer
- Defining Physical Constraints

Name Matching of extract_physical_constraints

When the `extract_physical_constraints` command applies physical constraints, it matches 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_exploration` command followed by the `change_names` command might result in the forward slash (/) separator being replaced with an underscore (_). 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 the 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 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 disable intelligent name matching, you can specify the `-exact` option of the `extract_physical_constraints` command. Setting this option specifies to match objects in the netlist in memory exactly with the corresponding objects in the DEF file.

When you use the `-verbose` option with the `extract_physical_constraints` command, the tool displays the following informational message:

```
Information: Fuzzy match cell %s in netlist with instance
%s in DEF.
```

To define the rules used by the intelligent name matching capability, use the `set_query_rules` command.

See Also

- Resolving Site Names Mismatches

Resolving Site Names Mismatches

When you use the `extract_physical_constraints` command to read a DEF file, DC Explorer automatically matches the site name in the floorplan with 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 define the name mappings if the site dimension does not match unit or more than one site type is used in the DEF files. To define the name mappings, use the `mw_site_name_mapping` variable. For example,

```
de_shell> set mw_site_name_mapping \
{ {def_site_name1 mw_ref_lib_site_name} \
{def_site_name2 mw_ref_lib_site_name} }
```

You can specify multiple pairs of values for the `mw_site_name_mapping` variable.

See Also

- Name Matching of extract_physical_constraints

Saving the Floorplan Information in IC Compiler

To improve timing, area, and power correlation between DC Explorer and IC Compiler, you can read your mapped DC Explorer netlist into IC Compiler, create a basic floorplan in IC Compiler, export this floorplan from IC Compiler, and read the floorplan back into DC Explorer.

You can use the `write_floorplan` command in IC Compiler to export floorplan information and write this information in a Tcl script that you read into DC Explorer to re-create the floorplan information of the specified design. The following example uses the `write_floorplan` command to write out all placed standard cells to a file called placed_std.fp:

```
icc_shell> write_floorplan \
-placement {io terminal hard_macro soft_macro} -create_terminal \
-row -create_bound -preroute -track floorplan_for_DC.fp
```

Reading the Floorplan Script in DC Explorer

You use the `read_floorplan` command to read in a script file generated by the `write_floorplan` command that contains commands describing 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 DC Explorer. 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.

To visually inspect your imported physical constraints, use the layout view in the Design Vision layout window.

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. This command saves the floorplan information so that you can read the floorplan back into DC Explorer.

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.

To learn more about imported physical constraints, floorplan modification issues, and port and macro name matching considerations, see

- Imported Physical Constraints
- Name Matching of read_floorplan

Imported Physical Constraints

These are some of the physical constraints 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, DC Explorer sets the location on the corresponding cell in the design.
- Hard and Soft Placement Blockages

For defined placement blockages, DC Explorer 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:

  - 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.
  - 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, DC Explorer sets the location on the corresponding port in the design.

  Ports with changed names and multiple layers are supported.

- Preroutes

  DC Explorer imports preroutes that are defined in the floorplan file. A preroute is represented with the `create_net_shape` command.

- User Shapes

  DC Explorer imports user shapes that are defined in the floorplan file and includes them in the preroute section.

- Site Array Information

  DC Explorer extracts site array information that is defined in the floorplan file. Site arrays define the placement area.

- Vias

  DC Explorer imports vias that are defined in the floorplan file and includes them in the preroute section.

- Tracks

  DC Explorer imports any track information that is defined in the floorplan file.

Name Matching of read_floorplan

When the `read_floorplan` command applies physical constraints in topographical mode, it automatically 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. 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_exploration` 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 DC Explorer'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_query_rules` 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 `enable_rule_based_query` variable to `true` before you source the floorplan file. For example,

```
de_shell> set enable_rule_based_query true
de_shell> source design.fp
de_shell> set enable_rule_based_query false
```

Physical Constraints Overview

You manually define physical constraints when you cannot obtain this information from a DEF file or from the `read_floorplan` command. After you have manually defined your physical constraints in a Tcl script file, use the `source` command to apply these constraints. You should read in the design before applying user-specified physical constraints.

To learn how to manually define physical constraints, see

- Defining Physical Constraints
- Commands for Defining Physical Constraints

To report any physical constraints that are applied to the design, use the `report_physical_constraints` command. To reset all physical constraints, use the `reset_physical_constraints` command before reading in a new or modified floorplan. To report a collection of polygons that exactly cover the region computed by performing a Boolean operation on the input polygons, use the `compute_polygons` command.

See Also

- Reading DEF Information in DC Explorer
- Reading the Floorplan Script in DC Explorer

Defining Physical Constraints

If you do not provide any floorplan physical constraints using a floorplanning tool, DC Explorer 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)

You can also manually define physical constraints as shown in the following topics:

- Defining the Die Area
- Defining Placement Area
- Defining the Core Placement Area
- Defining Port Locations
- Defining Macro Locations
- Defining Placement Blockages
- Defining Voltage Area
- Creating Placement Bounds

- Creating Wiring Keepouts
- Creating Preroutes
- Creating User Shapes
- Defining Physical Constraints for Pins
- Creating Vias
- Creating Routing Tracks
- Creating Keepout Margins

See Also

- Physical Constraints Overview
- Placement Bounds Overview
- Saving Physical Constraints

Defining the Die Area

DC Explorer 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 DC Explorer. 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 the die area from within DC Explorer.

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:

```
de_shell> create_die_area -coordinate { 0 0 100 100 }
de_shell> create_die_area -coordinate { {0 0} {100 100} }
de_shell> 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 create a new die area by entering the following command:

```
de_shell> 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, specify the `object_class` attribute.

  ```
  de_shell> get_attribute [get_die_area] object_class
  die_area
  ```

- To return the bounding box information and the coordinates of the die area of the current design, specify the `bbox` attribute.

  ```
  de_shell> get_attribute [get_die_area] bbox
  {0.000 0.000} {400.000 400.000}
  ```

- To return the die area boundary, specify the `boundary` attribute.

  ```
  de_shell> 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, specify the `name` attribute.

  ```
  de_shell> get_attribute [get_die_area] name
  {my_design_die_area}
  ```

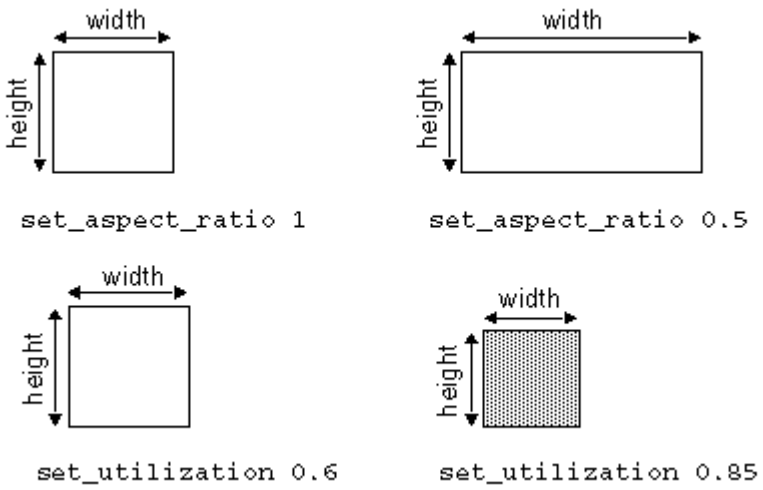
Defining Placement Area

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 9-2 illustrates how to use these commands.

Figure 9-2: Using the set_aspect_ratio and set_utilization Commands



Defining the Core Placement Area

Define the core placement area with the `create_site_row` command.

The core placement area is a box that contains all rows and 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 typically 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

```
de_shell> create_site_row-count 100 -kind CORE_2H \
-space 5 -coordinate {10 10}
```

Use the `report_area -physical` command to report floorplan information, such as core area, aspect ratio, utilization, total fixed cell area, and total movable cell area.

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

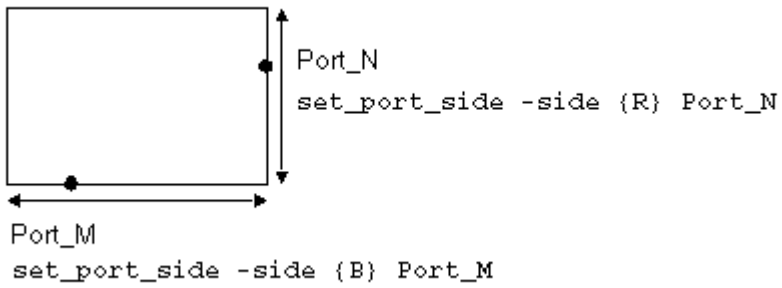
Defining Relative Port Locations

Use the `set_port_side` command to define relative port locations as follows:

```
de_shell> set_port_side port_name -side port_side
```

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 9-3 shows how you define port sides by using the `set_port_side` command.

Figure 9-3: Setting Relative Port Sides



Defining Exact Port Locations

Use the `set_port_location` command to annotate the port location on the specified port:

```
de_shell> set_port_location port_name -coordinate  {x y}
```

The following example annotates the Z port, a top-level port in the current design, with the location at 100, 5000.

```
de_shell> set_port_location -coordinate {100 5000} Z
```

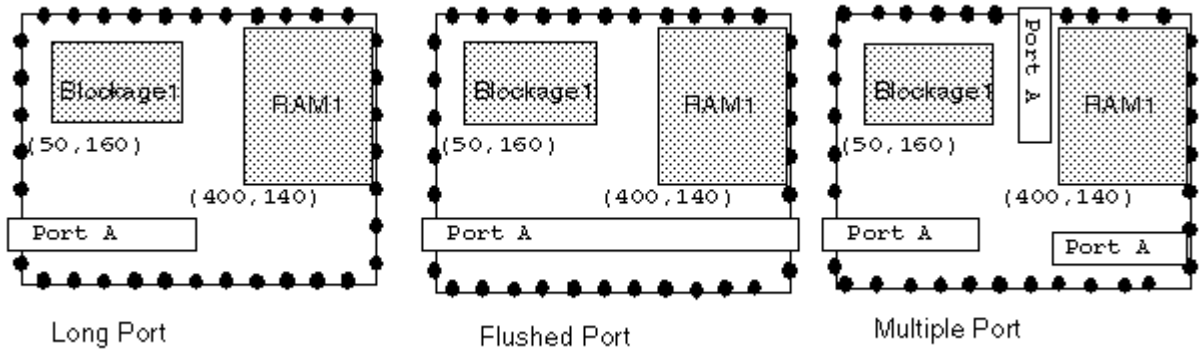
In addition, you can use the `-layer_name` option and `-layer_area` option to define metal layer geometry. For example, the following command specifies that the layer geometry is a rectangle of dimensions 10 by 20 for the Z port.

```
de_shell> 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 9-4 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 9-4: Types of Port Dimension Specifications



Defining Macro Locations

You can define exact macro locations by setting the following command:

```
de_shell> set_cell_location cell_name-coordinates{x y} -fixed \
-orientation orientation_value
```

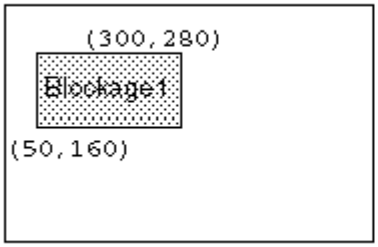
where `-coordinates` is the coordinate of the lower left corner of the cell's bounding box. The coordinate numbers are displayed in microns relative to the block orientation. The orientation value is one of the rotations: N (default), S, E, W, FN, FS, FE, and FW.

Defining Placement Blockages

Use the `create_placement_blockage` command to define placement blockages. The following example uses the `-bbox` option to specify the coordinates of the bounding box of the blockage and the `-type` option to specify the type of placement blockage to be created, as shown in Figure 9-5.

```
de_shell> create_placement_blockage -name Blockage1 \
-type hard -bbox {50 160 300 280}
```

Figure 9-5: Defining a Hard Placement Blockage



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, DC Explorer 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:

```
de_shell> 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, DC Explorer returns all placement blockages that have an area greater than 900:

```
de_shell> 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 9-27 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 9-27:

```
de_shell> create_voltage_area -name my_area \
-coordinate {100 100 200 200} INST_1
```

Example 9-28 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 9-28:

```
de_shell> create_voltage_area -power_domain PD1 \
-coordinate {100 100 200 200}
```

Voltage areas are automatically defined when you import your floorplan information from IC Compiler with the `read_floorplan` command. 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.

See Also

- Reading the Floorplan Script in DC Explorer

Creating Placement Bounds

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, you also reduce the solution space available to the placer to reach the optimal placement result.

You should create placement 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.

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.

Use the `create_bounds` command to specify placement bounds. You can specify the following types of placement bounds:

- Soft group bound

```
de_shell> create_bounds -dimension {100 100} -name temp1 INST1
```

- Soft move bound

```
de_shell> create_bounds -coordinate {0 0 10 10} -name temp2 INST2
```

- Hard group bound

```
de_shell> create_bounds -dimension {100 100} \
-type hard -name temp3 INST3
```

- Hard move bound

```
de_shell> create_bounds -coordinate {0 0 10 10} \
-type hard -name temp4 INST4
```

- Exclusive move bound

```
de_shell> create_bounds -coordinate {0 0 10 10} \
-type hard -exclusive -name temp5 INST5
```

To return a collection of bounds from the current design, use the `get_bounds` command. You can use the `get_bounds` command at the command prompt or nest it as an argument to another command, such as the `report_bounds` command. You can also 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.

The following example returns all bounds with the my_bound prefix:

```
de_shell> get_bounds my_bound*
```

See Also

- Placement Bounds Overview

Creating Wiring Keepouts

You can create wiring keepouts by using the `create_route_guide` command. The following example uses the `create_route_guide` command to create a keepout named my_keepout_1 in the METAL1 layer at coordinates {12 12 100 100}.

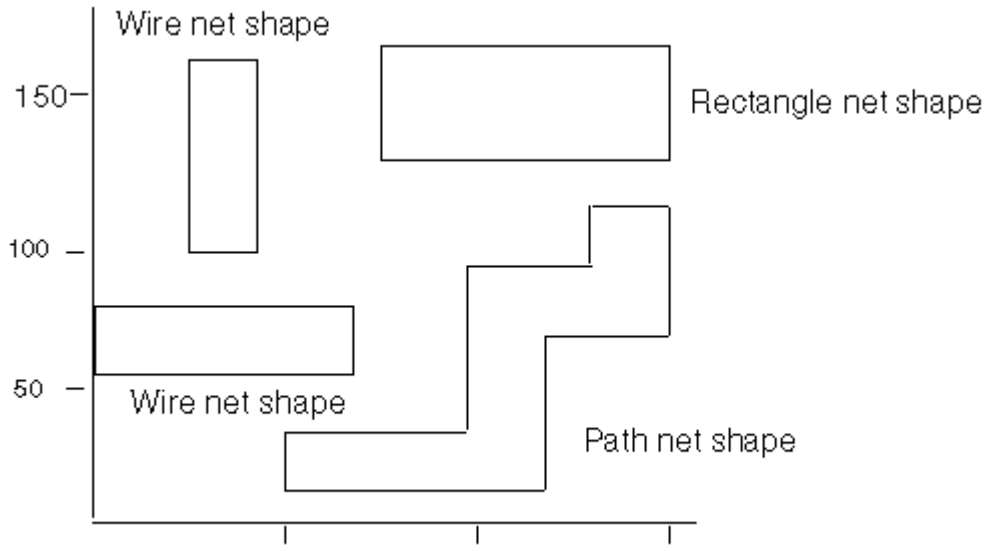
```
de_shell> create_route_guide -name "my_keepout_1" \
-no_signal_layers "METAL1"-coordinate {12 12 100 100}
```

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 9-6: path, wire (horizontal and vertical), and rectangle.

Figure 9-6: Types of Preroutes



The following three examples show how to create the net shapes shown in Figure 9-6.

Example 9-29 uses the `create_net_shape` command to create two wire net shapes.

Example 9-29:

```
de_shell> create_net_shape -type wire -net VSS \
-bbox {0 60 80 80} -layer METAL5 -route_type pg_strap \
-net_type ground
de_shell> create_net_shape -type wire -net VSS \
-origin {50 100} -width 20 -length 80 -layer METAL4 \
-route_type pg_strap -vertical
```

Example 9-30 uses the `create_net_shape` command to create the path net shape.

Example 9-30:

```
de_shell> 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 9-31 uses the `create_net_shape` command to create the rectangle net shape.

Example 9-31:

```
de_shell> create_net_shape -type rect -net VSS \
-bbox {{80 140} {160 180}} -layer METAL1 -route_type pg_strap
```

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, DC Explorer creates a wire user shape:

```
de_shell> 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 -type wire -layer METAL1 -datatype 0 \
-path_type 0 -width 2000 -route_type signal_route \
-length 3000 -origin {100 1500}
create_user_shape -type wire -layer METAL2 -datatype 0 \
-path_type 0 -width 2000 -route_type signal_route \
-length 3000 -origin {100 1500}
create_user_shape -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 -type rect -layer METAL5 -datatype 0 \
-route_type user_enter -bbox {{100 100} {600 600}}
create_user_shape -type poly -layer METAL6 -datatype 0 \
-boundary {{100 100}{300 200} {600 600} {400 400} {200 400} {100 100}}
create_user_shape -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 the asterisk (*) to indicate that all user shapes in the design should be removed, as shown in the following example:

```
de_shell> remove_user_shape *
```

You can use the layout window in DC Explorer 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.

See Also

- The “Examining Preroutes” topic in Design Vision Help

Defining Physical Constraints for Pins

You can use the `set_pin_physical_constraints` and the `create_pin_guide` commands to control the placement of I/O cells and terminals.

Note: DC Explorer 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 this command:

```
de_shell> 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 xyz prefix:

```
de_shell> create_pin_guide -bbox {{-20 50} {20 800}} \
[get_ports xyz*] -name abc
```

The `compile_exploration` command honors the constraints that you specify.

By default, DC Explorer 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 DC Explorer 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 DC Explorer 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.

DC Explorer does not pass the pin constraints to floorplan exploration, and floorplan exploration does not pass the pin constraints to DC Explorer.

See Also

- The “Examining Physical Constraints” topic in Design Vision Help

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:

```
de_shell> create_via -type via -net vdd -master via1 \
-route_type pg_strap -at {213 215} -orient E
```

The following example creates a via array with four columns and three rows, and the via master is via4:

```
de_shell> create_via -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
```

If a via is defined in the DEF file and it does not exist in the technology file, 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. To remove a via, use the `remove_via` command.

Note: The `write_floorplan` command cannot write out via master definitions to be used in a new session. The via master in the FRAM view is not passed to the new session; therefore, IC Compiler and DC Explorer cannot source the `create_via` commands that use undefined via masters in the technology file.

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.

See Also

- 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 tracks on a polygon layer does not support routing on the polygon layer.

The following example creates routing tracks for a routing layer named MET3:

```
de_shell> create_track -layer MET3 -dir Y -coord 0.000 -space 0.290 \
-count 4827 -bounding_box {{0.000 0.000} {1460.000 1400.000}}
de_shell> 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}:

```
de_shell> get_tracks * -within {{2 2} {25 25}}
{"USER_TRACK_5389"}
```

The following example returns all tracks:

```
de_shell> get_tracks "*TRACK_*"
{"DEF_TRACK_4683 USER_TRACK_5389"}
```

To get a report that shows the metal layer, the metal direction, the starting point, the number of tracks, the metal pitch, and the origin of the attributes, use the `report_track` command without any options.

Table 9-1 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 9-1: 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. You can also have keepout margins derived automatically by specifying the `-tracks_per_macro_pin` option, as shown in the following example. In this example, 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.

```
de_shell> 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.

To remove keepout margins of a specified type for the specified cells or library cells in the design, use the `remove_keepout_margin` command. To report keepout margins, use the `report_physical_constraints` command or the `report_keepout_margin` command.

The following example reports hard keepouts for cells in an object list named MY_CELL:

```
de_shell> report_keepout_margin -type hard MY_CELL
```

Placement Bounds Overview

A placement bound is a rectangular or rectilinear area within which to place cells and hierarchical cells. A placement bound groups together clock-gating cells or other timing-critical groups of cells. During placement, the tool ensures that the cells you grouped remain together and are not separated by other logic.

It is recommended not to impose bounds constraints but to allow the tool full flexibility to optimize placement for timing and routability. However, when QoR does not meet your requirements, you might improve QoR by using the `create_bounds` command.

You can specify two different types of bounds: move bounds and group bounds. 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.
- Exclusive bounds force the placement of the specified cell inside the bounds. All other cells must be placed outside the bounds.

Table 9-2 summarizes the commands related to placement bounds.

Table 9-2: 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`

See Also

- Creating Placement Bounds

Commands for Defining Physical Constraints

Table 9-3 lists the commands that you use to define physical constraints.

Table 9-3: Commands for Defining Physical Constraints

To define this physical constraint	Use this command
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.)	`set_aspect_ratio` and `set_utilization`
Exact core area (For details, see Defining the Core Placement Area.)	`create_site_row`
Relative port locations (For details, see Defining Port Locations)	`set_port_side`
Exact port locations (For details, see Defining Port Locations)	`set_port_location`
Macro location and orientation (For details, see Defining Macro Locations.)	`set_cell_location`
Placement keepout (blockages) (For details, see Defining Placement Blockages)	`create_placement_blockage`
Voltage area (For details, see Defining Voltage Area)	`create_voltage_area`
Placement bounds (For details, see Creating 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` and `create_pin_guide`
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`

To report any physical constraints that are applied to the design, use the `report_physical_constraints` command. To reset all physical constraints, use the `reset_physical_constraints` command before reading in a new or modified floorplan. To report a collection of polygons that exactly cover the region computed by performing a Boolean operation on the input polygons, use the `compute_polygons` command.

See Also

- Physical Constraints Overview

Saving Physical Constraints

You can use the `write_floorplan` command to save the floorplan information in IC Compiler or IC Compiler II format.

IC Compiler Format

By default, the `write_floorplan` command writes out physical constraint information in IC Compiler format. 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 DC Explorer, you should 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.

To use this output to create the floorplan information in IC Compiler, you must read it 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 DC Explorer. 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.

IC Compiler II Format

To save floorplan information in IC Compiler II format, use the `write_floorplan -format icc2` command. The output is a folder that contains files in Tcl and DEF format.

When you specify the `-format icc2` option, the tool writes out complete floorplan information. Any `write_floorplan` command options that allow you to write out partial floorplan information are ignored. To use this floorplan information in IC Compiler II, you must read it from the top-level design by sourcing the floorplan.tcl file in the output folder to load all the constraints to the IC Compiler II session.

The following table lists the files generated by the command and the constraints transferred through these files:

File name	Constraints transferred
floorplan.def	Die area, rows, tracks, components, pins, vias, special nets, keepout margins, and so on
floorplan.tcl	Route guides, placement blockages, bounds, layer routing directions
voltage_area.tcl	Voltage areas
rp.tcl	Relative placement groups
routing_rule.layer.tcl	Routing layer constraints on nets
routing_rule.tcl	Nondefault routing rule definitions
routing_rule.net.tcl	Routing rules on nets

Saving in ASCII Format for IC Compiler II

You can generate all the files needed to load a design into IC Compiler II by using the `write_icc2_files` command.

The command writes the design files and Tcl scripts for the current design into the directory specified by the `-output` option. The files can include the following data:

- Netlist in Verilog format
- Propagated switching activity information (backward SAIF file)
- Tcl floorplan and DEF files
- Design's power intent as a UPF command script
- Scan chain information in SCANDEF format
- Constraints in SDC format
- Timing contexts (scenarios)

In addition, the `write_icc2_files` command writes the following scripts:

- The DC Explorer settings in IC Compiler II format
- The top-level script to load all data generated in IC Compiler II

The `write_icc2_files` command writes the timing contexts (scenarios) in a format that can be read into IC Compiler II to create the same set of scenarios as specified in DC Explorer.

The scripts generated by the `write_icc2_files` command do not include scripts for library creation in IC Compiler II. You need to create a library that is consistent with the library used in DC Explorer and then source the top-level script.

When used with the `set_host_options` command, the `write_icc2_files` command uses up to the user-specified number of CPU cores on the same computer for parallel execution. For more details, see Using Multicore Technology.

The following examples show how to use the `write_icc2_files` command in DC Explorer to write the design files and Tcl scripts for the current design and then how to use the top-level script in IC Compiler II to load the design:

Example 9-32: ***Writing the Design Information From DC Explorer in IC Compiler II Format***

```
# Perform synthesis
de_shell> compile_exploration -scan -gate_clock

# Change names if needed
de_shell> change_names -rules verilog -hierarchy

# Save the file for IC Compiler II using the write_icc2_files command
de_shell> write_icc2_files -output ./icc2_files
de_shell> quit
```

Example 9-33: ***Reading the Design Into the IC Compiler II Tool***

```
# Perform library setting in IC Compiler II
icc2_shell> source -echo -verbose icc2_lib_setting.tcl

# Load the design using script created by the write_icc2_files command
icc2_shell> source -echo -verbose icc2_files/ORCA.icc2_script.tcl

# Continue with the flow
icc2_shell> commit_upf
icc2_shell> place_opt
```

If you run the command and warnings or errors occur, the tool issues an information message that points to the location of the log file that contains the warning or error message information:

```
de_shell> write_icc2_files -output ./icc2_files
Information: During the execution of write_icc2_files error messages were
issued, please check ./icc2_files/write_icc2_files.log files. (DCT-228)
```

Limitations

The `write_icc2_files` command currently has the following limitations:

- Hierarchical flows are not supported.
- Only the following constraints set on library cells are included:
  - SDC constraints: `set_timing_derate` and `set_driving_cell`
  - UPF constraints: `map_isolation_cell`, `map_retention_cell`, and
- The following commands are partially supported:
  - `set_congestion_options`

    The `-max_util` option is included only for global settings, but not for regional settings.
  - `set_multi_vth_constraint`

    The setting is passed only for threshold-voltage groups defined in the library; the settings in user-defined groups are not included.
- The following variables are partially supported:
  - `power_default_toggle_rate_type`

    The equivalent IC Compiler II variable only supports the `fastest_clock` value. The `absolute` setting is not passed.
  - `placer_channel_detect_mode`

    The equivalent IC Compiler II variable only supports the `true` and `false` values. The `auto` value is not passed.
  - `bus_naming_style`

    Only these delimiters defined in IC Compiler II are supported: [ ], { }, ( ), and < >.

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.

The following sections describe support for physical-only cells:

- Specifying Physical-Only Cells
- Extracting Physical-Only Cells From a DEF File
- Reporting Physical-Only Cells

Specifying Physical-Only Cells

To specify physical-only cells manually, perform the following steps:

1. Create the physical-only cell by specifying the `-only_physical` option with the `create_cell` command. For example,

```
de_shell> 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, DC Explorer 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 define 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:

```
de_shell> set obj [get_cells {"MY_U_PO_CELL"} -all]
de_shell> set_attribute -quiet $obj orientation N
de_shell> set_attribute -quiet $obj origin {3100.000 700.000}
```

```
de_shell> 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 DC Explorer by 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 DC Explorer.

  For example,

```
de_shell> set_cell_location -coordinates {3100.00 700.00} \
-orientation N -fixed MY_U_PO_CELL
```

See Also

- Extracting Physical-Only Cells From a DEF File
- Reporting Physical-Only Cells

Extracting Physical-Only Cells From a DEF File

To extract physical-only cells from a DEF file, 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 9-34, use the following command:

```
de_shell> extract_physical_constraints \
-allow_physical_cells my_design.def
```

Example 9-34: 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. DC Explorer marks these cells with the `is_physical_only` attribute when you run the `extract_physical_constraints` command with the `-allow_physical_cells` option:

- 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.

Note: The physical-only cell definitions in the DEF file must include cell locations. DC Explorer does not assign locations to physical-only cells during synthesis.

See Also

- Specifying Physical-Only Cells
- Reporting Physical-Only Cells

Reporting Physical-Only Cells

By default, the tool sets the `is_physical_only` attribute on all physical-only cells. To check if a cell is a physical-only cell, enter

```
de_shell> get_attribute [get_cells -all $cell_name] is_physical_only
```

You can use the following ways to report physical-only cells:

- To report physical-only cells for the current design, specify the `-only_physical` option with the `report_cell` command. For example,

  ```
  de_shell> report_cell -only_physical
  ```

- To report the floorplan information, which includes physical-only cell information, use the `report_physical_constraints` command.
- To return a collection of all physical-only cells in the design, use the `all_physical_only_cells` command.

Saving Physical-Only Cells

To save physical-only cell information, use the `write_floorplan` command. The command writes out floorplan information, including the physical-only cell information. To read the physical-only cell information, use the `read_floorplan` command.

The physical-only cell information and other constraints, such as timing, are saved in the .ddc file and can be read back into DC Explorer.

Note: Similar to all other floorplan 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 floorplan information to IC Compiler is to use floorplan exploration in DC Explorer.

Relative Placement

The relative placement capability in DC Explorer allows you to create structures in which you specify the relative column and row positions of instances. These structures are called relative placement structures, which are placement constraints.

During placement and legalization, these structures are preserved and the cells in each structure are placed as a single entity. Relative placement is also called physical datapath and structured placement. The following topics describe the relative placement flow in DC Explorer and how to perform relative placement:

- Relative Placement Overview
- Running the Relative Placement Flow
- Specifying Relative Placement Constraints
- Creating Relative Placement Using HDL Compiler Directives

Relative Placement Overview

Relative placement is often applied to datapaths and registers, but you can apply it to any cells to control the exact relative placement topology of gate-level logic groups and define 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.

Relative Placement Constraints

You specify relative placement constraints in either of the following ways:

- Specifying Relative Placement Constraints

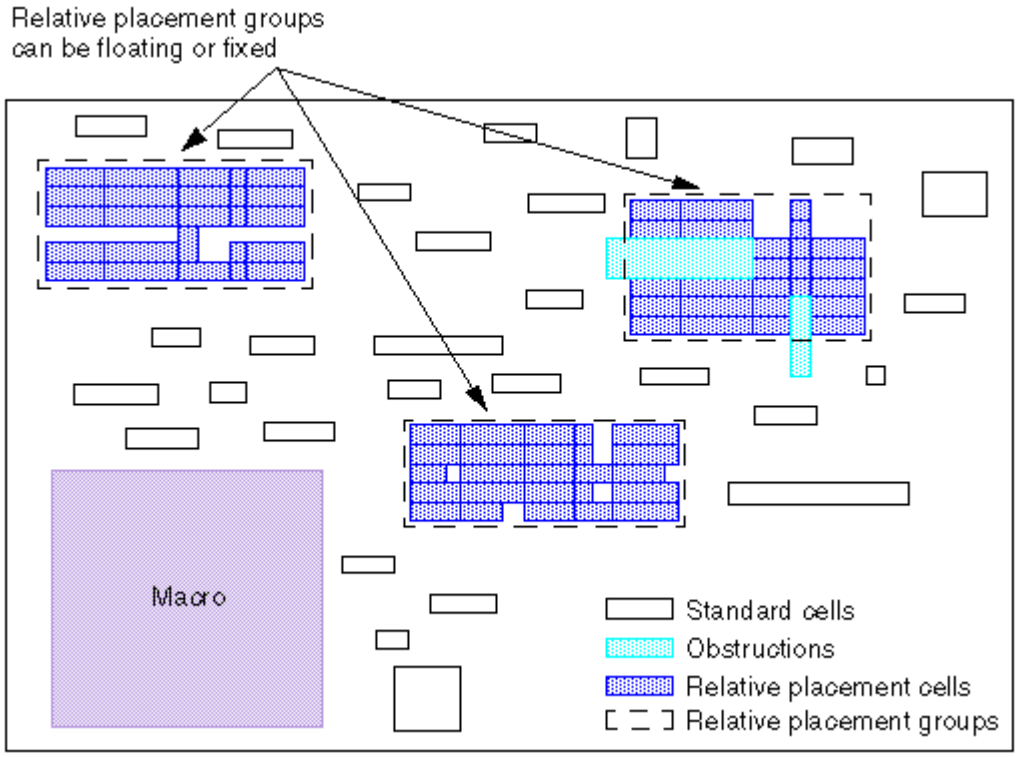
  Use a dedicated set of Tcl commands similar to the commands in the IC Compiler tool.

- Creating Relative Placement Using HDL Compiler Directives

  Use HDL compiler directives in an RTL design, a GTECH netlist, or a mapped gate-level netlist.

You create and annotate relative placement constraints to generate a matrix structure of instances and control their placement. You use the annotated netlist for physical optimization, during which the tool preserves the structure and places it as a single entity or group, as shown in Figure 9-7.

Figure 9-7: Relative Placement in a Floorplan



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 to yield greater predictability of QoR (wire length, timing, and power)
- Correlates better with the IC Compiler tool
- Can minimize congestion and improve routability

Considerations for Relative Placement

Keep the following points in mind when you perform relative placement:

- Relative placement constraints are preserved by the `uniquify` and `ungroup` commands.
- DC Explorer automatically places the `size_only` attribute on each relative placement cell

to preserve the relative placement structure.

- Ensure 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 produce poor results.
- Relative placement constraints are kept in the .ddc file and are not automatically transferred to the IC Compiler tool.

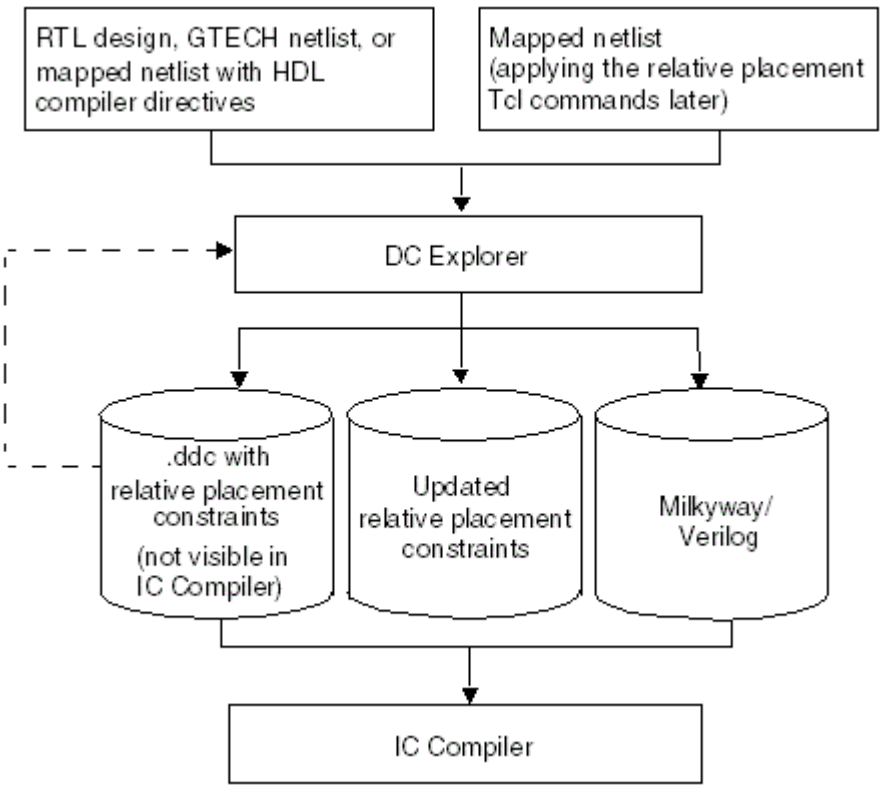
  You can use the `write_rp_groups` command to write out the relative placement constraints to a Tcl script that can be read into the IC Compiler tool.

See Also

- Specifying Relative Placement Constraints
- The IC Compiler Implementation User Guide

Running the Relative Placement Flow

The following figure shows an overview of the relative placement flow in DC Explorer.



To run the relative placement flow, follow these steps:

1. Read one or more of the following files in DC Explorer:
   - An RTL design, a GTECH netlist, or a mapped gate-level netlist with relative placement constraints specified by HDL compiler directives.
   - A mapped gate-level netlist without relative placement constraints; you specify the constraints with Tcl commands in step 2.
2. For a mapped gate-level netlist without relative placement constraints, define the relative placement constraints using Tcl commands.
   1. Create the relative placement groups by using the `create_rp_group` command.
   2. Add relative placement objects to the groups by using the `add_to_rp_group` command.

   The tool annotates the netlist with the relative placement constraints and places an implicit `size_only` constraint on these cells.
3. Read the floorplan information. For example,

   ```
   de_shell> extract_physical_constraints floorplan.def
   ```
4. Check the relative placement by using the `check_rp_groups` command.

   The command reports only the relative placement constraints defined in step 2. The command does not report relative placement data specified by HDL compiler directives until after the `compile_exploration` step.
5. Compile and optimize the design by using the `compile_exploration` command.
6. Visually verify the placement in the Design Vision layout view.

   Use the Design Vision layout window to examine the floorplan according to your expectations. The layout view displays floorplan constraints read in with the `extract_physical_constraints` command or other Tcl commands. You need to link all applicable designs and libraries to obtain an accurate floorplan.

   Note: You must set the `de_enable_physical_flow` variable to `true` to enable the physical flow capability.
7. Write out the relative placement constraints and data to a file by using the `write_rp_groups` command.

   To import the constraints and data into the IC Compiler tool, source the file or use the `import_designs` command with the `-rp_constraint` option.

See Also

- Specifying Relative Placement Constraints
- The *HDL Compiler for SystemVerilog User Guide*
- The HDL Compiler for VHDL User Guide
- The “Viewing the Floorplan” topic in Design Vision Help

Specifying Relative Placement Constraints

Relative placement 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.

To learn how to specify relative placement constraints, see

- Summary of Relative Placement Commands
- Creating Relative Placement Groups
- Anchoring Relative Placement Groups
- Adding Objects to a Group
- Aligning Leaf Cells Within a Column
- Querying Relative Placement Groups
- Checking Relative Placement Constraints
- Saving Relative Placement Information
- Removing Relative Placement Groups, Objects, and Attributes

Summary of Relative Placement Commands

You can specify relative placement constraints by using a dedicated set of commands similar to the commands in the IC Compiler tool. This table summarizes the commands available in DC Explorer for relative placement.

Table 9-4: Relative Placement 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.
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 a cell, relative placement group, or keepout from the specified relative placement groups.
rp_group_inclusions	Returns collections for directly 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 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 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, groups, and keepouts. A group is defined by the number of rows and columns it uses. To create a relative placement group, use the `create_rp_group` command.

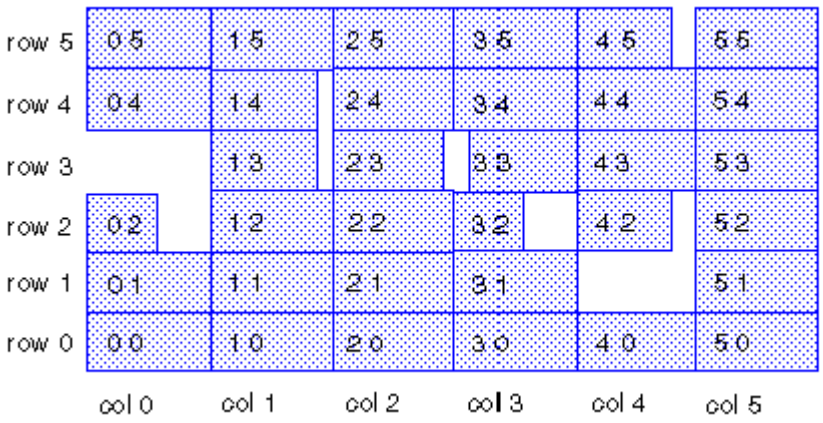
The `create_rp_group` command creates a relative placement group named *design_name*::*group_name*, where *design_name* is the design specified by the `-design` option. To refer to this group in other relative placement commands, you must use this name or a collection of relative placement groups. If you do not use the `-design` option, the default is the current design. If you do not specify any option, the command creates a relative placement group that has one column and one row without objects. To add objects, such as leaf cells, relative placement groups, and keepouts, to a relative placement group, use the `add_to_rp_group` command as described in Adding Objects to a Group.

For example, to create a group named designA::rp1 that has six columns and six rows, enter

```
de_shell> create_rp_group rp1 -design designA -columns 6 -rows 6
```

The following figure shows the column and row positions in a relative placement group.

Figure 9-8: 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 determined by 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.
- You do not need to use all positions in the structure. For example, positions 0 3 (column 0, row 3) and 4 1 (column 4, row 1) are not used.

Anchoring Relative Placement Groups

By default, the tool 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 `-x_offset` and `-y_offset` options with the `create_rp_group` or `set_rp_group_options` command. 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, the tool can determine the placement by sliding the group along the unspecified coordinate.

The following example specifies a relative placement group anchored at (100, 100):

```
de_shell> create_rp_group misc1 -design block1 \
   -columns 3 -rows 10 -x_offset 100 -y_offset 100
```

Adding Objects to a Group

You can add leaf cells, other relative placement groups, and keepouts to a relative placement group that was 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, ensure that

- The relative placement group to which you are adding the object exists
- The object must be added to an empty location in the relative placement group

To learn how to add objects to a group, see

- Adding Leaf Cells
- Adding Relative Placement Groups
- Adding Keepouts

Adding Leaf Cells

To add leaf cells to a relative placement group, use the `add_to_rp_group` command. In a relative placement group, a leaf cell occupying multiple column positions or multiple row positions is called leaf cell straddling. You can create a more compact relative placement group by specifying leaf cell straddling. To do this, specify multiple column or row positions by using the `-num_columns` or `-num_rows` options respectively with the `add_to_rp_group` command. The default is one column and one row.

For example, to create a leaf cell of two columns and one row, enter

```
de_shell> add_to_rp_group rp_group_name -leaf cell_name \
   -column 0 -num_columns 2 -row 0 -num_rows 1
```

The following restrictions apply when you specify cell straddling:

- You should not place a relative placement keepout at the same location of a straddling leaf cell. Straddling is for leaf cells only, but not for hierarchical groups or keepouts.
- You should not apply compression to a straddling leaf cell that has 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.

To specify the placement orientation of a leaf cell, use the `-orientation` option. To specify the alignment method of the cells in a column, use the `-alignment` or `-pin_align_name` option.

See Also

- Aligning Leaf Cells Within a Column

Adding Relative Placement Groups

Hierarchical relative placement allows relative placement groups to be embedded in other relative placement groups. You can use hierarchical relative placement to simplify relative placement constraints.

Using hierarchical relative placement, you do not need to provide relative placement information multiple times for a recurring pattern. You handle the embedded groups just like leaf cells. You add a relative placement group to a hierarchical group by using either of the following methods with the `add_to_rp_group` command:

- Including the relative placement group

  If the group 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 once in a group of the same design. However, a group that contains an included group can be further included in another group of the same design or can be instantiated in a group of a different design.

- Instantiating the relative placement group

  If the group to be added is in a subdesign instance of its parent group, it is an instantiated group. You can instantiate groups only in hierarchical designs.

To specify a group using the `-hierarchy` option, you must use the group in the reference design instance specified by the `-instance` option with the `add_to_rp_group` command. 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 can replicate relative placement information across multiple instances of a design and create relative placement relationships between those instances.

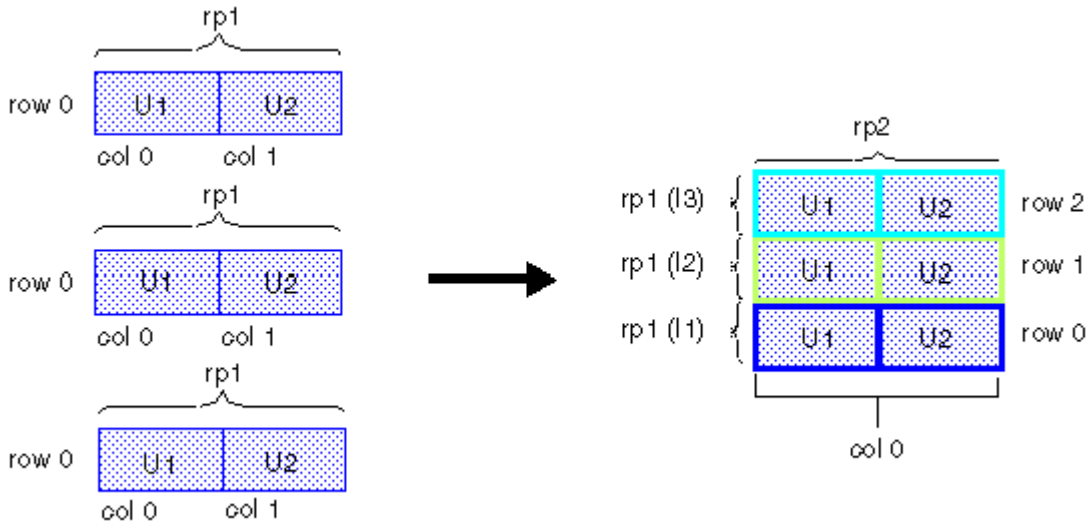
Example 9-35 creates a hierarchical group named rp2 that contains three instances of group rp1. Group rp1 is in the pair_design design and contains leaf cells U1 and U2. Group rp2 is a hierarchical group in the mid_design design that instantiates group rp1 three times (mid_design must contain at least three instances of pair_design). Group rp2 is treated as a leaf cell. You can instantiate group rp2 as many times as the mid_design design is instantiated in the netlist. The resulting hierarchical relative placement group is shown in Figure 9-9.

Example 9-35: 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 9-9: Instantiating Groups in a Hierarchical Group



Adding Keepouts

To add a keepout to a relative placement group, use the `add_to_rp_group` command. The following example creates a keepout named gap1 containing the TOP:misc group list:

```
de_shell> add_to_rp_group TOP::misc -keepout gap1 \
   -column 0 -row 2 -width 15 -height 1
```

Aligning Leaf Cells Within a Column

Controlling cell alignment can improve the timing and routability of the design. You can align leaf cells in a column of a relative placement group by using the following alignment methods:

- Aligning to the Bottom-Left Corner
- Aligning to the Bottom-Right Corner
- Aligning to a Pin Location

Aligning to the Bottom-Left Corner

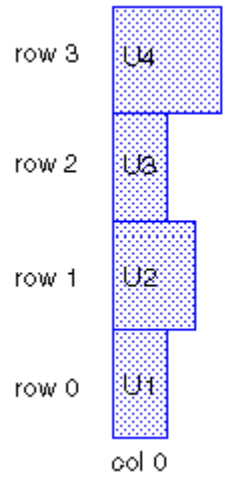
To align leaf cells to the bottom-left corner, use the `-alignment bottom-left` option with the `create_rp_group` or `set_rp_group_options` command. This is the default if you do not specify any alignment method. For example,

```
de_shell> set_rp_group_options -alignment bottom-left [get_rp_groups *]
```

The following script defines a relative placement group that is bottom-left aligned. The resulting structure is shown in Figure 9-10.

```
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 9-10: Bottom-Left Aligned Relative Placement Group



Aligning to the Bottom-Right Corner

To align leaf cells to the bottom-right corner, use the `-alignment bottom-right` option with the `create_rp_group` or `set_rp_group_options` command. For example,

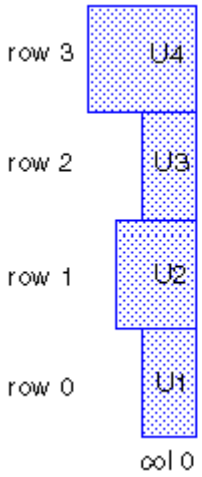
```
de_shell> set_rp_group_options -alignment bottom-right \
 [get_rp_groups *]
```

Note: For hierarchical relative placement groups, the bottom-right alignment does not propagate through the hierarchy.

The following script defines a relative placement group that is bottom-right aligned. The resulting structure is shown in Figure 9-11.

```
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 9-11: Bottom-Right Aligned Relative Placement Group



Aligning to a Pin Location

To align leaf cells to a pin location, use the `-alignment bottom-pin` and `-pin_align_name` options with the `create_rp_group` or `set_rp_group_options` command. For example,

```
de_shell> set_rp_group_options -alignment bottom-pin \
 -pin_align_name align_pin
```

The tool looks for the specified alignment pin in each cell within the column. If the pin is

- Found in the cell, the tool aligns the cell to the pin location
- Not found in the cell, the tool aligns the cell to the bottom-left corner and issues an information message
- Not found in any cell, the tool issues a warning message

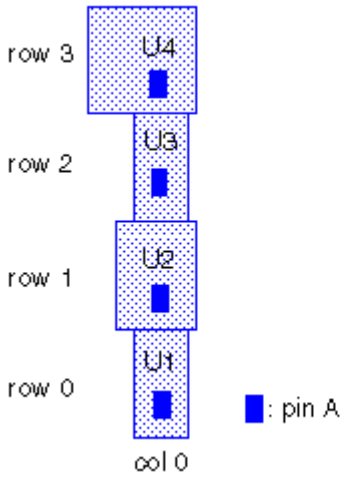
If you specify both pin alignment and cell orientation, the tool resolves potential conflicts as follows:

- User specifications for cell orientation take precedence over the pin alignment done by DC Explorer.
- Pin alignment done by DC Explorer takes precedence over the cell orientation optimization done by DC Explorer.

The following script defines a relative placement group that is aligned by pin A. The resulting structure is shown in Figure 9-12.

```
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 9-12: Relative Placement Group Aligned by Pin A



Overriding Pin Alignment

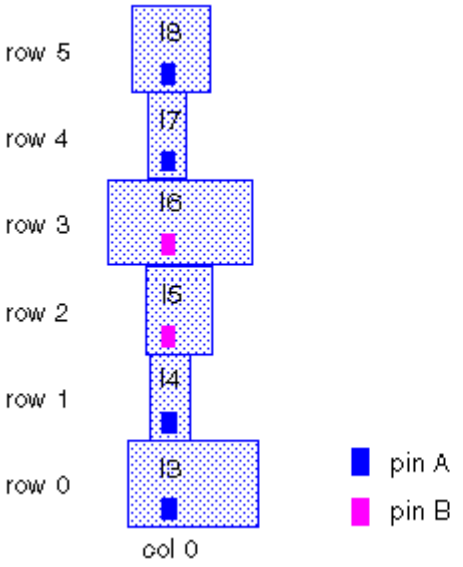
When you specify an alignment pin for a group, the pin alignment method applies to all cells in the column. To override the pin alignment for specific cells in the group, specify another pin by using the `-pin_align_name` option with the `add_to_rp_group` command.

Note: You cannot specify a cell-specific alignment pin when you add a leaf cell from the relative placement hierarchy browser.

The following script defines relative placement group misc1, which uses pin A as the group alignment pin, but instances I5 and I6 use pin B as the alignment pin. The resulting structure is shown in Figure 9-13.

```
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 9-13: ***Relative Placement Group Aligned by Pins***



Querying Relative Placement Groups

Table 9-5 table lists the commands for querying relative placement groups to annotate, edit, and view.

Table 9-5: Commands for Querying Relative Placement Groups

To return a collection of	Use this command
The relative placement groups that match certain criteria	`get_rp_groups`
All or specified relative placement groups and their included and instantiated groups 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

```
de_shell> get_rp_groups r*::g*
{ripple::grp_ripple}
```

To set the utilization to 95 percent for all relative placement groups, enter

```
de_shell> set_rp_group_options [all_rp_groups] -utilization 0.95
```

Checking Relative Placement Constraints

To check relative placement constraints, run the `check_rp_groups` command. The command reports relative placement failures.

The `check_rp_groups` command checks for the following failures:

- The relative placement group cannot be placed as a single entity.
- 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.

The `check_rp_groups` command does not check for failures when keepouts in a relative placement group are not created correctly.

The generated report contains separate sections for critical and noncritical failures. If a failure prevents the group from being placed as a single entity as defined by the relative placement constraints, the failure is critical. If a failure does not prevent placement but causes the relative placement constraint violations, the failure is noncritical.

You can check all relative placement groups by specifying the `-all` option. To check specific groups, specify the group names. By default, the command displays the report to the screen. To save the report to a file, specify a file name by using the `-output` option.

For example, the following command checks the relative placement constraints for groups compare17::seg7 and compare17::rp_group3 and saves the output in a file called rp_failures.log:

```
de_shell> check_rp_groups "compare17::seg7 compare17::rp_group3" \
 -output rp_failures.log
```

To show a detailed report, use the `-verbose` option as shown in the following example:

```
de_shell> check_rp_groups -all -verbose
```

Saving Relative Placement Information

To write out relative placement constraints, use the `write_rp_groups` command.

The following example saves all the relative placement groups, removes the information from the design, and re-creates the information about the design:

```
de_shell> get_rp_groups
{mul::grp_mul ripple::grp_ripple example3::top_group}
de_shell> write_rp_groups -all -output my_groups.tcl
1
de_shell> remove_rp_groups -all -quiet
1
de_shell> get_rp_groups
Error: Can't find objects matching '*'. (UID-109)
de_shell> source my_groups.tcl
{example3::top_group}
de_shell> 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 for adding leaf cells, hierarchical groups, and keepouts to these groups, but not commands for generating subgroups within hierarchical groups. The command writes out updated relative placement constraints and includes name changes from uniquification or ungrouping.

If you specify multiple column positions or multiple row positions for a cell using the `-num_columns` or `-num_rows` option with the `add_to_rp_group` command, the `write_rp_groups` command writes out the multiple-location cell.

Removing Relative Placement Groups, Objects, and Attributes

You can remove relative placement groups, objects from a relative placement group, or relative placement group attributes.

Removing Relative Placement Groups

To remove all or specific relative placement groups, specify the `-all` option or the relative placement groups with the `remove_rp_groups` command. When you specify a list of relative placement groups to be removed, the tool removes only the specified groups but not the groups included or instantiated within the specified group. To remove the included and instantiated groups of the specified groups, you must specify the `-hierarchy` option.

The following example removes the grp_ripple relative placement group and confirms the removal:

```
de_shell> get_rp_groups
{mul::grp_mul ripple::grp_ripple example3::top_group}
de_shell> remove_rp_groups ripple::grp_ripple
Removing rp group 'ripple::grp_ripple'
1
de_shell> get_rp_groups *grp_ripple
Error: Can't find object 'grp_ripple'. (UID-109)
de_shell> remove_rp_groups -all
Removing rp group 'mul::grp_mul'
Removing rp group 'example3::top_group'
1
```

Removing Objects From a Relative Placement Group

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`).

The following command removes leaf cell carry_in_1 from group grp_ripple:

```
de_shell> remove_from_rp_group ripple::grp_ripple -leaf carry_in_1
```

If you specify multiple column positions or multiple row positions for a cell using the `-num_columns` or `-num_rows` option with the `add_to_rp_group` command, the `remove_from_rp_group` command removes the cell from all its locations.

Removing Relative Placement Group Attributes

To remove relative placement group attributes, use the `remove_rp_group_options` command. You must specify the group name and at least one option; otherwise, this command has no effect.

The following example removes the `x_offset` attribute from the block1::misc1 group:

```
de_shell> remove_rp_group_options block1::misc1 -x_offset
{block1::misc1}
```

The command returns a collection of relative placement groups for which attributes were changed. If no attributes were changed, the command returns an empty string.

Creating Relative Placement Using HDL Compiler Directives

DC Explorer 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 documentation.

See Also

- The *HDL Compiler for SystemVerilog User Guide*
- The *HDL Compiler for VHDL User Guide*

10 Optimization

DC Explorer provides fast synthesis to speed up the development of high-quality RTL. With much faster runtime than full synthesis, the tool allows you to perform what-if analyses of various configurations to meet the design's functional, timing, and area requirements. You use the `compile_exploration` command to synthesize a design. DC Explorer optimizes all endpoints concurrently, using different optimization engines to attain linear runtime performance relative to the design size.

To learn the many factors affecting the optimization results, see

- Compile Strategies
- Performing a Top-Down Compile
- Performing a Bottom-Up Compile
- Performing Optimization for High-Performance Designs
- Performing Manual High-Fanout Synthesis
- Resolving Multiple Instances of a Design Reference
- Defining Library Subset Restrictions
- Optimizing Multicorner-Multimode Designs
- Performing Optimization With Floorplan Physical Constraints
- Performing Power Optimization
- Congestion Reporting in the Physical Flow
- Pipelined-Logic Retiming
- Additional Optimization Features

See Also

- The *Design Compiler User Guide*

Compile Strategies

You can use various strategies to compile your hierarchical design. The basic strategies are

- Top-down compile

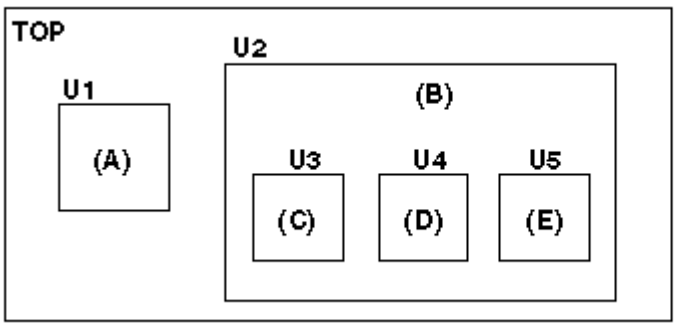
  The top-level design and all its subdesigns are compiled together.

- Bottom-up compile

  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.

The top-down compile strategy is demonstrated, using the simple design shown in Figure 10-1.

Figure 10-1: Design to Illustrate Compile Strategies



The top-level or global specifications for this design, given in Table 10-1, are defined by the script shown in Example 10-1. These specifications apply to TOP and all its subdesigns.

Table 10-1: Design Specifications for Design TOP

Specification type	Value
Operating condition	WCCOM
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 10-1: Constraints File for Design TOP (defaults.con)

```
set_operating_conditions WCCOM
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).

See Also

- Performing a Top-Down Compile
- Performing a Bottom-Up Compile

Performing a Top-Down Compile

You can use the top-down compile strategy for designs that are not memory or CPU limited. Top-level designs that are memory limited can often be compiled using this strategy if you first replace some of the subdesigns with block abstractions. Replacing a subdesign with a block abstraction can reduce the memory requirements for the subdesign instantiation in the top-level design.

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:

1. Load the entire design.
2. Apply attributes and constraints to the top level.

   Attributes and constraints implement the design specification.

   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.

Note: If your top-level design contains one or more block abstractions, use the compile flow described in Using Hierarchical Models.

A top-down compile script for the TOP design is shown in Example 10-2.The script contains comments that identify each of the steps. The constraints are applied by including the constraint file (defaults.con) shown in the example of Compile Strategies.

Example 10-2: ***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
report_design_mismatch
/* apply constraints and attributes */
source defaults.con
/* compile the design */
compile_exploration
```

See Also

- Compile Strategies
- Using Hierarchical Models
- Working With Designs in Memory
- Defining the Design Environment
- Defining Design Constraints

Performing a Bottom-Up Compile

The recommended strategy is a top-down compile flow. However, to address design and runtime challenges or use a divide-and-conquer synthesis approach, you can use a bottom-up compile, also known as a hierarchical compile flow. In the bottom-up compile flow, you compile the subblocks separately and then incorporate them in the top-level design. DC Explorer can read the following types of hierarchical blocks:

- Netlist generated in DC Explorer
- Block abstractions generated in the DC Explorer or IC Compiler tool

You can compile the subblock and provide it to the top-level design as a full .ddc netlist or a block abstraction. Alternatively, you can continue working on the subblock in the IC Compiler tool to create a block abstraction, which you can then provide to the top-level design. Timing and physical information of the subblock are propagated to the top-level for physical synthesis. In addition, you can provide 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_exploration -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. During top-level optimizations, the tool considers placement and can use the same physical constraints as your back-end tool.

To learn more about the bottom-up compile flow, see

- Overview of Bottom-Up Compile
- Compiling the Subblock
- Compiling the Design at the Top Level

See Also

- Compile Strategies
- Overview of Hierarchical Models

Overview of Bottom-Up Compile

In the bottom-up flow, compile the subblock and save the mapped subblock as a netlist in .ddc format and then work on the subblock (.ddc netlist) in the IC Compiler tool to generate a block abstraction.

Alternatively, you can compile the subblock and save the mapped subblock as a block abstraction in .ddc format.

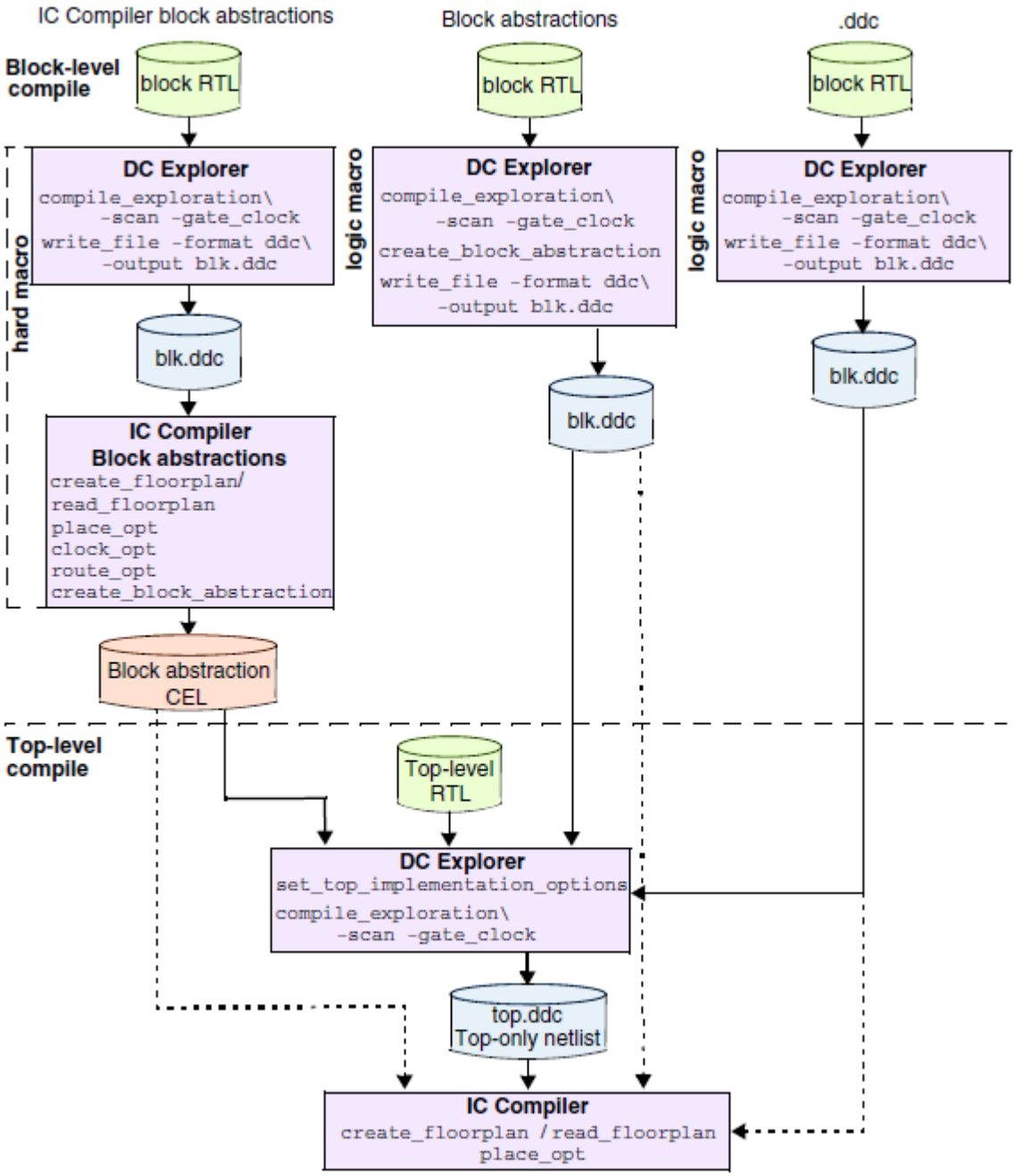
Top-level synthesis can accept the following types of mapped subblocks:

- .ddc netlist synthesized in DC Explorer
- Block abstraction created in DC Explorer
- Block abstraction created in the IC Compiler tool

Note: The IC Compiler tool cannot accept DC Explorer block abstractions.

Figure 10-2 provides an overview of the hierarchical flow for designs containing block abstractions.

Figure 10-2: Overview of the Hierarchical Flow for Designs Containing Block Abstractions



The bottom-up flow requires the following main tasks:

1. Compiling the Subblock

   Compile the subdesigns individually.

2. Compiling the Design at the Top Level

   Read in the top-level design and any compiled subdesigns not already in memory; compile the top-level design.

Compiling the Subblock

Follow the steps described in this procedure to compile subblocks.

To compile the subblock,

1. Specify the logic and physical 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.
5. (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.
6. Perform a test-ready and clock-gating compile of the subblock by using the `compile_exploration -scan -gate_clock` command.
7. Run the `uniquify -force` command to create a unique design name for each cell instance to avoid name collisions when integrating the design at the top level.
8. Apply the Verilog naming rules to all the design objects before writing out the design data files by using the `change_names -rules verilog -hierarchy` command.

The steps you continue from this point forward depend on which of the following tasks you want to perform:

- Creating a full .ddc netlist for the subblock

  Save the mapped subblock in the .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, as shown in Compiling the Design at the Top Level.

- Generating a Block Abstraction for the Subblock

  To generate a block abstraction for the subblock, continue with the steps in this task.

- Generating a Block Abstraction for the Subblock in IC Compiler

  To implement the subblock in the IC Compiler tool, continue with the steps in this task.

See Also

- Specifying Logic Libraries
- Specifying Physical Libraries
- Using Floorplan Physical Constraints
- Compiling the Design at the Top Level
- The “Viewing the Floorplan” topic in Design Vision Help

Generating a Block Abstraction for the Subblock

To generate a block abstraction for the subblock, 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,

   ```
   de_shell> write_file -hierarchy -format ddc -output
   myfile.mapped.ddc
   ```

   You should use the `write_file` command to save the block abstraction after you create it. The `write_file` command writes the complete design, including the block abstraction information, into a .ddc file.

To integrate the block abstraction, continue with the steps in Compiling the Design at the Top Level.

Generating a Block Abstraction for the Subblock in IC Compiler

You can continue to implement the subblock in the IC Compiler tool. To create a block abstraction 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 DC Explorer by using the `write_file -format ddc` command.
2. In the IC Compiler environment, 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. Perform placement by running the `place_opt` command.
6. Perform clock tree synthesis by running the `clock_opt` command.
7. Route the design by running the `route_opt` command.
8. Create the block abstraction by running the `create_block_abstraction` command.
9. Create the CEL view by running the `save_mw_cel` command.

For more information about IC Compiler commands, see the IC Compiler documentation.

Compiling the Design at the Top Level

To compile the design at the top level,

1. Specify the logic and physical libraries.

   If you are using a block abstraction created by the IC Compiler tool, you need to add the Milkyway design library that contains the block abstraction CEL view to the Milkyway reference library list at the top level.

2. If you are using block abstractions created either in the DC Explorer or IC Compiler tool, 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 DC Explorer (.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.

   - Block abstraction created in the IC Compiler tool (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.

Note: To ignore the timing paths that are entirely in block abstractions during top-level synthesis, see the Ignoring Timing Paths Within Block Abstractions topic.

3. (Optional) Ignore register-to-register paths that are entirely within block abstractions by setting the `timing_ignore_paths_within_block_abstraction` variable to `true`.
4. Read in the top-level design (Verilog, VHDL, netlist, or .ddc) file.
5. Read in the mapped subblock.

   The subblock can be any of the following:

   - Complete netlist in .ddc format created in DC Explorer.

     If the `set_top_implementation_options` command options are not set for the block, the full netlist .ddc will be read in for the block.

   - Block abstraction in .ddc format created in DC Explorer.

     You must use the `set_top_implementation_options` command before the .ddc file is loaded 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.

   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.

6. Set the current design to the top-level design.
7. (Optional) To specify that the subblock should be treated as a physical subblock, use the `set_physical_hierarchy` command for complete netlists in .ddc format.

   When you do this, top-level synthesis preserves both the logical structure and cell placement inside the block. If you do not want to treat a netlist as a physical block, omit the `set_physical_hierarchy` command.

8. Use the `link` command to link the subblocks specified in the previous step to the top level.
9. Apply the top-level timing and power constraints.
10. (Optional) Provide 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.

11. (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.

12. Run the `compile_exploration -scan -gate_clock` command.

13. Write out the top-level netlist by using the `write_file -format ddc` command.

    The `write_file` command merges the changes with the original .ddc design and writes the complete block as a new .ddc file.

See Also

- Specifying Logic Libraries
- Specifying Physical Libraries
- Ignoring Timing Paths Within Block Abstractions
- Using Floorplan Physical Constraints
- The “Viewing the Floorplan” topic in Design Vision Help

Performing Optimization for High-Performance Designs

You can use the `compile_exploration` command to perform optimization for better quality of results (QoR), especially for high-performance designs that have significantly tight timing constraints. This command allows you to apply the best possible set of timing-centric variables during compile for critical delay optimization and to improve area QoR. The default compile generates good results for most designs. If your design does not meet the optimization goals after design exploration, try the following techniques:

- Optimizing for Minimum Area
- Datapath Optimization

Datapath Optimization

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.

Datapath optimization is enabled by default when you run the `compile_exploration` command. DC Explorer uses the datapath generator to build arithmetic components for optimal QoR considering the bit-level timing context during optimization. To use this datapath generator with the `compile_exploration` command, include the dw_foundation.sldb library in the synthetic library list. If necessary, use the `set synthetic_library dw_foundation.sldb` command. A DesignWare license is required.

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.

DC Explorer enables datapath extraction and explores various datapath and resource-sharing options during compile. The 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

Optimizing for Minimum Area

If your design has timing constraints, these constraints always take precedence over area requirements. For area-critical designs, specify realistic constraints. To fine-tune the area, you can leave the hierarchy intact and let DC Explorer perform boundary optimization by default. For greater area reduction, you might have to remove hierarchical boundaries.

If your design does not meet the area constraints, you can try the following methods to optimize across hierarchical boundaries:

- Boundary Optimization
- Hierarchy Removal

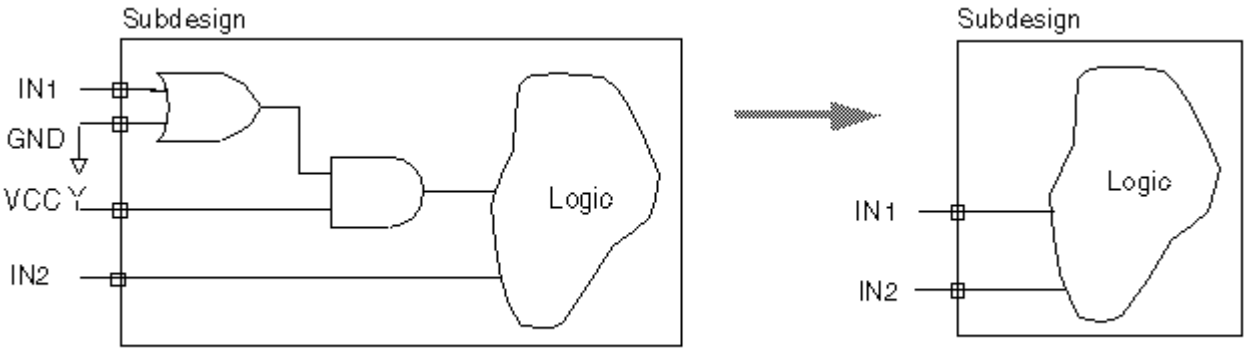
Boundary Optimization

By default, DC Explorer performs optimization across hierarchical boundaries (boundary optimization). You can also use the following command to run boundary optimization:

```
de_shell> set_boundary_optimization subdesign true
```

When you enable boundary optimization, DC Explorer propagates constants, unconnected pins, and complement information. In designs that have many constants (VCC and GND) connected to the inputs of subdesigns, propagation can reduce area. Figure 10-3 shows this relationship.

Figure 10-3: Benefits of Boundary Optimization



To enable area reduction with boundary optimization disabled, set the `compile_optimize_unloaded_seq_logic_with_no_bound_opt` variable to `true` to remove unconnected logic. The default is `false`. When this variable is set to `true`, the tool

- Removes unconnected registers across design hierarchies
- Removes unused bits and infers smaller-sized multibit registers by propagating unconnected logic through multibit registers

Hierarchy Removal

By default, DC Explorer automatically ungroups small design hierarchies in the first pass of the compile, and it ungroups hierarchies along critical paths using delay-based auto-ungrouping strategy in the second pass of the compile. Ungrouping removes hierarchy boundaries and allows DC Explorer to optimize over a larger number of gates. You can also ungroup specific hierarchies before optimization by using the `set_ungroup` command to designate which cells to be ungrouped.

By default, all DesignWare hierarchies are unconditionally ungrouped in the second pass of the compile. You can prevent this ungrouping by setting the `compile_ultra_ungroup_dw` variable to `false`, changing it from its default of `true`.

See Also

- Removing Levels of Hierarchy

Performing Manual High-Fanout Synthesis

To control the creation and removal of buffer trees, use the `create_buffer_tree` command to manually run high-fanout synthesis. By default, the tool performs automatic high-fanout synthesis when you run the `compile_exploration` command.

Note: You must have a mapped design to run manual high-fanout synthesis.

The `create_buffer_tree` command removes the preexisting buffer trees in the design and re-creates the buffer trees. To preserve preexisting buffer trees, specify the `-incremental` option. In this case, the command constructs a buffer tree, if needed, on each net specified by the `-from` option to reduce the high fanout of each net, but it does not remove any existing buffers or inverters.

To run manual high-fanout synthesis,

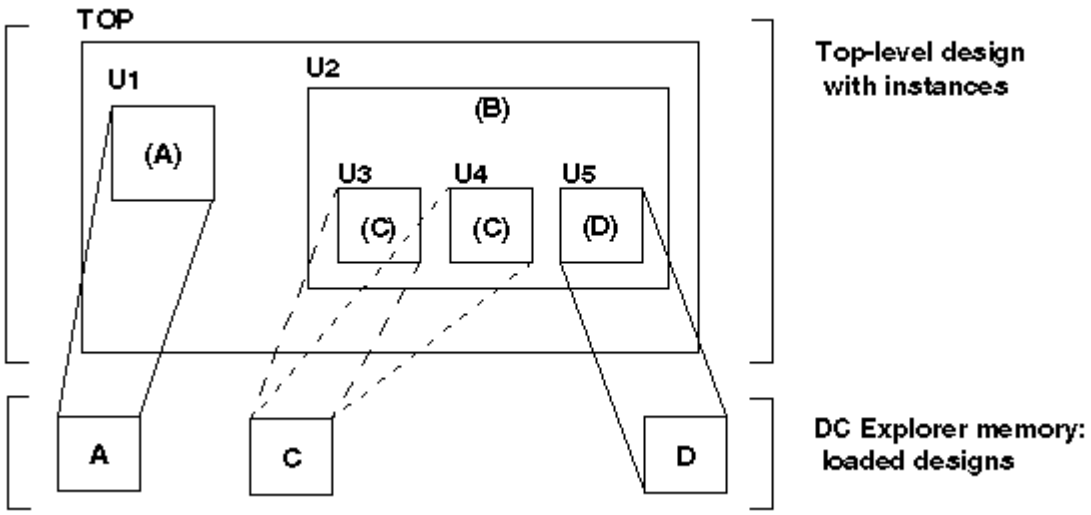
1. Report the current buffer tree implementation on a mapped netlist by running the `report_buffer_tree` command .
2. Identify any suboptimal buffering that was performed by the `compile_exploration` command.
3. Remove the existing buffer trees and create additional buffer trees as needed using the `remove_buffer_tree` and `create_buffer_tree` commands, respectively.

The inserted buffers and inverters might be optimized further by other optimization commands. To preserve specific buffers and inverters during optimization, set optimization restriction attributes such as `size_only` or `dont_touch` on them.

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 10-4 shows the design TOP, in which design C is referenced twice (U2/U3 and U2/U4).

Figure 10-4: Multiple Instances of a Design Reference



Use the `-multiple_designs` option with `check_design` command to report information messages related to multiply-instantiated designs. 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:

- Uniquify Method

  DC Explorer automatically uniquifies designs as part of the compile process. However, 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 uniquifies the designs again when compiling the design. You cannot turn off the uniquify process.

- 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.

- Ungroup Method

  This method uses the `ungroup` command to remove the hierarchy.

See Also

- Preserving Subdesigns

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.

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 that have 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. DC Explorer makes unique copies for cells specified with the `-reference` or the `-cells` option even if they are marked with the `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:

```
de_shell> uniquify -cell mid1/bot1
```

DC Explorer 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`

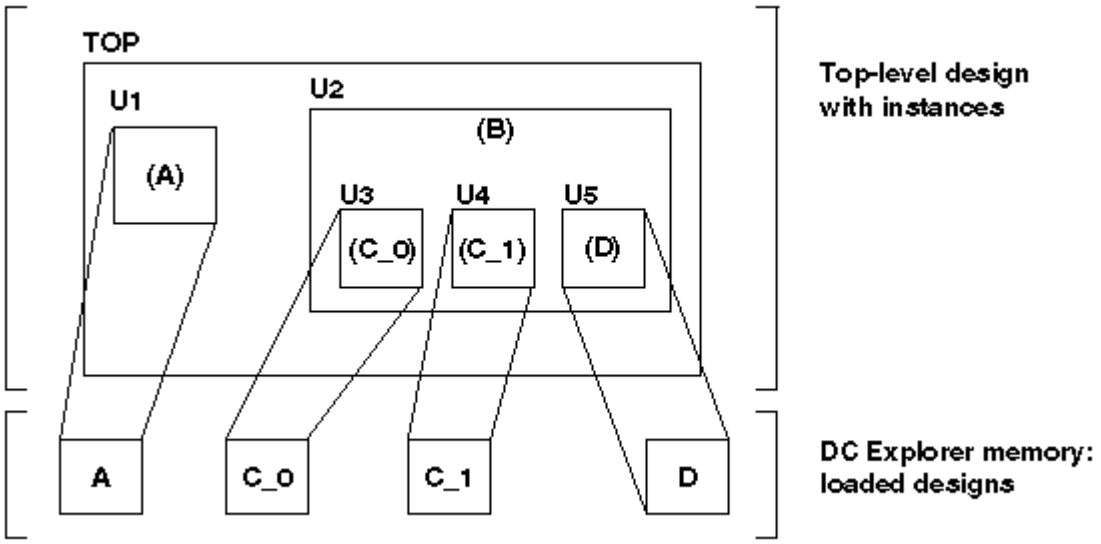
The smallest integer value that forms a unique subdesign name.

The following command sequence resolves the multiple instances of design C in design TOP shown in the figure of Resolving Multiple Instances of a Design Reference; 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.

```
de_shell> current_design top
de_shell> compile_exploration
```

Figure 10-5 shows the result of running this command sequence.

Figure 10-5: 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.

To use the compile-once-don't-touch method to resolve multiple instances, follow these steps:

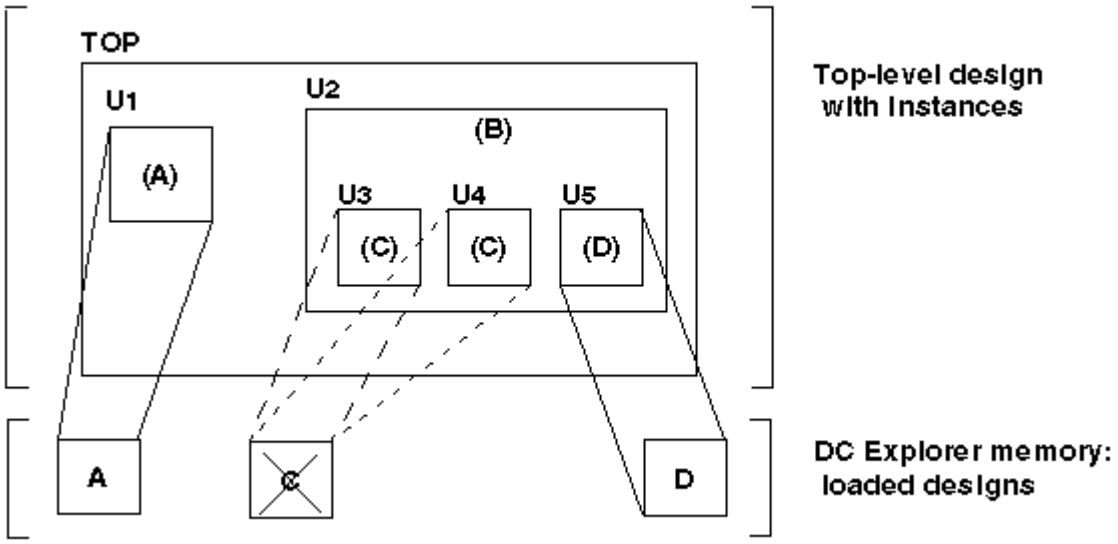
1. Compile the referenced subdesign.
2. Use the `set_dont_touch` command to set the `dont_touch` attribute on all instances that reference the compiled subdesign.
3. Compile the entire design.

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.

```
de_shell> current_design top
...
de_shell> current_design C
de_shell> compile_exploration
de_shell> current_design top
de_shell> set_dont_touch {U2/U3 U2/U4}
de_shell> compile_exploration
```

Figure 10-6 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 10-6: Compile-Once-Don't-Touch Results



The compile-once-don't-touch method has the following advantages:

- Compiles the reference design once
- 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.

See Also

- Preserving Subdesigns

Ungroup Method

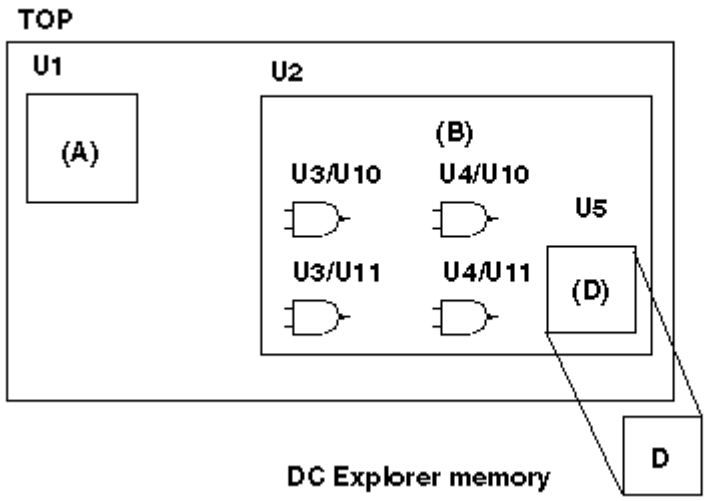
The ungroup method has the same effect as the uniquify method (it makes unique copies of the design), but in addition, it removes levels of hierarchy. This method uses the `ungroup` command to produce a flattened netlist.

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:

```
de_shell> current_design B
de_shell> ungroup {U3 U4}
de_shell> current_design top
de_shell> compile_exploration
```

Figure 10-7 shows the result of running this command sequence.

Figure 10-7: 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 disadvantage of using the ungroup method is the removal of user-defined design hierarchy.

See Also

- Removing Levels of Hierarchy

Preserving Subdesigns

To preserve a subdesign during optimization, use the `set_dont_touch` command. The command 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.

Note: Any block abstractions in your design are automatically marked with the `dont_touch` attribute. Also, the cells of block abstractions are marked as don't touch.

Use the `set_dont_touch` command to specify subdesigns not to be optimized with the rest of the design hierarchy. The `dont_touch` attribute does not prevent or disable timing through the design.

When you use the `set_dont_touch` command, remember the following points:

- Setting the `dont_touch` attribute on a hierarchical cell sets an implicit don't touch value on all cells below that cell.
- Setting the `dont_touch` attribute on a library cell sets an implicit don't touch value on all instances of that cell.
- Setting the `dont_touch` attribute on a net sets an implicit don't touch value only on mapped combinational cells connected to that net. If the net is connected only to generic logic, optimization might remove the net.
- Setting the `dont_touch` attribute on a reference sets an implicit don't touch value 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 the `dont_touch` attribute on the top-level design has no effect because the top-level design is not instantiated within any other design.
- You cannot ungroup objects marked with the `dont_touch` attribute by using the `ungroup`

command. The automatic ungrouping of DC Explorer has no effect on don't touch objects.

Note: The `dont_touch` attribute is ignored on synthetic part cells (for example, many of the cells read in from an HDL description) and on nets that have unmapped cells on them. During compilation, warnings appear for don't touch nets connected to unmapped cells (generic logic).

To remove the `dont_touch` attribute, use the `remove_attribute` command or the `set_dont_touch` command set to `false`.

See Also

- Using Hierarchical Models

Defining Library Subset Restrictions

You can restrict the mapping and optimization of sequential cells and instantiated combinational cells to a user-specified subset of library cells in a target library. To define one or more subsets of library cells, use the `-family_name` option with the `define_libcell_subset` command. The command groups the library cells into a subset only if they meet the following criteria:

- They must be sequential cells or instantiated combinational cells.
- They cannot belong to another subset.
- They must have the same functional identification.

After the library cells are grouped into a subset, they are not available for general mapping during compilation. DC Explorer uses these library cells exclusively to map the sequential cells and instantiated combinational cells specified by the `set_libcell_subset` command during optimization. To specify a list of cells to be optimized, use the `-object_list` option with the `set_libcell_subset` command. The specified cells must be either unmapped or instantiated as one of the library cells in the subset.

When you run the `define_libcell_subset` and `set_libcell_subset` commands, DC Explorer restricts all optimizations, including cell sizing and cell swapping, to the cells specified in the library subset. This subset restriction applies only to cells that are created, mapped, or modified during optimization, but not to any portions of the design that are not optimized.

In the following example, the `define_libcell_subset` command groups the SDFLOP1 and SDFLOP2 logic library cells into a subset called specialflops. Next, the `set_libcell_subset` command sets the specialflops library subset for the reg01 and reg02 instances.

```
de_shell> define_libcell_subset -libcell_list {SDFLOP1 SDFLOP2} \
-family_name specialflops
de_shell> set_libcell_subset -libcell_list [get_cells {reg01 reg02}] \
-family_name specialflops
```

To report information about user-defined library subsets, use the `report_libcell_subset` command. The following example reports the library subset for the reg01 sequential cell:

```
de_shell> report_libcell_subset -object_list [get_cells reg01]
```

To remove a library subset constraint from specified cells or a user-defined library subset, use the `remove_libcell_subset` command. The following example removes the library subset constraint from the reg01 and reg02 instances:

```
de_shell> remove_libcell_subset -object_list [get_cells {reg01 reg02}]
```

The following example removes the user-defined specialflops library subset:

```
de_shell> remove_libcell_subset -family_name specialflops
```

Optimizing Multicorner-Multimode Designs

DC Explorer analyzes and optimizes multicorner-multimode designs across multiple modes and multiple corners concurrently. This feature provides ease-of-use and compatibility between flows in the DC Explorer and IC Compiler tools due to the similar user interface.

The following topics describe multicorner-multimode design optimization:

- Multicorner-Multimode Concepts
- Unsupported Features for Multicorner-Multimode Designs
- Basic Multicorner-Multimode Flow
- Handling Libraries in the Multicorner-Multimode Flow
- Defining Scenarios
- Managing Scenarios
- Concurrent Multicorner-Multimode Optimization and Timing Analysis
- Reporting Commands for Multicorner-Multimode Designs
- Supported SDC Commands for Multicorner-Multimode Designs
- Using Block Abstractions in Multicorner-Multimode Designs

Multicorner-Multimode Concepts

Designs that operate under multiple operating conditions (or corners) and in multiple modes are referred to as multicorner-multimode designs.

The following terms are commonly used to describe multicorner-multimode technology:

- Corner

  A *corner* is an operating condition consisting of a set of process, voltage, and temperature (PVT) conditions. 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 multicorner-multimode analysis and optimization. Some constraints can be part of both the mode and corner specifications. Optimization of multicorner-multimode design involves managing the scenarios of the design.

See Also

- Managing Scenarios

Unsupported Features for Multicorner-Multimode Designs

DC Explorer does not support the following features for multicorner-multimode designs:

- Power-driven clock gating

  If you use the `compile_exploration -gate_clock` command, clock-gate insertion is performed on the design, independent of the scenarios.

- Clock tree estimation

- k-factor scaling

  Because multicorner-multimode design libraries do not support 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 link library list.

- The `set_min_library` command for individual scenarios.

  The library settings defined by the `set_min_library` command apply to all scenarios.

- The `-leakage_power` and `-dynamic_power` options of the `set_scenario_options`

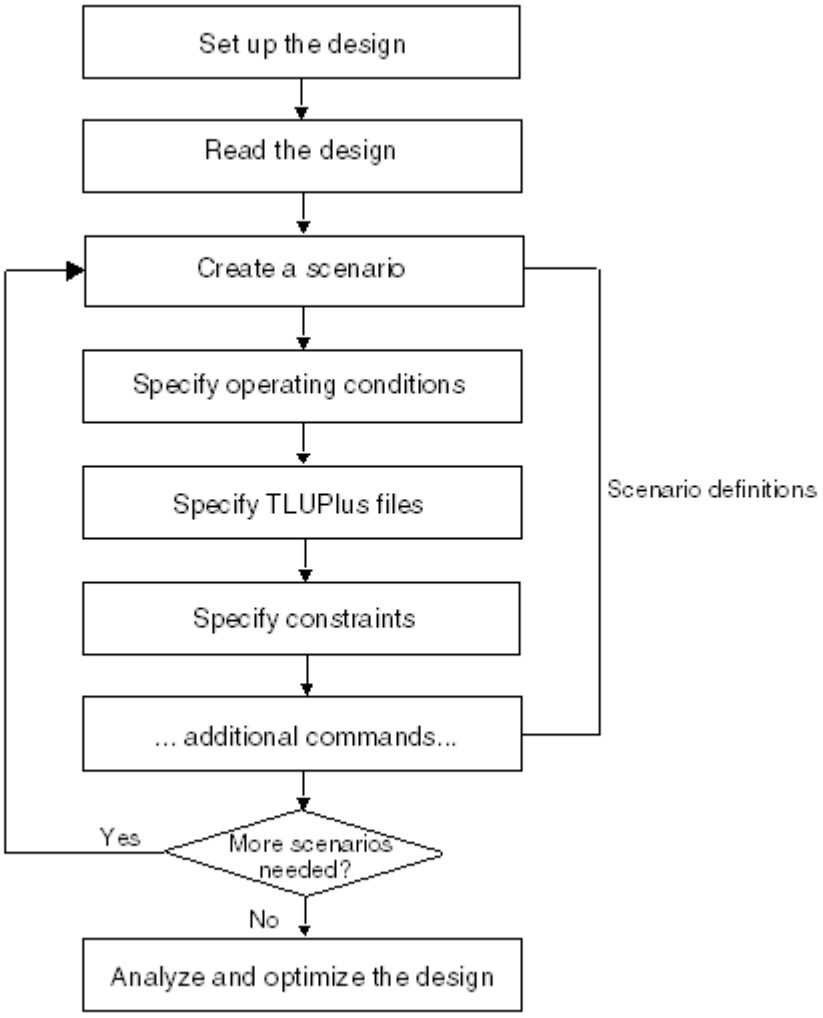
command

If you specify these options, the tool ignores the options and issues warning messages.

Basic Multicorner-Multimode Flow

Figure 10-8 shows the basic multicorner-multimode flow. The key steps involved in multicorner-multimode optimization are creating scenario definitions. At a minimum, a scenario definition includes commands that specify the scenario name, operating conditions, TLUPlus libraries, and constraints. You can create more than one scenario. For each scenario, you specify constraints specific to the scenario mode and operating conditions specific to the scenario corner.

Figure 10-8: ***Basic Multicorner-Multimode Flow***



Example 10-3 shows a basic script for the multicorner-multimode flow.

Example 10-3: ***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

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_exploration -scan -gate_clock
report_qor
report_constraint
report_timing -scenarios [all_scenarios]
...
compile_exploration -scan
```

Handling Libraries in the Multicorner-Multimode Flow

The following topics describe how to handle libraries in multicorner-multimode designs:

- Using Link Libraries With the Same PVT Nominal Values
- Using Distinct PVT Values

- Defining Minimum Libraries
- Automatic Detection of Driving Cell Libraries

Using Link Libraries With the Same PVT Nominal Values

The link library contains all the libraries for linking the design in all scenarios. Because a scenario can often use several libraries, such as a standard cell library and a macro library, the tool automatically groups the libraries into sets and identifies which set to link for each scenario.

The tool groups libraries that have the same PVT values into the same set. The tool uses the PVT values of the maximum operating conditions in a scenario definition to select the appropriate set for the scenario. If the tool finds no suitable cell in the link libraries, it issues an error message. You should verify the operating conditions and library setup to fix that error before optimizing the design.

Link Library Example

This table shows the libraries in the link library list, the nominal PVT values, and the operating conditions specified in the library. The design has instances of combinational, sequential, and macro cells.

Table 10-2: ***Link Libraries With PVT Values 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

The following example creates the s1 scenario with cell instances linked to the Combo_cells_slow, Macros_slow, and Sequentials_slow libraries, and the slow library is defined in the Combo_cells_slow file.

```
de_shell> create_scenario s1
de_shell> set_operating_conditions -max WORST -library slow
```

Using the `-library` option with the `set_operating_conditions` command helps the tool identify the correct PVT values for the operating conditions. During linking, the tool uses the PVT values of the maximum operating conditions to find the correct matches in the link library list.

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, one library file for standard cells and another for macros.

Preventing Inconsistent Libraries

For designs with multiple libraries, if library cells with the same name do not have identical functions, same pin names, and same pin order, the tool issues a warning indicating inconsistent libraries.

You should run the `check_library` command to check for inconsistent libraries before running the multicorner-multimode flow. For example,

```
de_shell> set_check_library_options -mcmm
de_shell> check_library -logic_library_name {a.db b.db}
```

When you specify the `-mcmm` option with the `set_check_library_options` command, the `check_library` command performs multicorner-multimode specific checks, such as identifying inconsistencies of operating conditions or power-down functions. When inconsistencies are detected, the tool generates a report that lists the inconsistencies and issues an information message.

Setting the dont_use Attribute on Library Cells

If you set the `dont_use` attribute on a library cell, the multicorner-multimode flow requires that all characterizations of this cell have the `dont_use` attribute. Otherwise, the tool might consider the libraries inconsistent. You can use the wildcard character (*) to set the `dont_use` attribute as follows:

```
de_shell> set_dont_use */AN2
```

When library cells that have the `dont_use` attribute have a different pin order from the library cells of various corners, the tool continues with the flow without issuing any error or warning messages. If you remove the `dont_use` attribute on these cells, the tool issues an MV-087 error message.

See Also

- Using Distinct PVT Values
- Automatic Detection of Driving Cell Libraries
- Defining Minimum Libraries

Using Distinct PVT Values

To ensure cell instances are linked correctly, use distinct nominal PVT values for operating conditions. The tool uses the PVT values to group the link libraries into sets, such as maximum libraries of different corners into one set. When the maximum libraries for individual scenarios do not have distinct PVT values, the cell instances might be incorrectly linked. That is, the cell

instances are linked to the first cell with a matching type in that set even though the `set_operating_conditions library -library` command specifies a scenario-specific library.

Example 10-4 shows the script that creates indistinct PVT values using the link libraries and operating conditions in Table 10-3 and Table 10-4. The tool groups the Ftyp.db and TypHV.db libraries into one set where Ftyp.db is the first library in the set. The cell instances in scenario s2 are not linked to the library cells in TypHV.db as intended, they are incorrectly linked to the library cells in Ftyp.db.

Table 10-3: ***Link Libraries With Nominal PVT Values and Operating Conditions***

Link library (in order)	Nominal PVT	Operating conditions (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 10-4: ***Scenarios and Their Operating Conditions***

	Scenario s1	Scenario s2	Scenario s3	Scenario 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 10-4: ***Indistinct PVT Values Causing Linking Problems***

```
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 values. The warning states that the libraries are ambiguous and identifies which libraries are being used and which are being ignored.

See Also

- Using Link Libraries With the Same PVT Nominal Values
- Automatic Detection of Driving Cell Libraries
- Defining Minimum Libraries

Defining Minimum Libraries

To define a minimum library for each scenario, use the `set_operating_conditions` command. If you use the `set_min_library` command 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 for all the scenarios.

A minimum library can be associated with multiple maximum libraries. As shown in Table 10-5, the Fast_0yr.db minimum library is associated with the Slow.db maximum library of scenario s1 and the SlowHV.db maximum library of scenario s2.

Table 10-5: ***Supported Library Configuration***

	Scenario s1	Scenario s2
Maximum library	Slow.db	SlowHV.db
Minimum library	Fast_0yr.db	Fast_0yr.db

However, you cannot associate two different minimum libraries, Fast_0yr.db and Fast_10yr.db, with the Slow.db maximum library in two separate scenarios as shown in Table 10-6. The tool applies the first minimum library to both scenarios in this case.

Table 10-6: ***Unsupported Multiple Minimum Library Configuration***

	Scenario s1	Scenario s2
Maximum library	Slow.db	Slow.db
Minimum library	Fast_0yr.db	Fast_10yr.db

See Also

- Using Distinct PVT Values
- Using Link Libraries With the Same PVT Nominal Values
- Automatic Detection of Driving Cell Libraries

Automatic Detection of Driving Cell Libraries

To build timing arcs for driving cells, specify different libraries for different scenarios by using the `-library` option with the `set_driving_cell` command. The tool searches for the specified

library in the link library set. If the specified library does not exist, the tool issues an error message.

The tool can automatically detect the driving cell library. If you do not specify the `-library` option with 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 conditions is used. If no library in the link library set matches the operating conditions, 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 an error message.

See Also

- Using Distinct PVT Values
- Defining Minimum Libraries
- Using Link Libraries With the Same PVT Nominal Values

Defining Scenarios

To setup a design for a multicorner-multimode flow, you must specify the TLUPlus files, operating conditions, and SDC constraints for each scenario. DC Explorer 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 values of the maximum operating conditions are used to select the appropriate set.

To create scenario definitions,

1. Create a scenario name by using the `create_scenario` command.

   When the first scenario is created, all previous scenario-specific constraints are removed from the design. For example,

   ```
   de_shell> create_scenario s1
   Warning: Any existing scenario-specific constraints
   are discarded. (MV-020)
   Current scenario is: s1>
   ```

2. Specify the operating conditions for each scenario by using the `set_operating_conditions` command. For example,

   ```
   de_shell> set_operating_conditions WORST \
   -library stdcell.setup.typ.db:stdcell_typ
   ```

3. Specify TLUPlus files for each scenario by using the `set_tlu_plus_files` command. For example,

   ```
   de_shell> set_tlu_plus_files \
   -max_tluplus design.tlup -tech2itf_map layermap.txt
   ```

To enable temperature scaling, the TLUPlus files must contain the `GLOBAL_TEMPERATURE` and `CRT1` variables, as shown in the following example. The `CRT2` variable is optional.

```
TECHNOLOGY = 90nm_lib
GLOBAL_TEMPERATURE = 105.0
CONDUCTOR metal8 {THICKNESS= 0.8000
CRT1=4.39e-3 CRT2=4.39e-7
 ...
```

4. Specify the design constraints for each scenario. For example,

```
de_shell> read_sdc s1.sdc
```

The tool discards any previous scenario-specific constraints when you run the `create_scenario` command in step 1.

5. (Optional) Set the scenario options, such as setup and hold delay optimizations by using the `set_scenario_options` command.
6. (Optional) Specify the correct net RC and pin-to-pin delay information by using the `read_sdf` command.

See Also

- Managing Scenarios

Managing Scenarios

After you created scenario definitions, you can manage the scenarios as described in these sections.

- Setting the Current Scenario
- Defining Active Scenarios
- Enabling Scenario Reduction
- Specifying Scenario Options

Setting the Current Scenario

To specify the current scenario, use the `current_scenario` command. When you define more than one scenario, the tool performs concurrent analysis and optimization across all scenarios, independent of the current scenario setting.

Defining Active Scenarios

During concurrent analysis and optimization, you can reduce memory usage and runtime by restricting 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, such as delay constraints associated with a pin, the design rule constraints for a net, and

leakage power. The scenario that has the worst slack value for one of its constrained objects 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`.

Enabling Scenario Reduction

By default, the tool uses the current scenario for optimization. To enable scenario reduction, you first set the `de_enable_physical_flow` variable to `true` to enable the physical flow.

Note: You need a DC-Extension license to run scenario reduction.

In the physical flow, restricting concurrent analysis and optimization to a subset of dominant scenarios does not lead to significant differences in QoR. The tool analyzes all the active scenarios to determine a set of dominant scenarios, based on the number of violating endpoints, and then optimizes the design for timing, power, and design rules. 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 preferred scenario as the most constraining one.

Specifying Scenario Options

To define specific constraint options for optimization in a scenario, use the `set_scenario_options` command. You can apply the constraint options to more than one scenario at a time by using the `-scenarios` option. The constraint options can be specified on both active and inactive scenarios. If you do not specify any scenario, the options are applied to the current scenario only.

To enable or disable the maximum delay optimization for the specified scenarios, use the `-setup` option. The default for the `-setup` option is `true`, which enables maximum delay optimization. The following example disables maximum delay optimization on specific scenarios:

```
de_shell> set_scenario_options -setup false -scenarios scenario_list
```

To enable or disable the 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 minimum delay violations in the specified scenarios.

To report the scenario options, use the command. To remove the specified scenarios, use the `remove_scenario` command. All scenario-specific constraints defined in the removed scenario are deleted.

See Also

- Defining Scenarios

Concurrent Multicorner-Multimode Optimization and Timing Analysis

The tool can perform concurrent multicorner-multimode optimization 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

  Define your analysis type by using the `bc_wc` keyword. For example,

  ```
  de_shell> set_operating_conditions -analysis_type bc_wc
  ```

- Early-late analysis

  Define your analysis type by using the `on_chip_variation` keyword. For example,

  ```
  de_shell> set_operating_conditions -analysis_type on_chip_variation
  ```

  The PrimeTime tool also uses this on-chip variation (OCV) method.

Reporting Commands for Multicorner-Multimode Designs

To report the scenario setup information for multicorner-multimode designs, including the logic libraries, operating conditions, and TLUPlus files, use the `report_scenarios` command. This command reports all the defined scenarios. To report the scenario options set by the `set_scenario_options` command, use the `report_scenario_options` command to specify a list of scenarios. By default, this command reports scenario options for the current, active scenario.

The following reporting commands support the `-scenario` option to report scenario-specific information:

Table 10-7: ***Reporting Commands That Support the -scenario Option***

Commands	
`report_timing`	`report_path_group`
`report_timing_derate`	`report_extraction_options`
`report_constraint`	`report_tlu_plus_files`
`report_clock`	

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:

Table 10-8: ***Commands That Report the Current Scenario***

Commands	
report_net	report_delay_estimation_options
report_annotated_check	report_transitive_fanout
report_annotated_transition	report_disable_timing
report_annotated_delay	report_net
report_attribute	report_power_calculation
report_case_analysis	report_timing_derate
report_ideal_network	report_timing_requirements
report_internal_loads	report_transitive_fanin
report_clock_gating_check	report_crpr
report_delay_calculation	report_clock_timing

Supported SDC Commands for Multicorner-Multimode Designs

This table lists the SDC commands supported in the multicorner-multimode flow.

Table 10-9: ***Supported SDC Commands***

Commands	
all_clocks	set_false_path
create_clock	set_fanout_load
create_generated_clock	set_input_delay
get_clocks	set_input_ transition
group_path	set_load
set_annotated_delay	set_max_ capacitance
set_case_analysis	set_max_delay
set_clock_gating_check	
	set_max_ transition

set_clock_latency	set_min_delay
set_clock_transition	set_multicycle_path
set_clock_uncertainty	set_output_delay
set_data_check	set_propagated_clock
set_disable_timing	set_resistance
set_drive	set_timing_derate

Using Block Abstractions in Multicorner-Multimode Designs

Block abstractions are compatible with multicorner-multimode scenarios. You can apply multicorner-multimode constraints to a block abstraction and use the block abstraction in a top-level design.

Note: You must set the `de_enable_physical_flow` variable to `true` when using block abstractions in multicorner-multimode designs.

When you use block abstractions with multicorner-multimode scenarios, follow these guidelines:

- For each top-level multicorner-multimode scenario, an identically named scenario must exist in each of the block abstraction blocks used in the design.

  When a mismatch occurs, use the `select_block_scenario` command to map the scenarios in the block abstraction to the top-level design. Mapping scenarios enables the tool to support name mismatches between the top-level design and the block abstraction.

  To reset user-specified multicorner-multimode scenario mapping, use the `-reset` option.

- A block abstraction can have additional scenarios that are not used at the top level.
- If a top-level design without multicorner-multimode scenarios, only block abstractions without multicorner-multimode scenarios can be used. Using the `select_block_scenario` command to specify scenario mapping allows you to use multicorner-multimode block abstractions even if the top-level design does not have multicorner-multimode scenarios.
- For each TLUPlus file that is used, the block abstraction stores the extraction data and the specified operating conditions. In the top-level design, you cannot use additional TLUPlus files or define additional temperature corners for the existing TLUPlus files.
- A top-level design with multicorner-multimode scenarios can be used with block abstractions that do not have any multicorner-multimode scenario definition. No log

message is issued in this case, and the same block data is used in all the top-level scenarios.

To learn how to use block abstractions with multicorner-multimode scenarios at the top level, see Using Block Abstractions With Scenarios at the Top Level.

See Also

- Using Block Abstractions With Scenarios at the Top Level
- Using Hierarchical Models

Using Block Abstractions With Scenarios at the Top Level

Follow these steps to use a block abstraction at the top level when it has scenario definitions:

1. Set the current design to the top-level design.
2. Remove all scenarios by using the `remove_scenario -all` command.
3. Define scenarios for the top-level design.

   The scenarios defined in the top-level design must match the scenario definitions in the block abstraction blocks of the design. In the top-level design, all the scenarios must be defined before the next step.
4. Perform optimization by using the `compile_exploration` command.

   At the beginning of compilation, the `compile_exploration` command performs the following checks to ensure no scenario mismatches between the top-level design and the block abstractions. The compilation is terminated when any of the following mismatches is encountered:

   - The number of scenarios in the top-level design does not match the number of scenarios in the block abstraction blocks.

     If the tool detects that the top-level design has more scenarios than the block abstraction blocks, the tool issues an error message and compilation is terminated.
   - The scenario definitions in the top-level design are not consistent with the scenario definitions in the block abstraction blocks.

     If scenarios are not defined in the top-level design and the block abstraction blocks have scenario definitions, the tool issues an error message and compilation is terminated.

You can also use the `check_scenarios` command to check consistency between scenarios.

See Also

- Using Block Abstractions in Multicorner-Multimode Designs

Performing Optimization With Floorplan Physical Constraints

The principal reason for using floorplan physical constraints is to accurately represent the placement area and to improve timing correlation with the post-layout design. Running placement-driven optimization improves the QoR and correlation between DC Explorer and Design Compiler topographical mode, especially for designs with complex floorplans.

By default, the optimization does not include any floorplan information. You can provide high-level floorplan physical constraints that determine core area and shape, port locations, macro locations and orientations, voltage areas, placement blockages, and placement bounds. These constraints can be derived from IC Compiler floorplan information, extracted from an existing Design Exchange Format (DEF) file, or created manually.

To perform optimization with floorplan physical constraints, follow these steps:

1. Set up the physical libraries.
2. Enable the physical flow capability (physical mode) by setting the `de_enable_physical_flow` variable.

   ```
   de_shell> set_app_var de_enable_physical_flow true
   ```

3. (Optional) Provide floorplan physical constraints.

   If no floorplan physical constraints are specified, the tool automatically derives the floorplan information.

4. Run the `compile_exploration` command.

   ```
   de_shell> compile_exploration
   ```

See Also

- Specifying Physical Libraries
- Using Floorplan Physical Constraints
- Congestion Reporting in the Physical Flow

Performing Power Optimization

When performing power optimization, you can enable clock gating, run leakage power optimization, and annotate the switching activity information of design objects. DC Explorer does not support the IEEE 1801 Unified Power Format (UPF), but it can handle previously instantiated power management cells, such as isolation cells and level-shifter cells.

- To enable clock gating, enter

```
de_shell> compile_exploration -gate_clock
```

- To enable leakage power optimization, use the `set_multi_vth_constraint` command. For example,

```
de_shell> set_multi_vth_constraint -lvth_groups {lvt svt} \
 -lvth_percentage 15 -type soft -cost cell_count
```

  When running this command in DC Explorer, you should specify the `cell_count` keyword with the `-cost` option for the percentage of low threshold-voltage cells in the design.

- To annotate the switching activity information of the design nets, use the `set_switching_activity` command. For example,

```
de_shell> set_switching_activity [get_net net1] \
 -static_probability 0.2 -toggle_rate 10 -period 1000
```

Note: DC Explorer does not support SAIF files.

See Also

- The *Power Compiler User Guide*

Pipelined-Logic Retiming

You enable pipelined-logic retiming by using the `set_optimize_registers` command.

When you describe circuits at the RTL level before logic synthesis, it is very difficult and time-consuming to find the optimal register locations and code them into the HDL description. With register retiming, the locations of the flip-flops in a sequential design can be automatically adjusted to equalize as nearly as possible the delays of the stages.

The retiming techniques that the `set_optimize_registers` command uses are particularly useful to pipeline datapath blocks. You first specify the required number of pipeline stages and place all the pipeline registers at the inputs or outputs of the combinational logic in your RTL. During retiming, the tool moves these pipeline registers into the combinational logic, balances the delay of each pipeline stage, and archives the optimal timing and area results based on the given clock period.

Register retiming leaves the behavior of the circuit at the primary inputs and primary outputs unchanged (unless you add pipeline stages or choose special options that do not preserve the reset state of the design). Therefore, you do not need to change any simulation test benches developed for the original RTL design.

However, retiming changes the location, contents, and names of registers in the design. A verification strategy that uses internal register inputs and outputs as reference points will no longer work. Retiming can also change the function of hierarchical cells inside a design and add clock, clear, set, and enable pins to the interfaces of the hierarchical cells.

Pipelined-Logic Retiming Command

You can use the `set_optimize_registers` command followed by the `compile_exploration` command to enable pipelined-logic retiming.

This command sequence performs retiming during synthesis when you run the `compile_exploration` command after reading the RTL. You must run the `set_optimize_registers` command before running `compile_exploration`. The `set_optimize_registers` command sets the `optimize_registers` attribute on the specified design or on the current design so that the attribute is automatically invoked during compile and the design is retimed during optimization.

This method is also known as one-pass retiming because retiming is performed in one step by the `compile_exploration` command.

Register Retiming Example

During retiming, registers are moved forward or backward through the combinational logic of a design. The following figures illustrate an example of delay reduction through backward retiming of a register.

Figure 10-9: Circuit Before Retiming

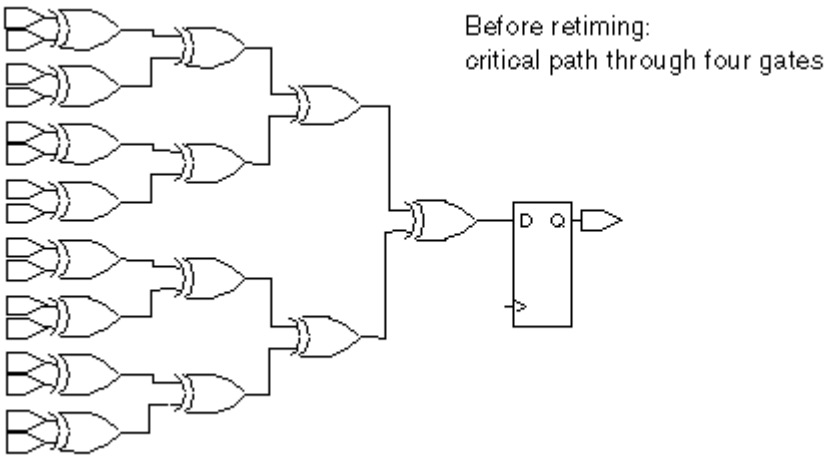
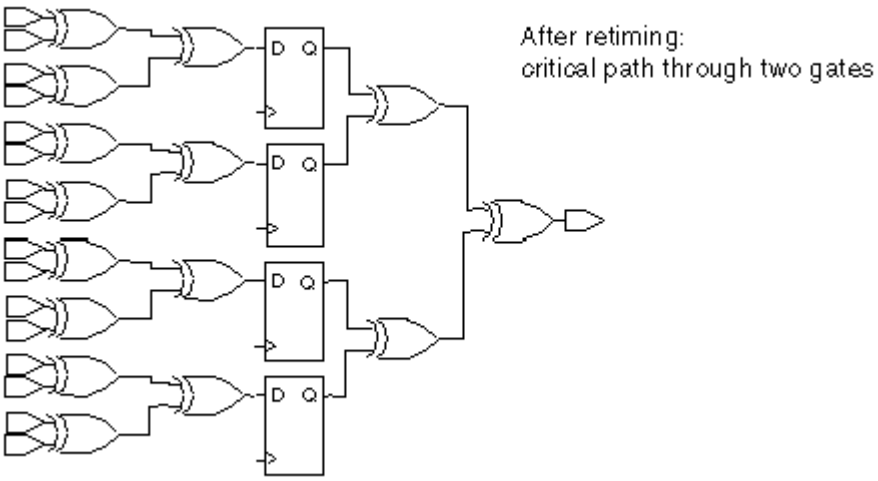


Figure 10-10: Circuit After Retiming



In this example, before register retiming, there are four levels of combinational logic and only one register at the endpoint of the critical path. After retiming, the register that has been replaced by four registers has been moved back through two levels of logic, and the critical path now consists of two stages. The critical path delay in each stage is less than the critical path delay in the initial single stage design. As in this example, delay reduction through retiming often leads to an increase in the number of registers in the design, but usually this increase is small.

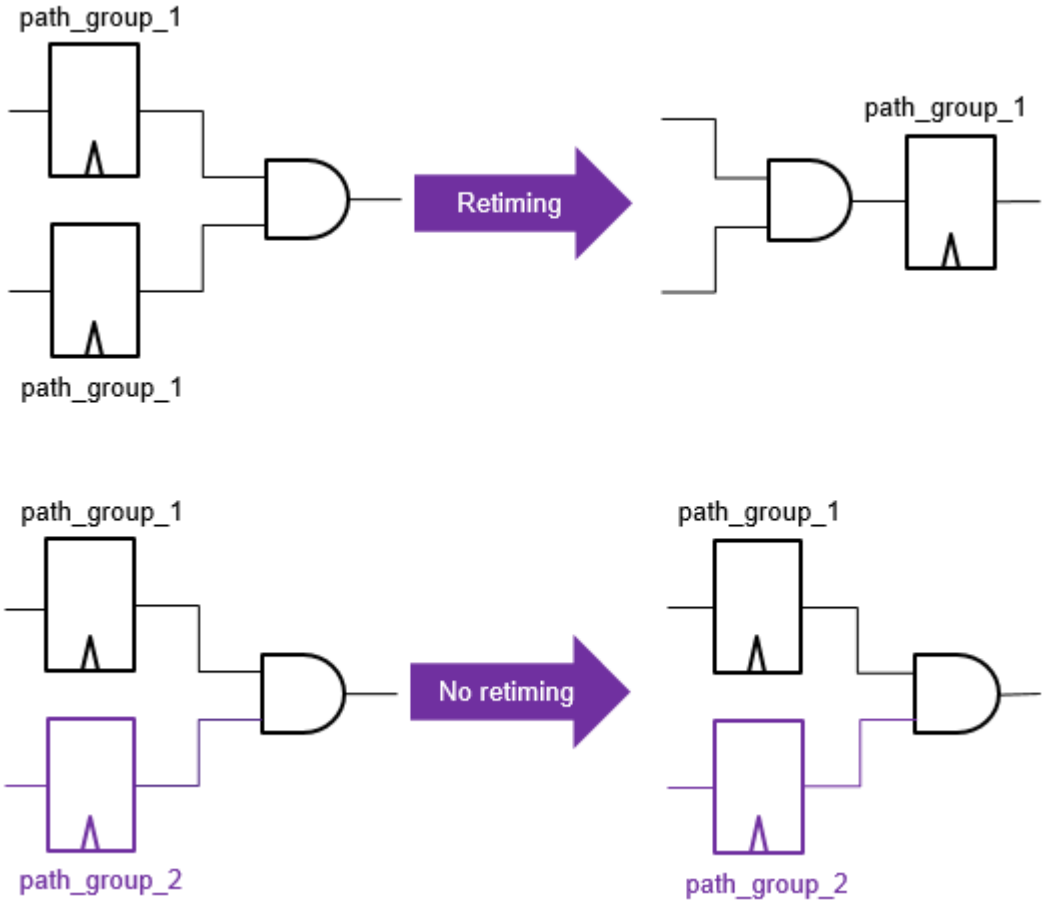
Retiming Registers Containing Path Group Constraints

Registers with path group constraints can be retimed if the new registers inherit the constraints from the original registers and contain no other timing exceptions except those defined on the path groups. Registers that belong to different path groups are considered separately during pipelined-logic retiming.

The tool allows the following pipelined-logic retiming for registers with path group constraints:

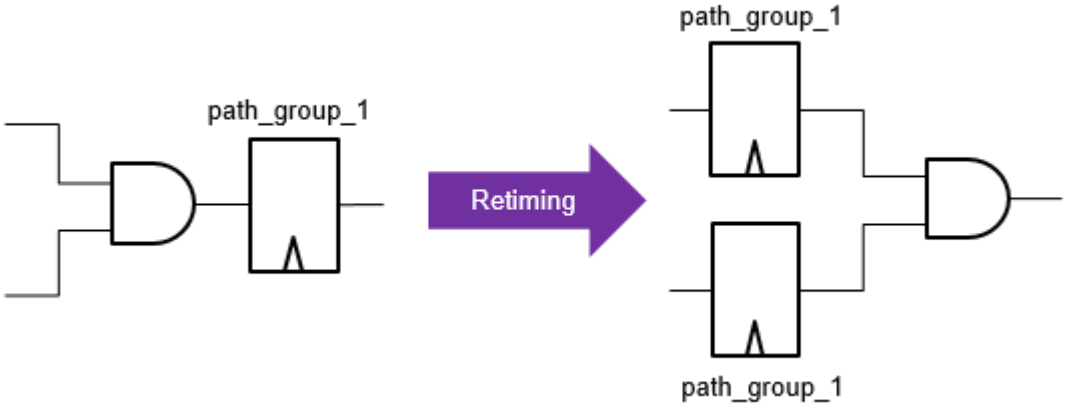
- Forward moves

  The tool allows forward moves when the registers belong to the same path groups; otherwise, no retiming is performed. Registers created as a result of combinational logic through forward moves inherit the path group constraints from the original registers before retiming. For example,



- Backward moves

  Duplicate registers created as a result of backward moves inherit the path group constraints from the original registers before retiming. For example,



- Retiming in multicorner-multimode designs

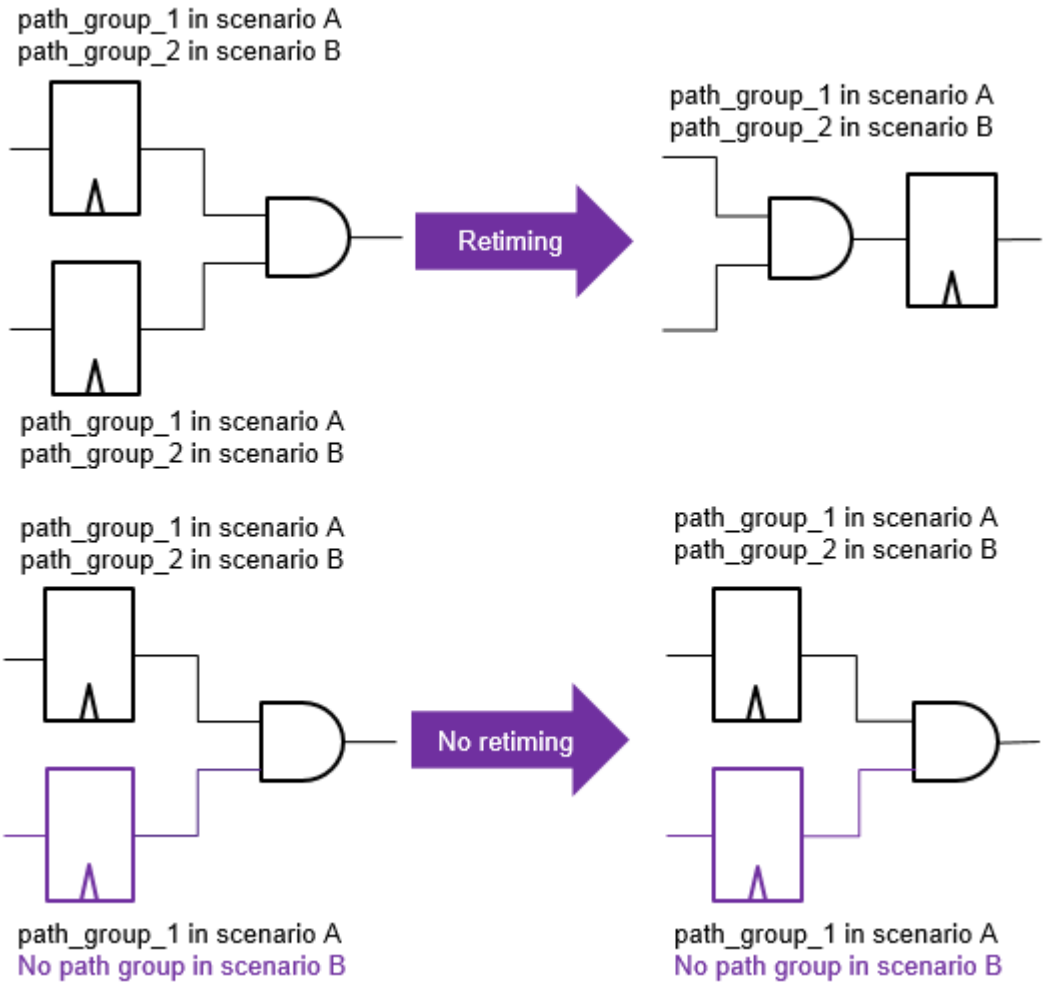
  The tool identifies registers that belong to the same path groups in all scenarios and performs retiming. Retiming is not allowed across path groups.

  As shown in the following table, the reg_0 and reg_1 registers share the same path group specifications of all multicorner-multimode scenarios, so they are grouped together during retiming. However, the reg_2 and reg_3 registers do not share the same path group specifications; they remain two separate groups.

Scenario	reg_0	reg_1	reg_2	reg_3
S0	pg_A	pg_A	pg_A	pg_B
S1	pg_C	pg_C	pg_C	pg_C
S2	no group	no group	pg_D	pg_D
	group_01		group_2	group_3

- Forward moves in multicorner-multimode designs

  The tool allows forward moves only when the registers share the same path group specifications across all scenarios. For example,



The tool does not allow retiming in the following situations:

- Path groups are created using registers pins instead of registers. For example,

```
de_shell> create_path_group -to A_reg/D
```

- Path groups include paths originating from and ending at the same register. For example,

```
de_shell> create_path_group \
   -from A [get_cells ff_reg*] -to [get_cells ff_reg*]
```

Additional Optimization Features

You can use the optimization techniques described in these topics to improve optimization results:

- Preserving Subdesigns
- Fixing Multiple-Port Nets
- Optimizing Multibit Registers
- Congestion Reporting in the Physical Flow
- Controlling Path Group Creation

See Also

- The *Design Compiler User Guide*

Preserving Subdesigns

To preserve a subdesign during optimization, use the `set_dont_touch` command. The command 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.

Note: Any block abstractions in your design are automatically marked with the `dont_touch` attribute. Also, the cells of block abstractions are marked as don't touch.

Use the `set_dont_touch` command to specify subdesigns not to be optimized with the rest of the design hierarchy. The `dont_touch` attribute does not prevent or disable timing through the design.

When you use the `set_dont_touch` command, remember the following points:

- Setting the `dont_touch` attribute on a hierarchical cell sets an implicit don't touch value on all cells below that cell.
- Setting the `dont_touch` attribute on a library cell sets an implicit don't touch value on all instances of that cell.
- Setting the `dont_touch` attribute on a net sets an implicit don't touch value only on mapped combinational cells connected to that net. If the net is connected only to generic logic, optimization might remove the net.
- Setting the `dont_touch` attribute on a reference sets an implicit don't touch value 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 the `dont_touch` attribute on the top-level design has no effect because the top-level design is not instantiated within any other design.

- You cannot ungroup objects marked with the `dont_touch` attribute by using the `ungroup` command. The automatic ungrouping of DC Explorer has no effect on don't touch objects.

Note: The `dont_touch` attribute is ignored on synthetic part cells (for example, many of the cells read in from an HDL description) and on nets that have unmapped cells on them. During compilation, warnings appear for don't touch nets connected to unmapped cells (generic logic).

To remove the `dont_touch` attribute, use the `remove_attribute` command or the `set_dont_touch` command set to `false`.

See Also

- Using Hierarchical Models

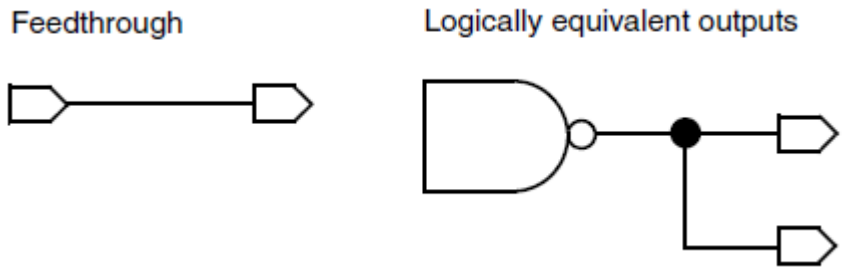
Fixing Multiple-Port Nets

To fix nets connected to multiple ports, you can use the `set_fix_multiple_port_nets` command to insert buffers or inverters to the nets, to duplicate the logic, or to rewire the logic.

As shown in the following figure, nets that are connected to multiple ports include

- Feedthrough nets, where an input port feeds into an output port
- Nets connected to multiple output ports, logically equivalent outputs

Figure 10-11: Nets Connected to Multiple Ports



These multiple-port nets, which are represented with the `assign` statements in the gate-level netlist, might cause design rule violations in back-end tools. To prevent multiple-port nets in the netlist, follow these steps:

1. Enable the tool to fix multiple-port nets and constant-driven ports by setting the `compile_advanced_fixed_multiple_port_nets` variable to `true`.

The default is set to `true` in DC Explorer.

2. Use the `set_fix_multiple_port_nets` command with the appropriate options to set the fix_multiple_port_nets attribute on the designs.

   For example, the following command specifies to insert buffers to both feedthrough nets and logic constants and to restrict each driver cell to only one output port on the current design:

   ```
   de_shell> set_fix_multiple_port_nets -all -buffer_constants
   ```

   - To disable fixing the multiple-port nets in clock network, specify the `-exclude_clock_network` option.
   - To disable rewiring the logic before buffer insertion, specify the `-no_rewire` option.

3. Compile the design by running the `compile_exploration` command.

   The tool fixes multiple-port nets in the designs marked with the `fix_multiple_port_nets` attribute by inserting buffers, duplicating the logic, or rewiring the logic according to the specified options.

When DC Explorer fixes multiple-port nets and constant-driven ports, the tool performs the following tasks in sequence:

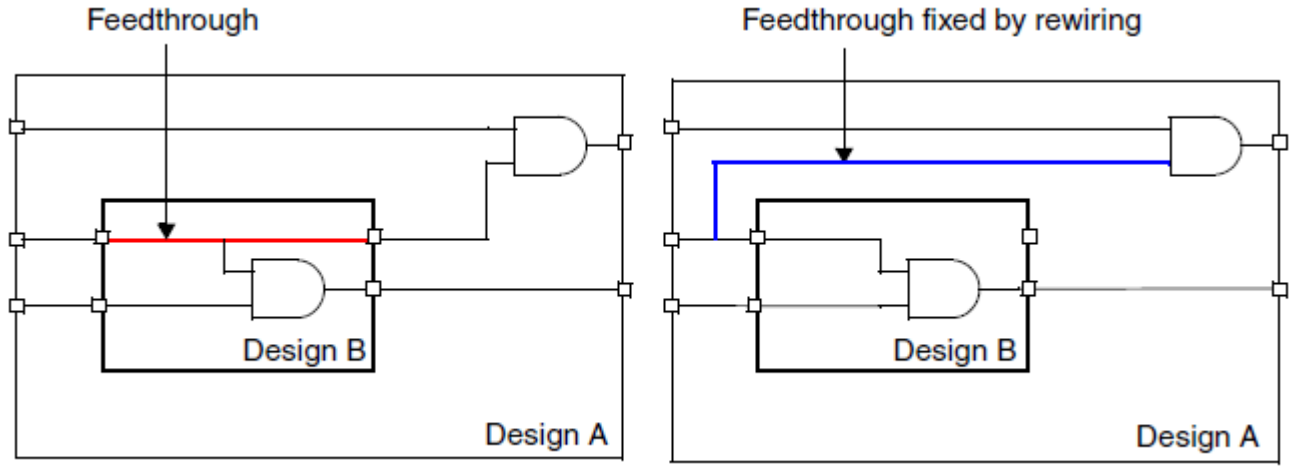
1. Rewire the connections to ensure that each of these nets is connected to one hierarchical port.

   The tool first performs rewiring to avoid unnecessary buffering. The tool also rewires across the hierarchy because boundary optimization is enabled by default.

2. Insert buffers or inverters to the nets that are not fixed by rewiring.

This figure shows that the tool fixes a feedthrough in Design B by rewiring across the hierarchy.

Figure 10-12: Feedthrough Fixed by Rewiring Across the Hierarchy



This table summarizes the options of the `set_fix_multiple_port_nets` command.

Table 10-10: ***set_fix_multiple_port_nets Command Options***

To do this	Use this option
Disable fixing multiple-port nets.	`-default`
Insert buffers to prevent feedthrough nets.	`-feedthroughs`
Insert buffers so that each driver cell drives one output port.	`-outputs`
Duplicate logic constants so that each logic constant drives one output port.	`-constants`
Equivalent to `-feedthroughs -outputs -constants`	`-all`
Insert buffers to logic constants instead of duplicating the logic.	`-buffer_constants`
Disable rewiring before the tool fixes multiple-port nets and constant-driven ports.	`-no_rewire`
Exclude fixing the multiple-port nets in clock network.	`-exclude_clock_network`

Optimizing Multibit Registers

DC Explorer can replace single-bit register cells with multibit register cells if such cells are available in the logic library. Using multibit register cells in place of single-bit cells can reduce clock tree power, the number of clock tree buffers, clock tree net length, and area.

Multibit Registers

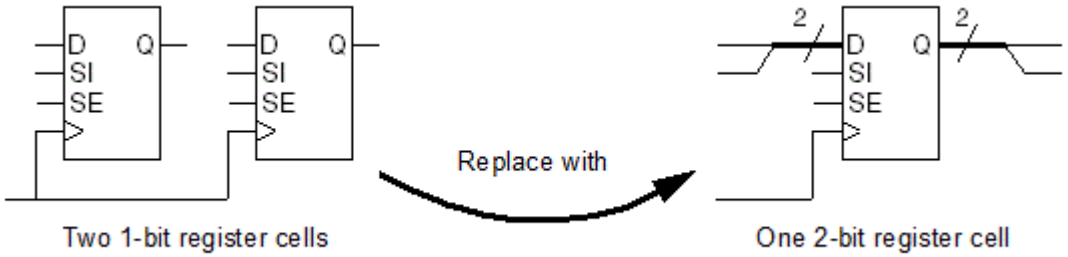
The tool represents register bits using single-bit registers. You can instruct the tool to organize multiple register bits into groups called multibit components in the RTL bus inference flow. The register bits in the group are targeted for implementation using multibit registers. For example, a group of eight register bits can be implemented as one 8-bit library register or two 4-bit library registers.

Replacing single-bit cells with multibit cells, as shown in Figure 10-13, can reduce

- Clock tree power and the number of clock tree buffers
- The total clock tree net length
- Area due to shared transistors and optimized transistor-level layout

These benefits must be balanced against the loss of optimization flexibility in placing the register bits and routing the connections.

Figure 10-13: Replacing Two Multiple Single-Bit Register Cells With One Multibit Register Cell



Running the RTL Bus Inference Flow

When you run the RTL bus inference flow, the tool groups the register bits belonging to each bus (as defined in the RTL) into multibit components. The bits in each multibit component are targeted for implementation using multibit registers. The actual replacement of register bits occurs during execution of the `compile_exploration` command. You can also group bits manually by using the `create_multibit` command.

To run the RTL bus inference flow,

1. Direct the tool how to infer multibit registers from the RTL buses by setting the `hdlin_infer_multibit` variable.

   The following example specifies to infer multibit registers for all buses:

   ```
   de_shell> set_app_var hdlin_infer_multibit default_all
   ```

2. Read the RTL.
3. (Optional) Specify the inference mode and the minimum width for multibit registers by using the `set_multibit_options` command.

   The command sets the `multibit_mode` and `minimum_multi_width` directives for multibit registers in the current design.

4. (Optional) Manually group specific single-bit cells or instances into multibit components by using the `create_multibit` command.

   You must have a mapped design or GTECH netlist to use this command. The instances and cells must not belong to another multibit component. For example,

   ```
   de_shell> create_multibit x_reg[0] x_reg[1] -name my_multi
   ```

5. (Optional) Exclude a particular set of single-bit registers from being grouped into a multibit component by using the `remove_multibit` command.

   You must have a mapped design or GTECH netlist to use this command. For example, the following command removes the y_reg multibit component, causing its register bits to

be ungrouped during multibit synthesis.

```
de_shell> remove_multibit y_reg
```

6. Compile the design by using the `compile_exploration` command.

   When you specify the `-scan` option, the command replaces single-bit cells with scan cells before multibit synthesis. For example,

   ```
   de_shell> compile_exploration -gate_clock -scan
   ```

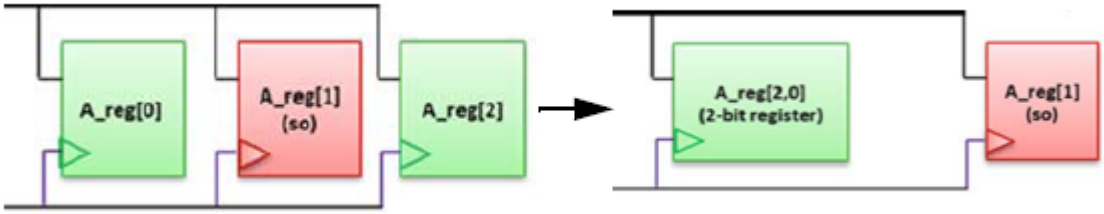
7. Report multibit components by using the `report_multibit` command.

   The command reports all multibit components whether they are inferred by the tool from the RTL or created by the `create_multibit` command.

Noncontiguous Multibit Mapping During Compile

The `compile_exploration` command can map the noncontiguous bits of a multibit component to a multibit register. As shown in Figure 10-14, the command packs the noncontiguous A_reg[0] and A_reg[2] registers into the two-bit register, A_reg[2,0]. Because the A_reg[1] register is marked with the `size_only` attribute, it is preserved as a single-bit register.

Figure 10-14: Noncontiguous Multibit Mapping



See Also

- The *Multibit Register Synthesis and Physical Implementation Application Note*

Congestion Reporting in the Physical Flow

DC Explorer performs congestion-driven placement to reduce routing congestion during the physical flow. To enable the physical flow capability (physical mode), set the `de_enable_physical_flow` variable to `true`, changing it from its default of `false`. Running congestion-driven placement improves the correlation between DC Explorer and the Design Compiler topographical mode, especially for designs with complex floorplans.

To report the congestion status of the design in the physical flow, use the commands listed in Table 10-11. When you run these commands, the tool uses Zroute to estimate the congestion information.

Table 10-11: Commands for Congestion Reporting

Command	Description
`report_congestion`	Generates a global route congestion report.
`report_congestion_options`	Reports the specified congestion options for the current design.
`set_congestion_options`	Sets congestion options for congestion optimization.
`remove_congestion_options`	Removes the specified congestion options from the current design.

In addition, you can use the commands in Table 10-12 to set route options for congestion estimation or to better understand what route options are used for the congested areas.

Table 10-12: Commands for Route Options

Command	Description
`set_route_zrt_common_options`	Sets the options that are common to global routing, track assignment, and detail routing.
`get_route_zrt_common_options`	Returns the value of a specified common route option.
`report_route_zrt_common_options`	Reports the settings of all common route options.
`set_route_zrt_global_options`	Sets the global route options.
`get_route_zrt_global_options`	Returns the value of a specified global route option.
`report_route_zrt_global_options`	Reports the settings of all global route options.
`create_route_guide -horizontal_track_utilization -vertical_track_utilization -track_utilization_layers`	Creates a route guide with the settings of these three options.

Note: DC Explorer does not support the `set_congestion_optimization` command.

In cases where the design is reported as significantly congested, you can further analyze congestion by generating a congestion map in the GUI. By viewing the congestion map, you can identify areas of high congestion and determine whether the design can be routed. For information about generating and analyzing the congestion map, see the *Design Vision User Guide* or Design Vision Help.

Reading and Writing Cell Expansion Data

During congestion optimization, cells in congested regions are expanded to occupy more space to create lower density areas. The tool generates cell expansion data so that the data can be used for better congestion correlation with the IC Compiler and IC Compiler II tools. The cell expansion data includes the area, width, and height of each expanded cell. You can read and write the cell expansion data by using the `read_cell_expansion` and `write_cell_expansion` commands, respectively.

For example, the following command writes cell expansion data to the out.exp file:

```
de_shell> write_cell_expansion out.exp
```

Table 10-13: Commands for Reading and Writing Cell Expansion Data

Command	Description
`read_cell_expansion`	Reads the cell expansion data from a file to support ASCII flow.
`write_cell_expansion`	Writes the cell expansion data to a file to support ASCII flow.

See Also

- Performing Optimization With Floorplan Physical Constraints
- The "Analyzing Congestion" topic in Design Vision Help
- The "Setting Zroute Options" topic in the *IC Compiler User Guide*

Controlling Path Group Creation

Use the `create_auto_path_groups` command to automatically create specific path groups for I/Os, macros, and integrated clock-gating cells.

Improving QoR

You can direct the tool to automatically create path groups to improve QoR. To do so, specify the `-mode rtl` option with the `create_auto_path_groups` command after elaboration but before optimization.

For example,

```
de_shell> create_auto_path_groups -mode rtl
de_shell> compile_exploration
de_shell> remove_auto_path_groups
de_shell> report_qor
```

The command saves user-created path groups before automatically creating path groups, one path group per hierarchy in RTL mode. If there are multiple levels of hierarchy, the tool creates path groups for each level. To generate a QoR report with only user-specified path groups, use the `remove_auto_path_groups` command to remove the path groups created by the tool. This

command removes all path groups, but it keeps the path groups that you created with the `group_path` command.

Preventing Path Group Creation

To prevent the creation of specific path groups, specify the `-exclude` option to list a combination of the following values: `IO`, `macro`, and `ICG`. For example, the following command prevents the creation of specific path groups for I/Os and macros:

```
de_shell> create_auto_path_groups -mode rtl -exclude {IO macro}
```

In RTL mode, to prevent the creation of path groups for hierarchies that have a small number of registers, specify the `-min_regs_per_hierarchy` option. The option allows you to specify the minimum number of registers per hierarchy. No path group is created for hierarchies with a number of registers lower than this value. The default is 10.

See Also

- Reporting Quality of Results

11 Using Hierarchical Models

When performing top-level optimization on large designs, you can use hierarchical models to reduce the number of design objects and the memory requirements. The term hierarchical models refers to block abstractions and physical hierarchies.

To learn how to use hierarchical models in DC Explorer, see

- Overview of Hierarchical Models
- About Block Abstractions

See Also

- Performing a Bottom-Up Compile

Overview of Hierarchical Models

Hierarchical models are used in DC Explorer to reduce the number of design objects and the memory requirements during top-level optimization on large designs.

For interface logic models (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. However, DC Explorer does not support ILMs; it uses block abstractions instead.

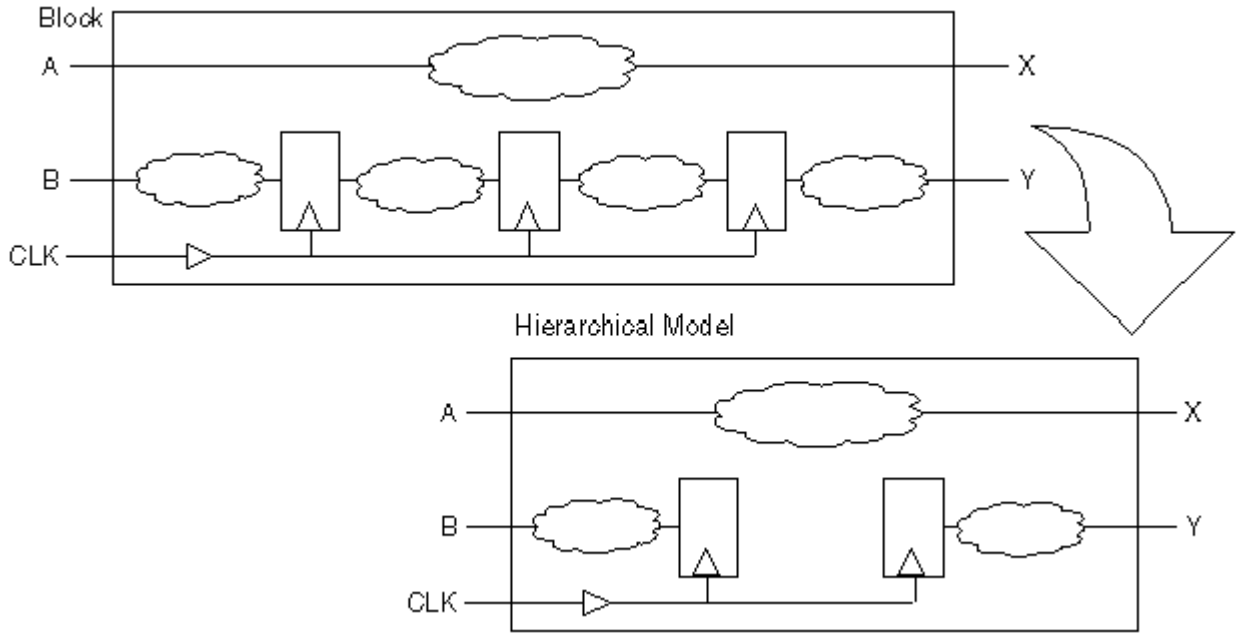
Block abstractions are an extension to interface logic models. Unlike an ILM where the internal logic is removed and saved in a separate .ddc file, when you use block abstractions, the full design and the block abstraction information are both saved in the same .ddc file. The interface logic is marked, and the tool loads only the interface logic when reading the block abstraction. DC Explorer can create block abstractions and support block abstractions created in the IC Compiler tool. However, the IC Compiler tool only supports block abstractions created in the IC Compiler tool.

Figure 11-1 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 11-1: A Block and Its Hierarchical Model



DC Explorer supports the following flows that use hierarchical models:

- Multicorner-multimode

  The tool 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

  The tool supports multiple levels of physical hierarchy only through the use of block abstractions.

To learn more about hierarchical models, see

- Information Used in Hierarchical Models
- Hierarchical Models in a Multicorner-Multimode Flow
- Viewing Hierarchical Models in the GUI

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

  DC Explorer treats generated clocks like clock ports and 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.

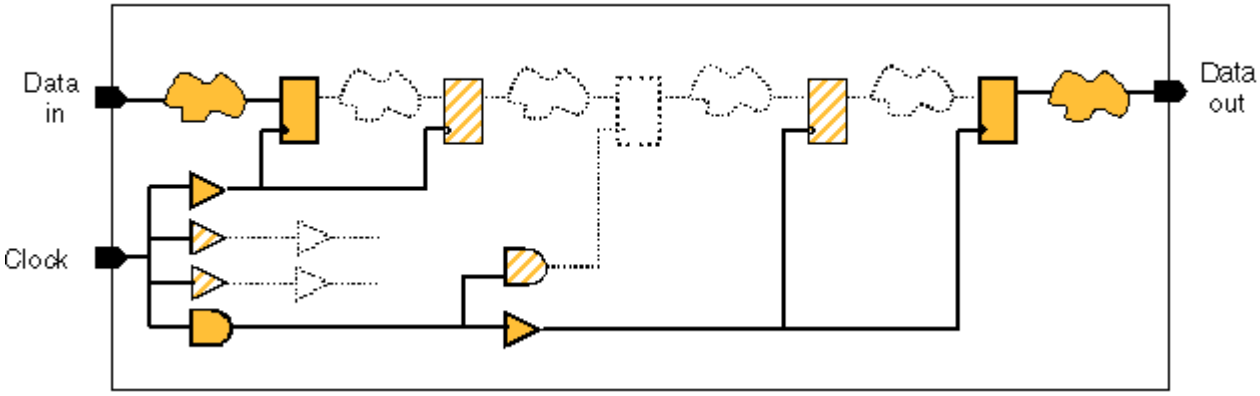
Note: By default, latches are treated as combinational logic.

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
- UPF constraints

Figure 11-2 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 11-2: Hierarchical Model



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.

- 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. The tool automatically uses the same block-level data, such as timing and power, for each active scenario defined at the top level.

- 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.

Note: DC Explorer supports the creation of multiple scenarios and optimization in the current scenario only; however, the tool does not perform scenario reduction.

Viewing Hierarchical Models in the GUI

In the DC Explorer 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 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, unexpanded view, type or select 0 in the “Level” box.

See Also

- The “Examining Physical Hierarchy Blocks” topic in Design Vision Help

About Block Abstractions

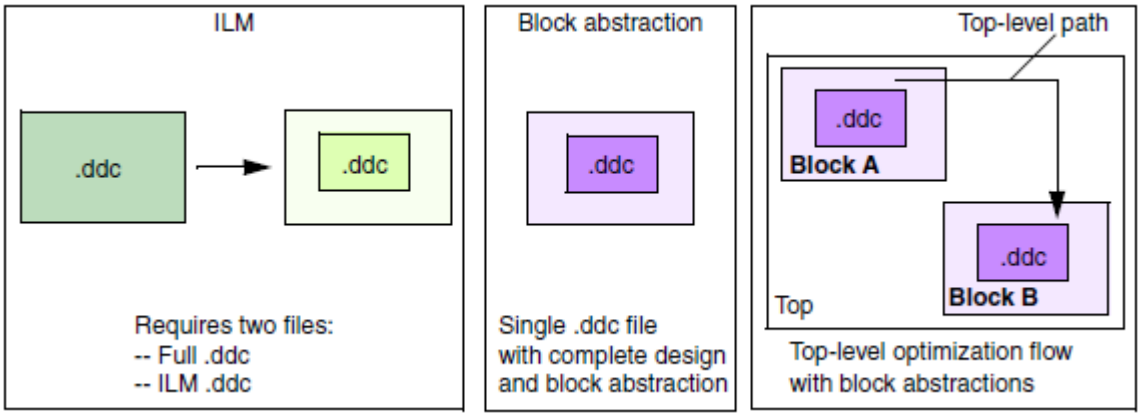
A top-level flow using block abstractions allows optimization of the top-level logic.

Block abstractions are an extension to interface logic models (ILMs). Unlike an ILM where the internal logic is removed and saved in a separate .ddc file, when you use block abstractions, the full design and the block abstraction information are both saved in the same .ddc file. When reading the design in as a block abstraction at the top level, the tool loads only the interface logic from the .ddc file.

Note: DC Explorer does not support ILMs. Block abstractions should be used instead.

Figure 11-3 illustrates the difference between an ILM and a block abstraction and shows how Design Compiler topographical supports a top-level optimization using block abstractions. Block abstractions cannot be optimized in DC Explorer.

Figure 11-3: ILM and Block Abstraction Comparison



To learn the hierarchical flow and the limitations of block abstractions, see

- Block Abstraction Hierarchical Flow
- Limitations

Block Abstraction Hierarchical Flow

The DC Explorer tool can create block abstractions and supports block abstractions created in the DC Explorer, Design Compiler, or IC Compiler tool. However, the IC Compiler tool only supports block abstractions created in the IC Compiler tool.

To run the block abstraction hierarchical flow, perform the tasks described in the following topics in sequence:

1. Creating and Saving Block Abstractions
2. Setting Top-Level Design Options
3. Controlling the Extent of Logic Loaded for Block Abstractions
4. Loading Block Abstractions
5. Checking Block Abstraction Readiness
6. Performing Top-Level Synthesis

To ignore the timing paths that reside entirely in block abstractions during top-level synthesis, see Ignoring Timing Paths Within Block Abstractions.

See Also

- Limitations
- *IC Compiler Implementation User Guide*

Creating and Saving Block Abstractions

When you create and save block abstractions, they are stored in a single .ddc 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.

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:

```
de_shell> create_block_abstraction
de_shell> write_file -hierarchy -format ddc \
-output ${DESIGN_NAME}.mapped.ddc
```

The .ddc file contains both the complete design and the DC Explorer block abstraction.

To specify a list of block references to be linked to block abstractions and integrated with the top-level design, use the `set_top_implementation_options` command as described in Setting Top-Level Design Options.

See Also

- Block Abstraction Hierarchical Flow

Setting Top-Level Design Options

To use block abstractions at the top-level, use the `set_top_implementation_options` command with the `-block_references` option to specify a list of block references to be linked to the block abstraction. Do this for both DC Explorer 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 DC Explorer. The IC Compiler block

abstractions are automatically read from the list of Milkyway reference libraries when the tool links the top-level design.

The following example sets the top-level implementation options for the blk1 and blk2 blocks:

```
de_shell> set_top_implementation_options -block_references {blk1 blk2}
```

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 11-1 and Example 11-2 are equivalent.

Example 11-1: Specifying Options for Block Abstractions in One Command

```
de_shell> set_top_implementation_options -block_references {blk1 blk2}
```

Example 11-2: Specifying Options for Block Abstractions in Multiple Commands

```
de_shell> set_top_implementation_options -block_references {blk1}
de_shell> 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, use the `report_top_implementation_options` command.

To reset the top implementation options, use 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:

```
de_shell> set_top_implementation_options -block_references {blk1 blk2}
de_shell> set_top_implementation_options -block_references {blk2} -reset
```

See Also

- Block Abstraction Hierarchical Flow

Controlling the Extent of Logic Loaded for Block Abstractions

When loading blocks as block abstractions at the top level, you can control the extent of logic loaded either with a full interface or a compact interface by using the `-load_logic` option with the `set_top_implementation_options` command.

When you choose to load blocks with a compact interface, only the min/max and rise/fall interface logic is retained. For blocks with ports that fan out to a large number of boundary registers, using a compact interface can improve runtime and memory usage. When you create a block abstraction by using the `create_block_abstraction` command, the tool identifies both full interface logic and compact interface logic at the same time.

To control the extent of logic loaded for block abstractions, use

- The `-load_logic full_interface` option to load the full interface, which is the default.

  The following logic is marked as interface logic:

  - All the logic from input ports to registers or output ports
  - All the logic to output ports from registers or input ports

- The `-load_logic compact_interface` option to load the compact interface, consisting of only the logic for the timing-critical paths of every input and output port.

  The following logic is marked as interface logic:

  - The logic that belongs to min/max rise/fall from input ports to registers or output ports
  - The logic that belongs to min/max rise/fall to output ports from registers or input ports

The following clock circuitry is marked as interface logic whether you load the full interface or the compact interface:

- Any logic in the path from the master clock to the 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

The tool retains the following IEEE 1801 Unified Power Format (UPF) objects in a block abstraction whether you load the full interface or the compact interface:

- All retention registers, whether or not they are part of the interface logic
- Level-shifter cells and isolation cells that are part of the interface logic
- Control logic for the retained retention registers, isolation cells, and level-shifter cells

When the tool removes a UPF netlist object, it either removes or updates the associated UPF constraint. The tool ensures that the UPF strategies stored in the block abstraction are consistent with the UPF netlist objects.

To propagate the UPF objects in the block abstraction to the top level, run the `propagate_constraints -power_supply_data` command.

See Also

- Block Abstraction Hierarchical Flow

Loading Block Abstractions

Use the `read_ddc` command after you have used the `set_top_implementation_options` command to load DC Explorer block abstractions to be linked to the top-level design:

```
de_shell> read_ddc {DesignA.mapped.ddc DesignB.mapped.ddc}
```

The following message indicates that the DC Explorer block abstraction is being loaded while reading in the block .ddc file:

```
Information: Reading block abstraction for design 'blk1' and its
hierarchy.(DDC-27)
```

Ensure 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)
```

See Also

- Block Abstraction Hierarchical Flow

Checking Block Abstraction Readiness

To determine whether the loaded block abstractions are ready for top-level synthesis, you can perform the following tasks:

- Reporting Block Abstractions

  To report block abstractions that are linked to the current design, use the `report_block_abstraction` command.

  For block abstractions created by the IC Compiler tool, the `report_block_abstraction` command also reports UPF information.

- Querying Block Abstractions

  To create a collection of block abstractions, use the `get_cells` command:

  ```
  de_shell> get_cells -hierarchical -filter is_block_abstraction==true
  {blk1 blk2}
  ```

- Checking Block Abstractions

  To check the readiness of a block abstraction before optimization, use the `check_block_abstraction` command. 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.

  You can also use the `compile_exploration -check_only` command to verify that the design and libraries have all the data that the `compile_exploration` command requires to run successfully.

See Also

- Block Abstraction Hierarchical Flow

Performing Top-Level Synthesis

Before you perform top-level synthesis in the block abstraction flow, you should complete these tasks:

- Create the block abstraction.
- Specify the settings linking it to the top-level design.
- Run a report to check the top implementation settings.
- Run a report to get details about the block abstraction.
- Check the readiness of the block abstraction.
- (Optional) Ignore timing paths that are entirely in the block abstraction.

The following script shows part of a top-level synthesis flow. 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}
... #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_exploration -check_only
compile_exploration
```

See Also

- Block Abstraction Hierarchical Flow
- Ignoring Timing Paths Within Block Abstractions
- Performing a Bottom-Up Compile

Ignoring Timing Paths Within Block Abstractions

A block abstraction can contain register-to-register paths that are entirely within the block. These paths are the result of combinational logic in the block abstraction that is shared by input-to-register and register-to-output paths.

These paths are optimized during block-level optimization. To ignore these paths, set the `timing_ignore_paths_within_block_abstraction` variable to `true`, changing it from its default of `false`. For example,

```
de_shell> set_app_var timing_ignore_paths_within_block_abstraction true
```

This variable setting does not affect paths that begin within a block abstraction, traverse outside, and return to the block abstraction.

See Also

- Block Abstraction Hierarchical Flow

Limitations

Block abstractions have the following limitations:

- The IC Compiler tool does not support block abstractions that are created in DC Explorer.
- You cannot modify or optimize block abstractions in DC Explorer.

See Also

- Block Abstraction Hierarchical Flow

12 Working With the GUI

Design Vision is the graphical user interface (GUI) for the Synopsys logic synthesis environment. It provides analysis tools for viewing and analyzing your design at the generic technology (GTECH) level and the gate level. It also provides all of the synthesis capabilities of the DC Explorer tool.

The GUI provides a variety of visualization and analysis features. You can use it to visualize design data and analyze results. The GUI windows provide menu commands and dialog boxes for the most commonly used synthesis features. In addition, you can enter any de_shell command on the command line in the GUI or the shell.

Before you start using the GUI to analyze and troubleshoot a design, you should become familiar with its operation and the various tools that it provides. The following sections provide the general and specific information you need to know before using the GUI for the first time:

- Features and Benefits
- Using GUI Windows
- Performing Floorplan Exploration
- Floorplan Exploration With IC Compiler II
- Analyzing the RTL
- RTL Cross-Probing

Features and Benefits

The Design Vision GUI provides the following features:

- Window- and menu-driven interface for Synopsys logic synthesis
- Analysis visualization capabilities that include
  - A hierarchy browser for navigation through the design hierarchy and exploration of design structures, including viewing area attributes
  - Histograms for visualizing trends in various metrics, for example slack and capacitance
  - Timing analysis driver and path inspector for detailed timing path analysis
  - Path schematics for visually examining timing paths, fanin and fanout logic, and net connectivity
  - Path profiles for visually examining the contributions of different cells and nets to the total delay of a timing path
  - Design schematics and symbol views for visually examining high-level and low-level design connectivity
  - A properties viewer for viewing object properties and editing attribute values
  - Layout views for viewing and analyzing floorplan elements in the physical design
- Reporting capabilities that correlates reported objects to graphical views
- Command-line interface integrated in the GUI with scripting support for all Design Compiler tool command language (dctcl) commands
- An HTML-based document browser for viewing man pages and online Help pages

Using the features of the Design Vision GUI, you can

- Perform DC Explorer synthesis and reporting
- Explore design structures
- Visualize overall timing with Design Vision graphical analysis
- Perform timing analysis for blocks you are synthesizing
- Perform detailed visual analysis of timing paths and connected logic
- Visually examine and debug timing paths and certain floorplan constraints in the physical design layout

Using GUI Windows

The GUI provides a flexible working environment with multiple windows for performing different tasks. The GUI windows operate independently in your X windows environment, but they share the same designs in memory, the same current timing information, and the global selection data in the tool.

Design Vision provides the following GUI windows:

- The Design Vision main window appears automatically when you open the GUI from de_shell.

  The hierarchy browser (logic hierarchy view) and the command console (console panel) appear by default in the Design Vision main window.

- The layout window is available for visualizing the physical aspects of a design.

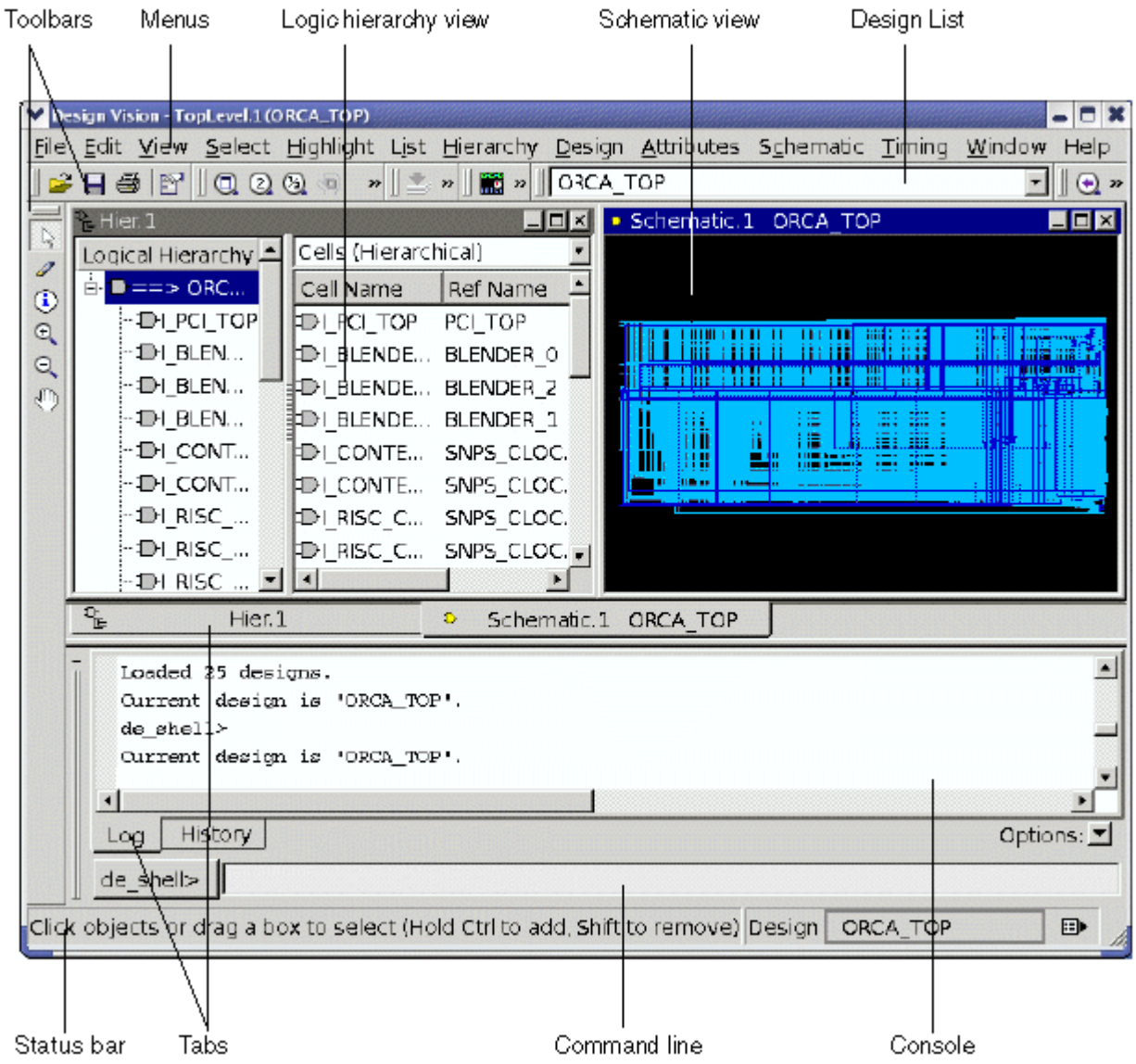
  To open a layout window, choose Window > New Layout Window.

Each GUI window consists of a title bar, a menu bar, and several toolbars at the top of the window and a status bar at the bottom of the window. The work space area between the toolbars and the status bar displays view windows and panels. View windows provide graphic or textual views of design information. Panels provide interactive tools for setting options or performing often used tasks.

You can resize, move, and minimize or maximize each GUI window by using the window management tools on your Linux desktop. For additional details, see the Design Vision Help. You can also open multiple instances of a GUI window and use them to compare views, or different design information within a view, side by side. The windows are numbered sequentially throughout the session.

Figure 12-1 shows an example of the Design Vision main window after you have read in a design and opened a design schematic for the top-level design.

Figure 12-1: Design Vision Main Window



You can display a brief message in the status bar about the action that a menu command performs by holding the pointer over the command name. To determine the function of a toolbar button, hold the pointer over the button. A ToolTip displays the name of the button, and the status bar displays a brief description of its use.

For menu commands that you can use by clicking a toolbar button or pressing a keyboard shortcut, the menus show representations of those alternatives. Some frequently used menu commands are also available on the pop-up menus for individual views. To use a pop-up menu, right-click and choose a command.

You can configure the toolbars, view windows, and panels independently for each window. Toolbars are always attached to a window edge. You can attach individual panels to window

edges or allow them to float inside or outside the workspace area. Most panels are associated with a particular type of view and operate on the active view. An exception is the console, which contains a command line and its own views.

You can enhance your working environment by adjusting the sizes of view windows and panels for viewing purposes. You can minimize individual view windows, or maximize a view window to fill the workspace area. In addition, you can arrange the open view windows by tiling or cascading them within the workspace area.

The GUI automatically saves your current window configuration when you close the GUI or exit DC Explorer, and restores it the next time you open the GUI. The window configuration includes the size and location of the main window and the layout window and the visibility and location of each toolbar and panel.

For more information about GUI windows, see the following sections:

- Entering Commands and Viewing Results
- Viewing Man Pages
- Saving an Image of a Window or View
- Getting Help in the GUI

Entering Commands and Viewing Results

When you open the GUI, the console panel is docked above the status bar by default. You can use the console to

- Enter de_shell commands on the console command line
- View either the session transcript (log view) or the command history list (history view) by clicking the tabs above the command line
- Copy and edit or reuse commands, and search for, select, and save commands or messages

You can enter any de_shell command on the console command line. When you issue a command (by pressing Return or clicking the prompt button to the left of the command line), the console and de_shell both echo the command output, including processing messages and any warnings or error messages. For example, if you enter the `get_selection` command, the console log view displays a list of the names of all selected objects.

You can display, edit, and reissue commands on the console command line by using the arrow keys to move the insertion point to the left or right on the command line or to scroll up or down the command stack.

The log view displays a transcript of session information, which includes the commands you run since you opened the GUI and the output and messages resulting from these commands. You can use this information to

- Check the tool status after performing functions
- Troubleshoot problems that you encounter
- Search the transcript for information about past functions
- Save the transcript, selected text, or just the error and warning messages in a text file

You can copy text in the log view and paste it on the command line the same way you would in a Linux shell, by selecting the text with the left mouse button and pasting it with the middle button.

The history view lists menu, dialog box, and de_shell commands you have used in the current GUI session. You can do the following with this list:

- See which commands you have used
- Find and reissue commands that you have used
- Save the list for later use

You can select commands in the history view and edit or reissue them on the command line.

See Also

- The “Viewing the Session Log” and “Viewing the Command History” topics in Design Vision Help
- The “Entering Commands in the Console” topic in Design Vision Help

Viewing Man Pages

The GUI provides an HTML-based browser window that lets you view man pages for commands, variables, and error messages. You can use it to

- Display a man page
- Search for text on the man page you are viewing
- Print the man page you are viewing
- Browse back and forth between man pages you have already viewed

To open the man page viewer,

- Choose Help > Man Pages.

  The home page displays a list of links for the different man page categories: commands, variables, and messages. Click the category link for the type of man page you want to view, and then click the title link for the man page you want to view.

You can 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 or by using the `gui_show_man_page` command in de_shell when the GUI is open.

See Also

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

Saving an Image of a Window or View

You can save an image of a GUI window or a view window in an image file. The image shows the window exactly as it appears on the screen but without the window border or title bar. For example, if you save an image of the active schematic view, the image shows the visible portion of the schematic at the current zoom level and pan position.

To save an image of the active GUI window or view window,

1. Choose View > Save Screenshot As.

   The Save Screenshot As dialog box appears.

2. Specify the filename and image format.

   The image format can be PNG (the default), BMP, JPEG, or XPM. To specify a nondefault image format, use file name extension for that format.

3. (Optional) To save an image of the active view window instead of the GUI window in which you are working, select the “Grab screenshot of active view only” option.

4. Click Save.

Alternatively, you can use the `gui_write_window_image` command by specifying the file name, image format, and window name. Window instance names appear in the window title bars and on the Window menu. For example, to save a JPEG image of the active schematic view in a file named schem_1.jpg, enter

```
de_shell> gui_write_window_image -file schem_1.jpg
```

You cannot save images of dialog boxes or other GUI elements such as toolbars or panels.

See Also

- The “Saving an Image of a Window” topic in Design Vision Help

Getting Help in the GUI

Design Vision Help is available in the DC Explorer GUI. You can find most of what you need to know about the GUI in Design Vision Help for DC Explorer. You can access Design Vision Help from the Help menu in the Design Vision main window or the layout window.

To view Design Vision Help for DC Explorer,

- Choose Help > Online Help.

To view Design Vision Help tips for using interactive tools in the layout window,

- Choose Help > Layout Help.

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 provide overviews of Design Vision window components and tools.
- How-to topics provide procedures for accomplishing synthesis and analysis tasks.
- Reference topics provide explanations of views, toolbar buttons, menu commands, and dialog box options.

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

Note: Before you can access Design Vision Help from within Design Vision, the Web browser executable file must be listed in your Linux $PATH variable.

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.

Other sources of information about the GUI include the `gui_*` command man pages and the SolvNet knowledge base.

See Also

- The “Using This Help System” topic in Design Vision Help

Performing Floorplan Exploration

You can perform floorplanning tasks, such as floorplan analysis, floorplan creation, and floorplan modification, from within the synthesis environment by using the IC Compiler design planning tools in the IC Compiler layout window. The interface between floorplan exploration in DC Explorer and the IC Compiler layout window is transparent, allowing you to move seamlessly between the Design Vision and IC Compiler layout windows.

To learn how to perform floorplanning tasks, see

- Enabling Floorplan Exploration
- Enabling Data Flow Analysis
- Using the Floorplan Exploration GUI

- Saving the Floorplan or Discarding Updates
- Exiting the Session
- Performing Synthesis After Floorplan Changes
- Using Floorplan Exploration in Batch Mode
- Black Boxes, Physical Hierarchies, and Block Abstractions

Enabling Floorplan Exploration

DC Explorer allows you to perform floorplan exploration by using the IC Compiler design planning tools in the IC Compiler layout window. The interface between floorplan exploration in DC Explorer and the IC Compiler layout window is transparent. You can perform floorplan exploration during the following design stages:

- Before synthesis

  To perform floorplan exploration, ensure that the `enable_presynthesis_floorplanning` variable is set to `true` (the default).

  Note: You must have a logic library and a physical library containing at least an AND gate, an inverter, and a buffer to start design planning in IC Compiler.

- After synthesis

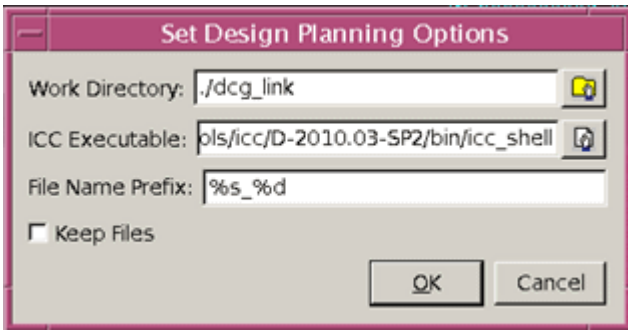
  Before performing floorplan exploration, you read the RTL files, specify the physical and logic libraries, define the Synopsys design constraints, specify a floorplan (if you have one), and synthesize the design in DC Explorer. After synthesis, you can evaluate the quality of results and use floorplan exploration to create a new floorplan or improve an existing floorplan, as needed.

To set up and start a floorplan exploration session,

1. (Optional) To invoke floorplan exploration before synthesis, set the `enable_presynthesis_floorplanning` variable to `true`.
2. In the DC Explorer GUI, choose Floorplan > Set Design Planning Options.

   The Set Design Planning Options dialog box appears, as shown in Figure 12-2.

Figure 12-2: Set Design Planning Options Dialog Box



3. 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 other necessary files are stored.

4. In the ICC Executable box, enter the name and location of the IC Compiler executable.

   By default, DC Explorer uses the icc_shell executable specified by the $path variable.

5. 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.

6. Click OK.

7. Choose Floorplan > Start Design Planning.

   A warning appears, showing the following message:

   Warning: Starting floorplan exploration will close the GUI. Do you want to continue?

8. Click OK.

   DC Explorer invokes floorplan exploration with simplified floorplanning menus in the IC Compiler layout window.

As an alternative to performing steps 1 through 8, use the following de_shell commands:

```
de_shell> set_icc_dp_options -work_dir "./dcg_link" \
-icc_executable "*/bin/icc_shell" \
-file_name_prefix "%s_%d" -keep_files
de_shell> start_icc_dp
```

See Also

- The “Using Floorplan Exploration Tools” topic in Design Vision Help
- Using Floorplan Exploration in Batch Mode

- IC Compiler Help

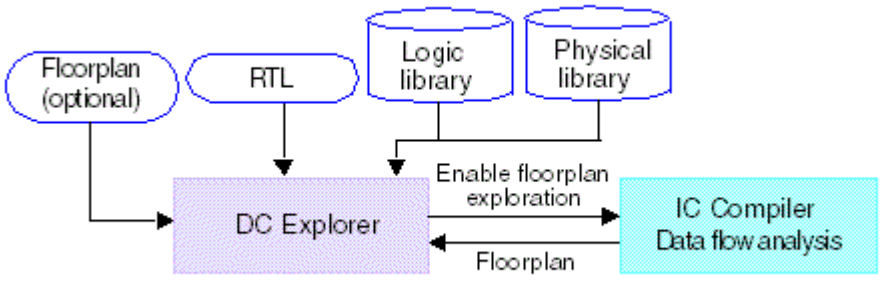
Enabling Data Flow Analysis

Floorplan exploration allows you to analyze and improve the data flow through the physical design using the data flow analysis tools in IC Compiler design planning. You can use the data flow analyzer window to perform the following data flow analysis:

- Logic connectivity analysis to view macro connections from the perspective of the logic data flow
- Advanced flyline analysis to identify the data flow by examining connectivity between macros and between macros and I/O cells
- Interactive macro editing to modify macro placement, create macro groups, and create macro arrays

Figure 12-3 shows an overview of early floorplanning from DC Explorer to design planning in IC Compiler.

Figure 12-3: Early Floorplanning Using Data Flow Analysis



To start a data flow analysis session,

1. Invoke the IC Compiler layout window by following the steps described in Enabling Floorplan Exploration.

   The IC Compiler layout window appears.

2. Choose Floorplan > Analyze Logic Connectivity.

   The data flow analyzer window appears.

3. Use IC Compiler floorplanning tools to perform floorplanning tasks, including creating, modifying, and analyzing a floorplan.

4. Save the floorplanning updates to DC Explorer by choosing Floorplan > Update DC Floorplan.

5. (Optional) Perform optimization by following the steps described in Performing Optimization With Floorplan Physical Constraints.

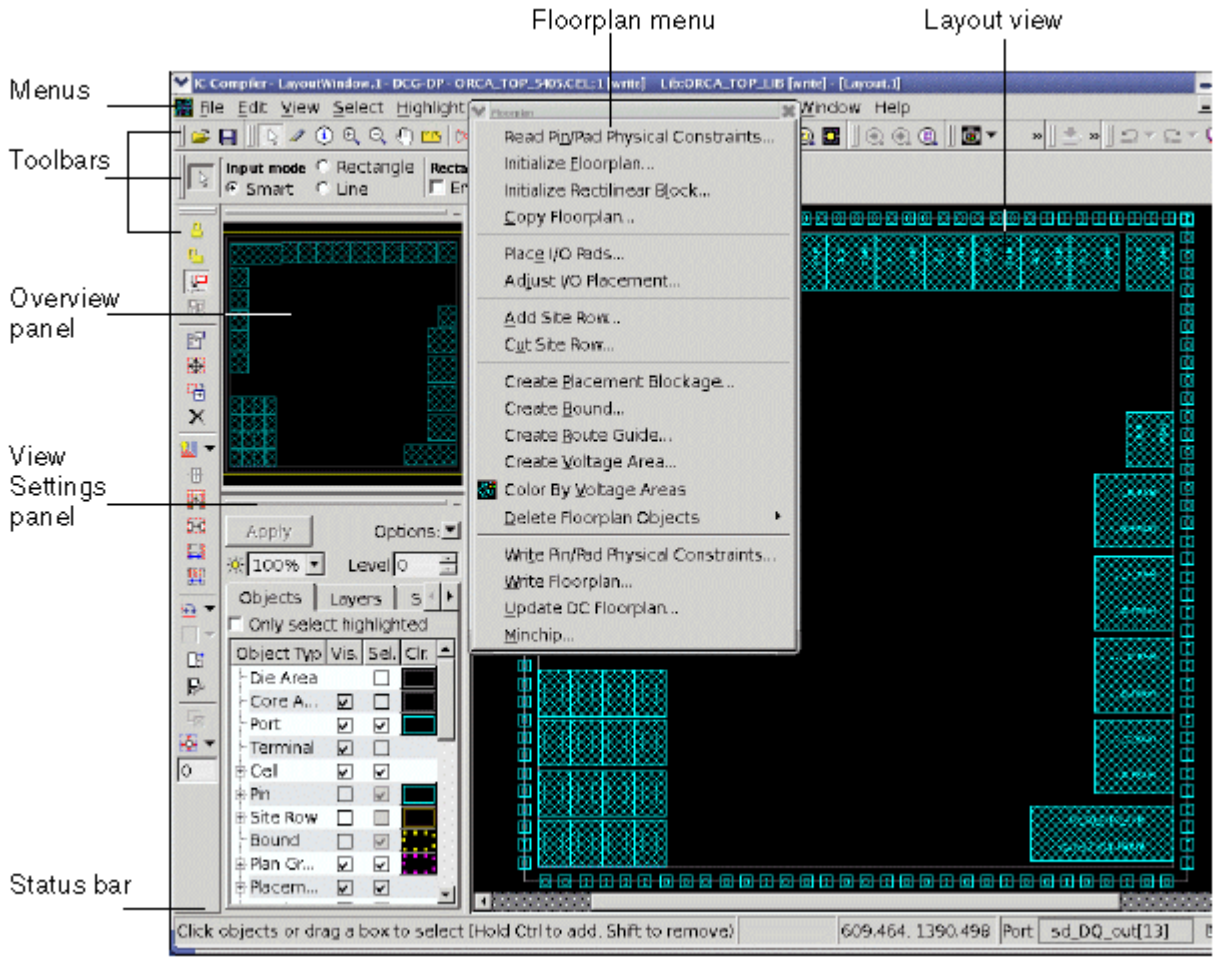
See Also

- Enabling Floorplan Exploration
- Performing Optimization With Floorplan Physical Constraints
- The “Analyzing the Data Flow for Macro Placement” topic in IC Compiler Help
- SolvNet article 030613, "Design Compiler Graphical Floorplan Exploration Application Note"
- *IC Compiler Implementation User Guide*

Using the Floorplan Exploration GUI

Floorplan exploration in DC Explorer uses the IC Compiler layout window. By default, when you enable floorplan exploration, DC Explorer configures the IC Compiler layout window so that it includes only the menus and menu commands you need to perform DC Explorer floorplanning. Figure 12-4 shows the simplified floorplanning menu structure.

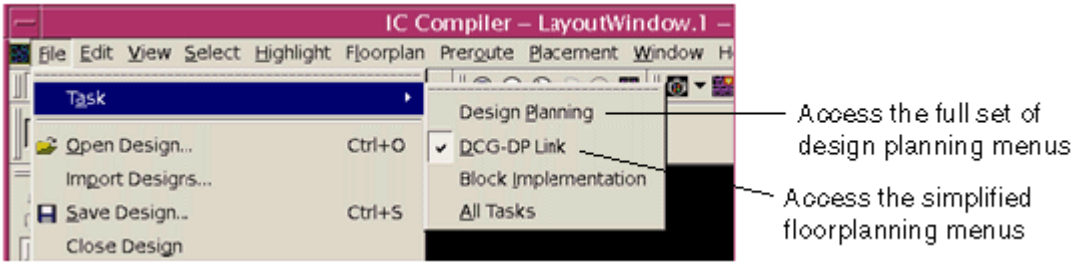
Figure 12-4: IC Compiler Simplified Layout Window Features



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 12-5. To switch from the full design planning menu structure back to the simplified menu structure, choose File > Task > DCG-DP Link.

Figure 12-5: 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.

Creating and Editing Floorplans

Whether you use the simplified DC Explorer 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, align, distribute, spread, split, reshape, or remove objects.
- Use editing commands to rotate, align, distribute, spread, 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.

See Also

- IC Compiler Help
- *IC Compiler Design Planning User Guide*

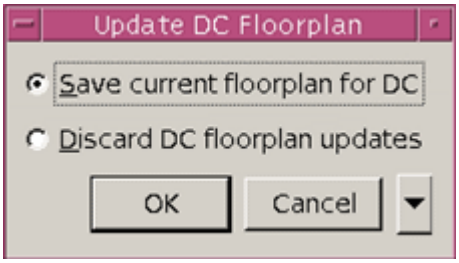
Saving the Floorplan or Discarding Updates

You can save or discard floorplan updates in DC Explorer from the IC Compiler floorplanning session at any time. 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 12-6.

   Figure 12-6: Update DC Floorplan Dialog Box

   The “Update DC Floorplan” command is available only in the simplified floorplanning menu structure for DC Explorer. 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. Select the appropriate option.
3. Click OK.

You can also save the floorplan in the following files:

- A Tcl script file

  Choose Floorplan > Write Floorplan. Alternatively, you can use the `write_floorplan` command at the tool prompt.

- A DEF file

  Choose File > Export > Write DEF. Alternatively, you can use the `write_def` command at the tool prompt.

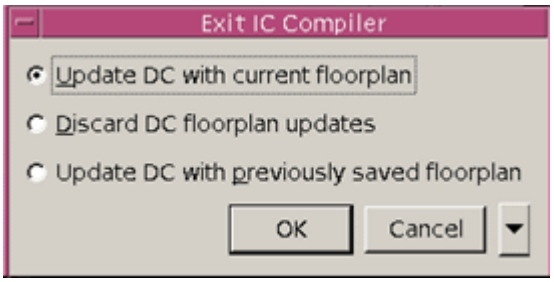
Exiting the Session

To exit the floorplan exploration session and save your floorplan for synthesis or discard the floorplan, perform the following steps:

1. Choose File > Exit.

   The Exit IC Compiler dialog box appears, as shown in Figure 12-7.

   Figure 12-7: Exit IC Compiler Dialog Box



2. Select the appropriate option.
3. Click OK.

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.

Performing Synthesis After Floorplan Changes

To prepare for full synthesis after floorplan changes,

1. You must save the floorplan into a Tcl script file by running the `write_floorplan` command. For example,

   ```
   de_shell> write_floorplan -all Design.fp
   ```

   The `-all` option is recommended for resynthesizing the design with the new or modified floorplan.

2. Read the floorplan in DC Explorer for synthesis by using the `read_floorplan` command.

3. After you read the RTL file, the SDC file, and the new floorplan file, perform full synthesis by using the `compile_exploration` command.

See Also

- Using Floorplan Physical Constraints

Using Floorplan Exploration in Batch Mode

You can also use floorplan exploration in batch mode. Batch mode allows you to create and modify floorplans on the command line using scripts.

Use the following commands in DC Explorer to perform floorplan exploration in batch mode:

1. Use the `set_icc_dp_options` command to specify the IC Compiler executable and other setup requirements. For example,

   ```
   de_shell> set_icc_dp_options -icc_executable icc_image
   ```

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 with the `set_icc_dp_options` command, the `report_icc_dp_options` command reports the default settings.

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 in batch mode, use the `-f` option with the `start_icc_dp` command. For example,

   ```
   de_shell> start_icc_dp -f ../../scripts/script_file_name.tcl
   ```

See Also

- Enabling Floorplan Exploration

Black Boxes, Physical Hierarchies, and Block Abstractions

DC Explorer supports floorplan exploration with netlists that contain black boxes, physical hierarchies, and block abstractions.

- Black Boxes

  If a design has black boxes, you do not need to perform any additional steps or modify the floorplan exploration flow. The black boxes that are defined or created in DC Explorer are transferred to IC Compiler design planning automatically. Using the IC Compiler tool, you can edit the floorplan with black boxes during the floorplanning session and transfer the modifications back to DC Explorer.

- Physical Hierarchies and Block Abstractions

  DC Explorer transfers physical hierarchies and block abstractions to IC Compiler design planning automatically during floorplan exploration. Physical hierarchies and block abstractions are transferred as macros with quick timing models in the blocks.

  When transferring physical hierarchies and block abstractions to the IC Compiler tool for design planning, DC Explorer automatically sets the `dct_physical_hier` attribute to `true` on the instances.

Querying Hierarchical Blocks and Pin Connections

To return all the DC Explorer hierarchical blocks, including physical hierarchies and block abstractions, use the `get_cells` command with the `dct_physical_hier` attribute. The DC Explorer physical hierarchies and block abstractions are preserved when they are transferred back to DC Explorer.

For example,

```
icc_shell> get_cells -hierarchical "dct_physical_hier==true"
{GPRs Multiplier}
```

To query the nets connected to the pins of a hierarchical block, use the `-boundary_type lower` or `-boundary_type upper` option with the `get_nets` command for the net inside or outside the hierarchical block, respectively. To query both nets, specify the `both` keyword. If you do not specify the `-boundary_type` option, the command returns the net outside the hierarchical block. When using this option, you must specify a pin by using the `-of_objects` option.

For example, the following two commands return the topnet net connected to the in pin outside the u2 hierarchical block:

```
de_shell> get_nets -of_objects u2/in
{topnet}
de_shell> get_nets -of_objects u2/in -boundary_type upper
{topnet}
```

The following command returns the u2/in net connected to the in pin inside the u2 hierarchical block:

```
de_shell> get_nets -of_objects u2/in -boundary_type lower
{u2/in}
```

Floorplan Exploration With IC Compiler II

DC Explorer allows you to perform floorplan exploration using the IC Compiler II design planning tools in the IC Compiler II layout window. After synthesis, you can evaluate the quality of results and use floorplan exploration to create a floorplan or improve an existing floorplan in IC Compiler II. The interface between floorplan exploration in DC Explorer and IC Compiler II layout window is transparent.

These topics describe how to perform floorplan exploration with IC Compiler II:

- Prerequisites for Floorplan Exploration
- Running the Floorplan Exploration Flow
- Data Transfer to IC Compiler II
- Saving in ASCII Format for IC Compiler II

See Also

- Performing Floorplan Exploration

Prerequisites for Floorplan Exploration

Before performing floorplan exploration in IC Compiler II, you must complete the following tasks:

- Synthesize the design

  You cannot run IC Compiler II on an unsynthesized (or unmapped) design.
- Generate IC Compiler II reference libraries

Because the reference libraries are not automatically converted by DC Explorer, you must generate the libraries using IC Compiler II Library Manager.

- Generate RTL UPF files for multivoltage designs

  To handle the power intent of the design, run the RTL UPF exploration flow to generate RTL UPF files.

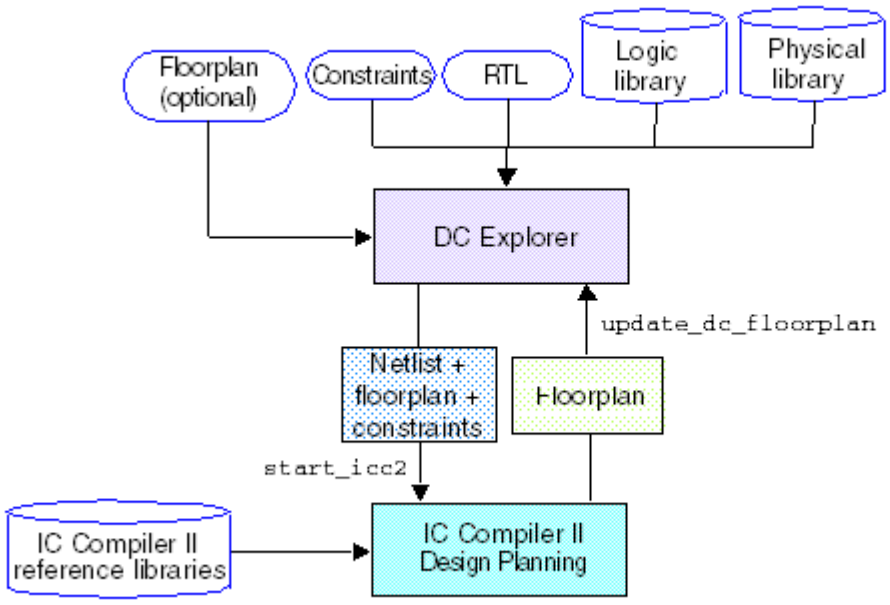
See Also

- UPF Exploration

Running the Floorplan Exploration Flow

The following figure shows an overview of floorplan exploration from DC Explorer to design planning in IC Compiler II:

Figure 12-8: ***Flow From DC Explorer to Floorplan Exploration in IC Compiler II***



The following sections describe the prerequisites and floorplan exploration flow either in interactive mode or batch mode using a de_shell script:

Prerequisites

Before floorplan exploration, you must complete the following tasks:

1. Synthesize the design by using the `compile_exploration` command.
2. Specify IC Compiler II reference libraries by using the `set_icc2_options` command.

For example,

```
de_shell> set_icc2_options -icc2_executable ../icc2_shell \
 -ref_libs "lib1.ndm lib2.ndm"
```

In Interactive Mode

Running the floorplan exploration flow using the `start_icc2` command requires an IC Compiler II license.

To start the floorplan exploration flow,

1. In DC Explorer, start an IC Compiler II design planning session by using the `start_icc2` command.

   The `start_icc2` command invokes the `icc2_shell>` prompt and opens the IC Compiler II layout window.

2. In IC Compiler II, perform floorplan exploration.

   You can create a floorplan or edit an existing floorplan using the design planning tools in the IC Compiler II layout window.

3. Generate the updated floorplan by using the `update_dc_floorplan` command in icc2_shell.

   The floorplan data in IC Compiler II are transferred to DC Explorer, and the placement is reset. If you do not perform this step, the floorplan changes done in icc2_shell are not updated to de_shell and are lost.

4. (Optional) To preserve the floorplan files in step 3, specify the `-keep_files` option with the `set_icc2_options` command.

5. End the design planning session by entering the `exit` command in icc2_shell.

   Return to the `de_shell>` prompt.

Note: When you run the `start_icc2` command, the files needed to invoke IC Compiler II are written to the $TMPDIR/dcg_unique_string working directory if you specified the TMPDIR Linux environment variable. To change the default directory, specify the `-work_dir` option with the `set_icc2_options` command.

In Batch Mode

To run floorplan exploration from a batch script, specify the `-f script_file` option with the `start_icc2` command when you start DC Explorer. In batch mode, the `start_icc2` command invokes icc2_shell and executes the specified script, but it does not open the GUI.

The following script shows an example of the general floorplan exploration flow with IC Compiler II:

```
#Perform synthesis using compile_exploration
compile_exploration -scan -gate_clock

#Set the IC Compiler II options
set_icc2_options -icc2_executable ../icc2_shell \
-ref_libs "lib1.ndm lib2.ndm"

#Launch IC Compiler II session in batch mode
start_icc2 -f my_icc2.tcl
```

Example batch script my_icc2.tcl:

```
#This example script displaces the macro I_RAM by 10 microns
set_placement_status placed [get_cells I_RAM]
move_objects -delta {10 10} [get_cells I_RAM]
set_placement_status fixed [get_cells I_RAM]

#Update the floorplan changes before exit
update_dc_floorplan
exit
```

Data Transfer to IC Compiler II

When you launch IC Compiler II within DC Explorer, design data is automatically transferred to IC Compiler II in the IC Compiler II compatible format. The following table shows which design and environment settings are transferred from DC Explorer to IC Compiler II.

Table 12-1: ***Design and Environment Settings Transferred to IC Compiler II***

Transfer	Design and environment settings

Transferred	• Verilog netlist
	• Timing constraints in IC Compiler II compatible format
	• Floorplan constraints in a DEF file and a Tcl floorplan file in IC Compiler II compatible format
	• Attributes, such as
	`dont_touch`
	`size_only`
	`dont_use`
	• Technology file
	• TLUPlus file
	• UPF file
	• Layer-related constraints, such as net routing layer constraints and preferred routing directions
	• Ignored layer settings

Not transferred	• Hierarchical models DC Explorer and IC Compiler block abstractions and physical hierarchies are treated as unresolved references in icc2_shell. • Estimations of physical black boxes These physical cells are created by the `estimate_fp_black_boxes` command. • Relative placement constraints These constraints are created in DC Explorer.

Analyzing the RTL

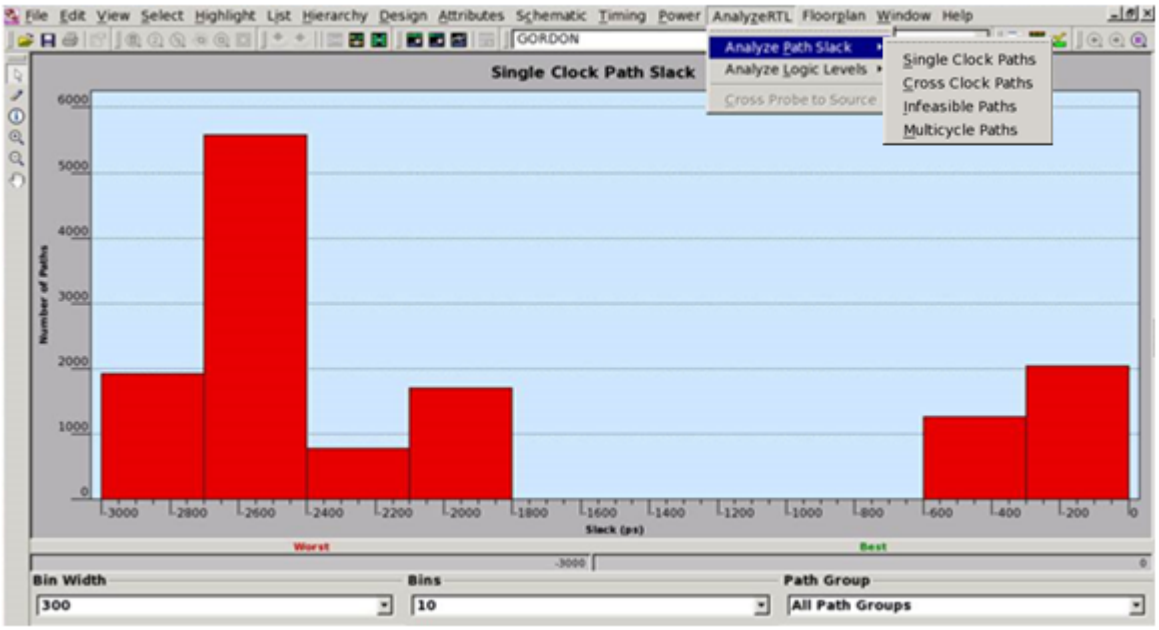
You can debug timing and congestion problems during early design stages by analyzing the overall timing and logic profiles of the design. The DC Explorer GUI enables you to generate such profiles to identify the worst-timing paths, paths with the most logic levels, infeasible paths, multicycle paths, and so on. To analyze the design, you generate a histogram to display the distribution of paths that fall in each slack range or logic-level range, called a bin. When a bin is selected, you can load the paths into a path data table and then cross-probe its cells in the RTL browser.

Note: To use this analysis capability, you must have a mapped or compiled design.

In the DC Explorer GUI, you can perform the following RTL analysis:

- To generate a path slack histogram,
    1. Choose AnalyzeRTL > Analyze Path Slack.
    2. Choose Single Clock Paths, Cross Clock Paths, Infeasible Paths, or Multicycle Paths.

Figure 12-9: Path Slack Histogram



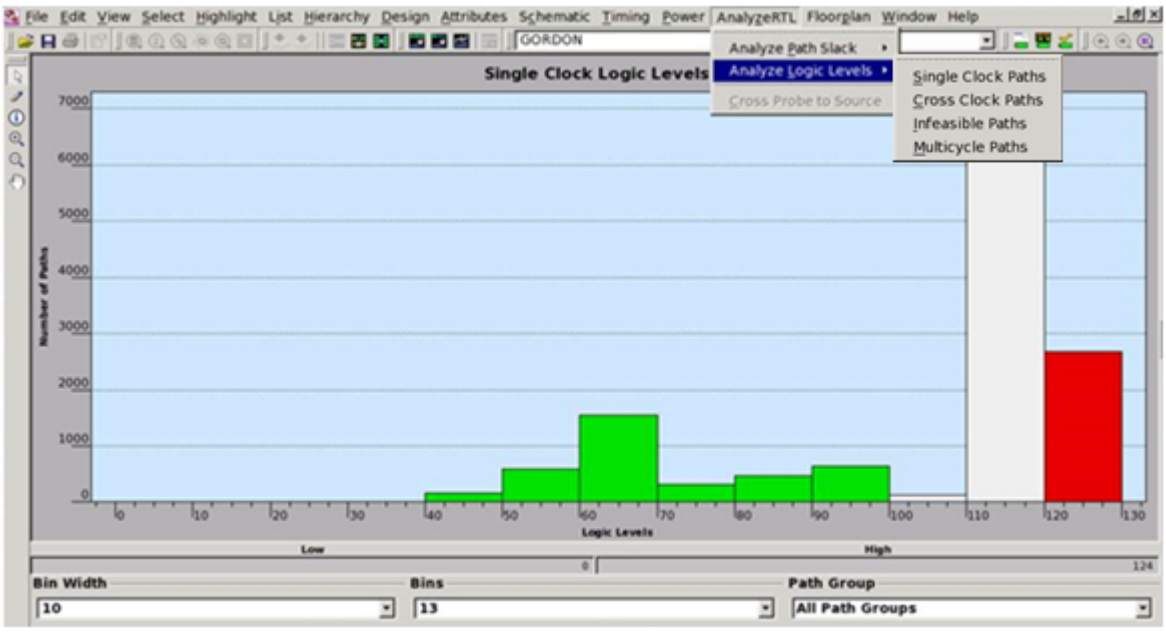
You can create a histogram of same-clock paths, cross-clock paths, infeasible paths, or multicycle paths that violate timing constraints. By default, a path slack histogram displays the distribution of slack values for all paths in 10 bins with a minimum range of 100 ps in each bin. The bins represent the number of paths (y-axis) versus their slack values (x-axis).

When a bin is selected, you can select the paths in the bin, display the paths in a new path slack or logic-level histogram, or load the paths into a path data table.

- To generate a logic-level histogram,

  1. Choose AnalyzeRTL > Analyze Logic Levels.
  2. Choose Single Clock Paths, Cross Clock Paths, Infeasible Paths, or Multicycle Paths.

Figure 12-10: Logic-Level Histogram



You can create a histogram of same-clock paths, cross-clock paths, infeasible paths, or multicycle paths that violate timing constraints. By default, a logic-level histogram displays the distribution of logic levels for all paths in 8 to 16 bins. The bins represent the number of paths (y-axis) versus the number of logic levels (x-axis).

When a bin is selected, you can select the paths in the bin, display the paths in a new path slack or logic-level histogram, or load the paths into a path data table.

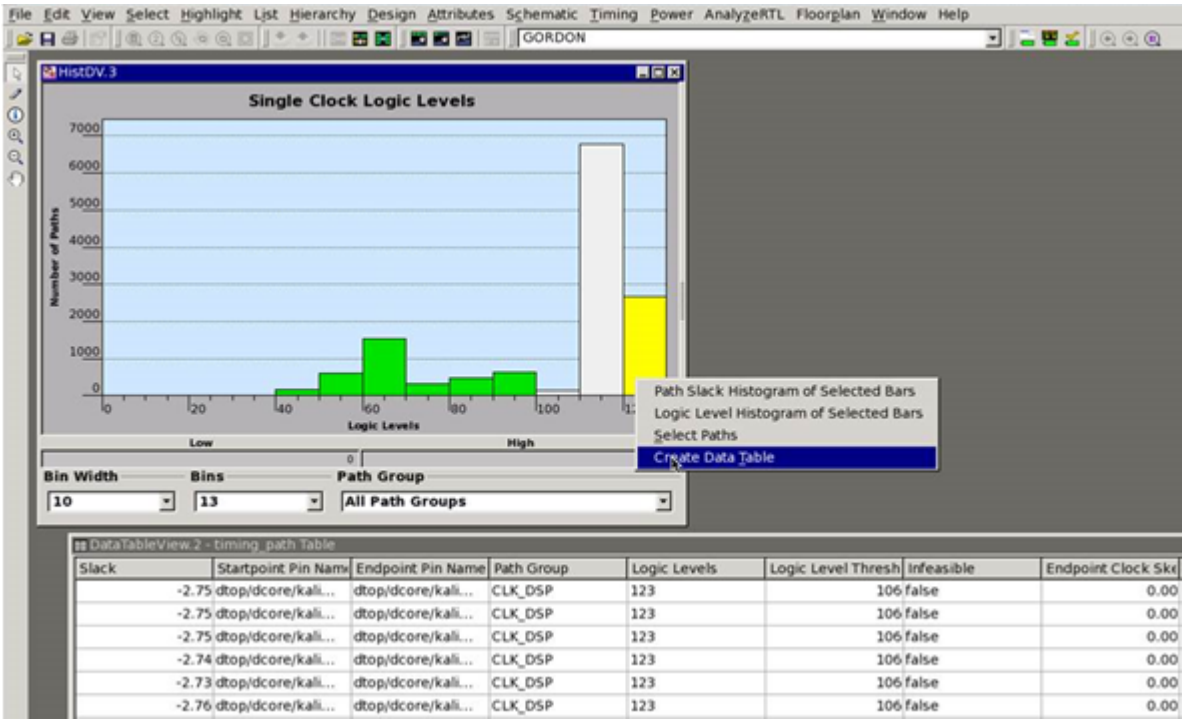
- To generate a path data table,

  1. Select one or more bins in a path slack or logic-level histogram.

     The color of the selected bins changes to yellow.

  2. Right-click and choose Create Data Table.

Figure 12-11: Logic-Level Histogram and Path Data Table



- To cross-probe cells,

  1. Select one or more cells, pins, or ports.

     You can select objects in a schematic or layout view, the hierarchy browser, or an object list, or by choosing commands on the Select menu.

  2. Use either one of the following ways to cross-probe:

     - Choose AnalyzeRTL > Cross Probe to Source.
     - Right-click the selected objects and choose Cross Probe to Source.

     The tool locates the RTL files in which the cells originate and opens the files in the RTL browser.

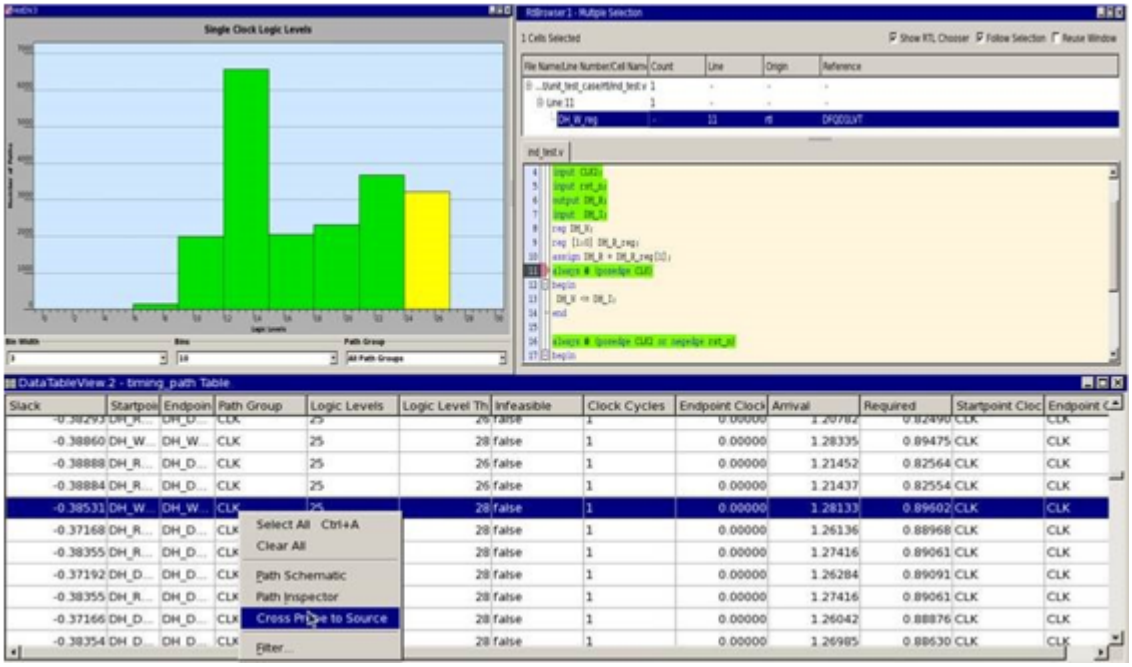
- To cross-probe timing paths,

  1. Select one or more paths in the path data table.
  2. Use either one of the following ways to cross-probe:

     - Choose AnalyzeRTL > Cross Probe to Source.
     - Right-click the selected path and choose Cross Probe to Source.

The tool creates a collection of all the cells on each path, locates the RTL files in which the cells originate, and opens the files in a new RTL browser window.

Figure 12-12: Cross-Probing Between Path Data Table and RTL Browser



You can cross-probe between each of the following design views and the RTL browser:

- Schematic view
- Path slack or logic-level histogram (via a path data table)
- Path inspector
- Congestion map

See Also

- The “Analyzing RTL” topic in Design Vision Help
- RTL Cross-Probing

RTL Cross-Probing

DC Explorer offers cross-probing capabilities in the GUI, so you can review, verify, and check the correspondence of your RTL design in the RTL browser. You can cross-probe cells, pins, or timing paths in a design from one view, such as a schematic, cell list, layout, or timing view, and examine the corresponding RTL file. This allows you to identify lines in the code that can be modified to improve timing, congestion, or datapath extraction.

Note: To use the cross-probing capabilities, you must have an elaborated or synthesized design.

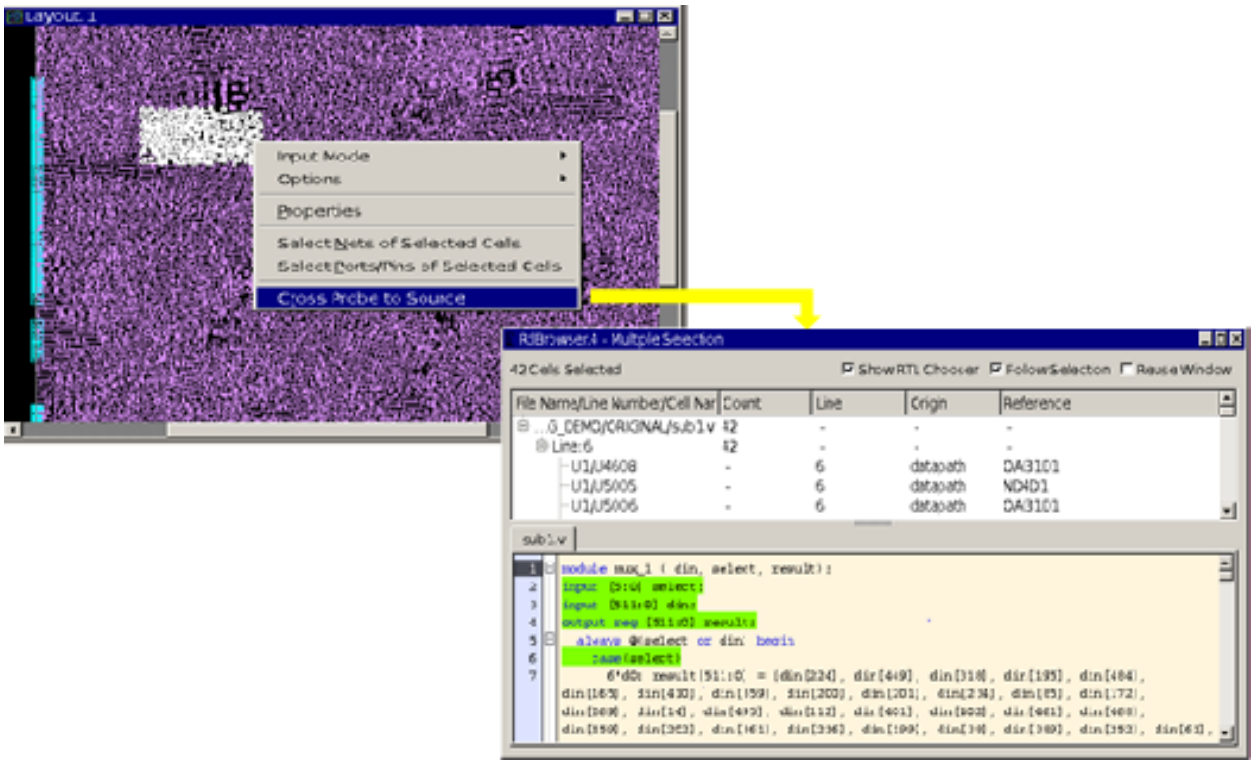
You can select a critical path from the path inspector, and the tool cross-probes the RTL to help you debug the worst-case timing paths. Cross-probing between the congestion map and the RTL browser helps you identify the RTL code that causes congestion. You can also cross-probe between a path data table and the RTL browser. The path data table shows the detailed information of the timing paths you are examining in a path slack or logic-level histogram.

The tool does not support the following cross-probing capabilities:

- From various design views to the UPF file
- Selecting nets as cross-probing objects

The following figure shows cross-probing between the Design Vision layout view and the RTL:

Figure 12-13: ***Cross-Probing Between the Layout View and the RTL Browser***



Repeating RTL Cross-Probing

Each time the tool reads an RTL for cross-probing, it stores the RTL line numbers and file paths in the database. If the RTL was moved to a different location after the cross-probing, repeating the same cross-probing sequence no longer reflects the previous line numbers. As a result, the tool cannot locate the file and issues an error message.

To repeat a cross-probing sequence or reference previous cross-probing data, you must first find the status of the files that were cross-probed. You can obtain the status by running the following commands:

- `report_cross_probing_files`

  The command reports the status of the cross-probed files, including OK, Missing, Modified, and No Permissions.

- `update_cross_probing_files`

  The command updates the locations of the files that were cross-probed and lists the previous and current paths of the files. The command does not update the cross-probed files if they have been modified.

See Also

- The “Analyzing RTL” topic in Design Vision Help
- The “Viewing Path Data Tables” topic in Design Vision Help
- Analyzing the RTL
- RTL Crossing-Probing Using Commands

13 Analyzing and Resolving Design Problems

Use the reports generated by DC Explorer 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 how to analyze and resolve design problems, see

- Fixing Errors Caused by Unsupported Technology File Attributes
- Comparing DC Explorer and IC Compiler Environments
- Checking for Design Consistency
- Checking for Unmapped Cells
- Analyzing Design Problems
- Analyzing Area
- Analyzing Timing
- Generating Quality of Results
- Debugging Cells and Nets With the dont_touch Attribute
- RTL Crossing-Probing Using Commands
- Reporting Logic Levels in Batch Mode

Fixing Errors Caused by Unsupported Technology File Attributes

Occasionally, IC Compiler adds support for new technology file attributes that are not yet supported in DC Explorer. In these cases, a TFCHK-009 error message is issued in DC Explorer but not in IC Compiler when you use 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 DC Explorer.

- 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.

Comparing DC Explorer and IC Compiler Environments

Sometimes DC Explorer and IC Compiler have different environment settings. These differences can lead to correlation problems. To help fix correlation issues between DC Explorer and IC Compiler, use the `consistency_checker` command in your Linux shell to compare the respective environments.

Before you 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 13-1 and Example 13-2. In DC Explorer, run the `write_environment` command before the `compile_exploration` command; in IC Compiler, run the `write_environment` command followed by the `place_opt` command.

Example 13-1: Using write_environment in DC Explorer

```
de_shell> write_environment -consistency -output DCE.env
```

Example 13-2: 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 by using the `consistency_checker` command, as shown in Example 13-3.

Example 13-3: Comparing DC Explorer and IC Compiler Environments

```
%consistency_checker -file1 DCE.env -file2 ICC.env \
-folder temp-html html_report | tee cc.log
```

The resulting HTML output report lists mismatched commands and variables. In Example 13-3, 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 Linux `tee` command, as shown in Example 13-3, to save this report to a file. In this example, 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.

Environment Files in Compressed Format

To generate an output of the tool environment in gzip format, specify the `-compress` option with the `write_environment` command. You can source the output file back to the IC Compiler tool. For consistency checking, you must decompress the output file that is generated using the `-compress` option. To do so, use the `gunzip` Linux command to decompress the output file before using the `consistency_checker` command.

The following example generates a DCE.tcl.gz compressed file and decompresses it in a Linux shell for consistency checking:

```
de_shell> write_environment -consistency -output DCE.tcl -compress
Terminals:                    145
Hard Macros:                   38
Site Rows:                    351
Bounds:                         3
Placement Blockages:           22
Route Guides:                   3
Preroutes:                     20


%gunzip DCE.tcl.gz
```

See Also

- SolvNet article 026366, "Using the Consistency Checker to Automatically Compare Environment Settings Between Design Compiler and IC Compiler"

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.

DC Explorer 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:

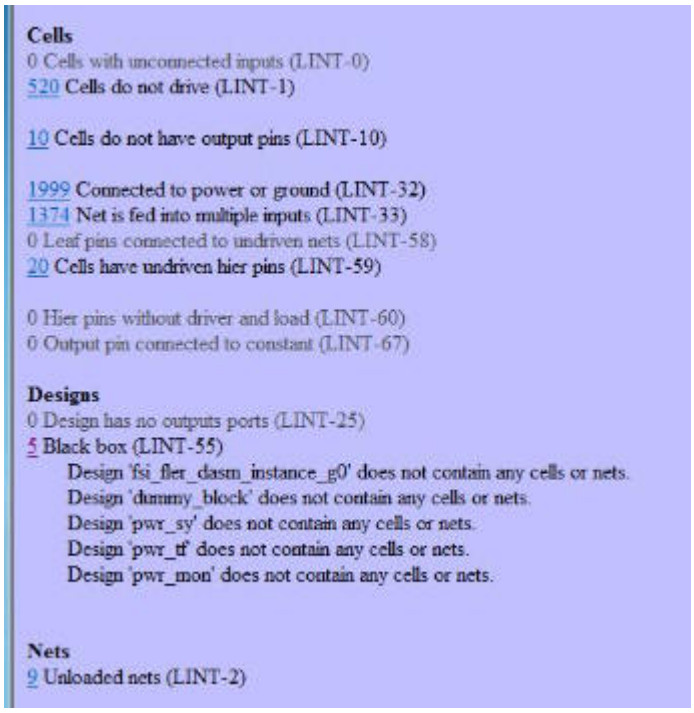
- It reports an error if it finds a problem that DC Explorer cannot resolve. For example, recursive hierarchy (when a design references itself) is an error. You cannot compile a design that has errors reported by the `check_design` command.
- It reports a warning if it finds a problem that indicates 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, DC Explorer creates an HTML file in the current directory.

This figure shows an example of a `check_design` report in HTML format. The report organizes the information into sections, such as Input/Outputs, Cells, Designs, 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 the figure shows five black box violations. Clicking the number 5 expands the report to show the five highlighted violations.

Figure 13-1: HTML check_design Report Example



By default, during compilation or execution of the `check_design` command, DC Explorer issues a warning message if a tristate bus is driven by a non-tristate driver. You can have DC Explorer 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_exploration` 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.

Checking for Unmapped Cells

You can report unmapped cells in the design by specifying the `-unmapped` option with the `check_design` command in the following ways:

- The `-unmapped` option

  When you specify this option, the `check_design` command reports the total number of unmapped cells and displays the LINT-61 warning messages with the names of unmapped cells and their designs. By default, these messages are suppressed.

For example,

```
de_shell> check_design -unmapped
*************************************
check_design summary:
Version:     H-2013.03
Date:        Tue Feb 12 12:32:24 2013
*************************************
Name                                Total
-------------------------------------------------------------------
Cells                                2
Cell is unmapped (LINT-61)           2
-------------------------------------------------------------------
Warning: In design 'unit_2', cell 'A_out_reg' is unmapped (LINT-61)
Warning: In design 'unit_0', cell 'I_2' is unmapped (LINT-61)
```

- The `-unmapped` option with the LINT-61 warning message ID number

  When you specify the `-unmapped {LINT-61}` option, as shown in the following example, the command displays a formatted report with detailed information about the unmapped cells:

```
de_shell> check_design -unmapped {LINT-61}
*************************************
Report:      LINT-61
Total Count: 2
Severity:    Warning
Message:     In design '', cell '' is unmapped
Version:     H-2013.03
Date:        Tue Feb 12 12:33:14 2013
*************************************
Design            Cell
-------------------------------------------------------------------
unit_2            A_out_reg
unit_0            I_2
-------------------------------------------------------------------
```

See Also

- Checking for Design Consistency

Analyzing Design Problems

You can use the commands listed in this table to analyze design problems.

Table 13-1: Commands to Analyze Design Objects

Object	Command	Description

Design	`report_area`
	`report_design`
	`report_design_mismatch`
	`report_hierarchy`
	`report_missing_constraints`
	`report_resources`
		Reports design size and object counts.
		Reports design characteristics.
		Reports design mismatches.
		Reports design hierarchy.
		Reports missing constraints in current design. 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.

Ports	`report_port`
	`report_bus`
	`report_transitive_fanin`
	`report_transitive_fanout`
		Displays information about ports.
		Displays information about bused ports.
		Reports fanin logic.
		Reports fanout logic.

Nets	`report_net`
	`report_bus`
	`report_transitive_fanin`
	`report_transitive_fanout`
		Reports net characteristics.
		Reports bused net characteristics.
		Reports fanin logic.
		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.

The following example shows a report generated by the `report_area` command:

Example 13-4: report_area Report Example

```
de_shell >report_area
...
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. DC Explorer 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, ensure that you have

- Defined the operating conditions
- Specified realistic constraints
- Budgeted the timing constraints appropriately
- Constrained the paths properly
- Described the clock skew

If you are running the physical flow, you can report pin locations and the capacitive loads for the pins and nets in the path report by using the `-physical` option with the `report_timing` command. The physical flow is enabled when the `de_enable_physical_flow` variable is set to `true`.

After producing the initial mapped netlist, use the `report_constraint` command to check your design's performance.

The following table lists the timing analysis commands.

Table 13-2: Timing Analysis Commands

Command	Analysis task description
`report_design`	Shows operating conditions, 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.
`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.

Generating 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 the `create_qor_snapshot` and `query_qor_snapshot` commands and the `report_qor` command. 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 into a set of snapshot files. You can later retrieve and report the snapshot with the `query_qor_snapshot` command. You can also selectively retrieve, sort, and display the desired information from the snapshot with the `query_qor_snapshot` command. The `report_qor` command displays QoR information and statistics for the current design.

To learn how to generate quality of results, see

- Measuring Quality of Results
- Analyzing Quality of Results
- Reporting Quality of Results

Measuring Quality of Results

To measure the quality of results (QoR) 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 using different optimization strategies or at different stages of the design and compare the quality of results. For example, you can use the command to create a snapshot after the first time you compile the design, after DFT insertion, and again after an incremental compilation.

You can use the command options to 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,

```
de_shell> create_qor_snapshot -name my_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 (my_snapshot1 in this example), the `query_qor_snapshot` command displays the results in text or HTML format.

Analyzing Quality of Results

The `query_qor_snapshot` command reads a QoR report previously generated by the `create_qor_snapshot` command and analyzes the results by collecting information about each path. You can query the collected information with various searches and display the results in text or HTML format. The HTML report lets you quickly find paths with certain problems such as large fanouts or transition degradation.

The command options let you filter and sort the search results and display those results in text or HTML format. 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 command does not perform any additional analysis of the design, but merely processes the report information already saved in the snapshot report files. Therefore, execution of the `query_qor_snapshot` command is much faster than execution of the `create_qor_snapshot` command.

You can 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:

```
de_shell> query_qor_snapshot -name my_snapshot -filters "-wns ,0"
```

The following example reports all paths that meet any of the specified filter requirements:

```
de_shell> 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:

```
de_shell> query_qor_snapshot -name my_snapshot \
-filters "-wns ,0 -zero_path ,0 -fanout 40 -logic_levels 50"
```

This example analyzes the snapshot named my_snapshot1 and reports the paths having a worst negative slack between –4.0 and –3.0 time units.

Example 13-5: Reporting Paths With Worst Negative Slack

```
de_shell> query_qor_snapshot -name my_snapshot1 \
-filters "-wns -4.0,-3.0"
```

This example reports the paths having a slack worse than –1.0 time units and a fanout greater than 40.

Example 13-6: Reporting Paths for Slack and Fanout

```
de_shell> query_qor_snapshot -name my_snapshot2 \
-filters "-wns ,-1.0 -fanout 40"
```

This example reads the snapshot named max_logic_level and specifies the `-filters` option to report all paths between logic levels 2 and 14.

Example 13-7: Reporting All Paths Between Specified Logic Levels

```
de_shell> 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, you must specify the `-to`, `-from`, or `-through` option. The following

example finds the violating paths that cross hard macros starting at module A and ending at module B:

Example 13-8: Reporting Timing Violations Across Hierarchical Boundaries

```
de_shell> query_qor_snapshot -name snap_shot3 -hierarchy \
-from {A/*} -to {B/*}
```

The `query_qor_snapshot` command automatically generates a report of all violating paths between the extracted timing models (ETMs) and hard macros. You can apply filters or expand the top-level paths by specifying the `-to`, `-from`, and `-through` options. The following example finds violating paths that pass through modules A, B, or C:

Example 13-9: Reporting Violating Paths That Pass Through Specified Modules

```
de_shell> query_qor_snapshot -name snap_shot4 -hierarchy \
-through {A B C}
```

This example 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 13-10: Incremental Reporting

```
de_shell> query_qor_snapshot -hierarchy -incremental \
-from {A/* B/*} -to {A/* B/*}
```

This example 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 13-11: Reporting Paths for WNS and Fanout Passing Through Specified Modules

```
de_shell> query_qor_snapshot -name my_snapshot5 \
-hierarchy -from top/A/C/* -to top/B/* \
-filters "-wns ,-3.0 -fanout 40"
```

Reporting Quality of Results

The `report_qor` command displays information about the quality of results and other statistics for the current design. It 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.

This example shows a report generated by the `report_qor` command. 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 13-12: report_qor Report Example

```
de_shell> report_qor
...
  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
  ---------------------------------------
  Overall Compile Time:              631.32
  Overall Compile Wall Clock Time:   288.11
```

Improving QoR by Controlling Path Groups Creation

You can direct the tool to automatically create path groups to improve QoR. To do so, specify the `-mode rtl` option with the `create_auto_path_groups` command after elaboration but before optimization. For example,

```
de_shell> create_auto_path_groups -mode rtl
de_shell> compile_exploration
de_shell> remove_auto_path_groups
de_shell> report_qor
```

The tool creates one path group per hierarchy. If there are multiple levels of hierarchy, the tool creates path groups for each level. To generate a QoR report with only user-specified path groups, use the `remove_auto_path_groups` command to remove the path groups created by the tool. This command removes all path groups, but it keeps the path groups that you created with the `group_path` command.

See Also

- Creating Reports and Missing Constraints

Debugging Cells and Nets With the dont_touch Attribute

If DC Explorer did not optimize or remove a cell or net, the cause is often a `dont_touch` attribute 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.

To help you debug why an object in a cell or a net is marked as don't touch, DC Explorer provides the commands described in the following sections.

Reporting Don't 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.

```
de_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:

```
de_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:

```
de_shell> list_dont_touch_types
...
************************************************
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
...
-------------------------------------------------------------------------------
1
```

Creating a Collection of Don't 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:

```
de_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:

```
de_shell> get_dont_touch_nets -type mv* block1/NET*
{block1/NET1QNX block1/NET2QNX}
```

RTL Crossing-Probing Using Commands

To improve timing, congestion, or datapath extraction, you can check the RTL design and identify lines in the code to be modified by using the RTL cross-probing capabilities on the command line.

You must have a design that was analyzed and elaborated or synthesized to use the following cross-probing commands:

- `get_cross_probing_info`

  The command returns cross-probing data for specified cells or ports as a string, specifying where the cell or port was created and which file and line number the object came from. Some objects in the design might have been merged, resulting in multiple source locations that are returned.

  The results of each object are separated by a pair of braces between each source position and between each object. For example, if you specify two objects, obj1 and obj2, where obj1 has one source position and obj2 has two source positions, the tool returns the results in the following format:

  ```
  { {obj1_file1:obj1_line1} } { {obj2_file1:obj2_line1}
  {obj2_file2:obj2_line2} }
  ```

  To prevent duplicate file:linenumber combinations in the results, use the `-unique_source` option with the `get_cross_probing_info` command. In this case, the tool returns the results in the following format:

  ```
  obj1_file1:obj1_line1 obj2_file1:obj2_line1 obj2_file2:obj2_line2
  ```

  The following example shows results from the `get_cross_probing_info` command with the `-unique_source` option:

  ```
  prompt> get_cross_probing_info [get_cells -hierarchical] \
  -unique_source
  /usr/joe/my_design/m.v:21 /usr/joe/my_design/m.v:22
  /usr/joe/my_design/m.v:28 /usr/joe/my_design/m.v:12
  /usr/joe/my_design/m.v:13 /usr/joe/my_design/m.v:14
  /usr/joe/my_design/m.v:4
  ```

- `cross_probing_filter`

  The command allows you to filter a collection by specifying a source file and line number in the following format:

  ```
  "file_pattern[:line_pattern]"
  ```

  The following example shows results from the `cross_probing_filter` command, returning cells that originated in a source file ending in m.v on line 21:

```
prompt> cross_probing_filter [get_cells -hierarchical] \
-source "*m.v:21"
{mid1/bot1/c1 mid1/bot2/c1}
```

Objects in the design that were merged, resulting in multiple source locations, are returned in the collection if any of the source positions match any of the specified source patterns.

- `report_cross_probing`

  The command reports cross-probing data for specified cells or ports in a report. If the object was generated by the tool, the file and line number are not applicable and are represented by a hyphen (-). In this case, the report specifies whether the origin of the object comes from UPF, the DFT engine, or another source. Objects that were merged are displayed in the report multiple times, once for each source location.

See Also

- RTL Cross-Probing

Reporting Logic Levels in Batch Mode

As an alternative to displaying logic-level histograms in the GUI, you can report logic levels by using the `report_logic_levels` command on the command line or in a script.

Note: You must have a mapped design to run logic-level reporting.

To integrate logic-level reporting with existing scripts, use the `report_logic_levels` command in batch mode. You can use the command to report the logic levels of the entire design or a selected path group. The command reports logic levels of all types of paths, including same-clock paths, cross-clock paths, infeasible paths, and multicycle paths. The number of logic levels reported excludes buffers and inverters.

To report logic levels,

1. (Optional) Set a specific logic-level threshold other than the default by using the `set_analyze_rtl_logic_level_threshold` command.

   By default, the tool derives the logic-level threshold based on the delay values required for the paths.

   - To set a logic-level threshold for the entire design, enter

     ```
     de_shell> set_analyze_rtl_logic_level_threshold threshold_value
     ```

   - To set a logic-level threshold for a specific path group, enter

     ```
     de_shell> set_analyze_rtl_logic_level_threshold \
      -group path_group threshold_value
     ```

The path group setting overrides the global threshold of the entire design.

- To reset all threshold values, enter

```
de_shell> set_analyze_rtl_logic_level_threshold -reset
```

- To reset a specific group setting, enter

```
de_shell> set_analyze_rtl_logic_level_threshold \
 -group path_group -reset
```

2. (Optional) Specify the fields in the summary section of the report by setting the `logic_level_report_summary_format` variable. The default is `{%group %period %wns %num_paths %max_level}`. For example,

```
de_shell> set_app_var logic_level_report_summary_format \
 {%group %num_paths %max_level %min_level %avg_level}
...
```

GROUP	NUM OF PATHS	MAX LEVEL	MIN LEVEL	AVERAGE LEVEL
All Path Groups	60862	26	2	16.39
CLK	36153	26	2	19.40
CLK_2	24709	20	3	12.00

3. (Optional) Specify the fields in the path group section of the report by setting the `logic_level_report_group_format` variable. The default is `{%slack %ll %llthreshold %startpoint %endpoint}`. For example,

```
de_shell> set_app_var logic_level_report_group_format \
 {%slack %ll %startclk %endclk %infeas %startpoint %endpoint}
...
```

Logic level report for top violating paths

WNS	LOGIC LEVELS	START CLOCK	END CLOCK	INFEASIBLE	START POINT	END POINT
-0.113	33	CLK	CLK	NO	reg[4]/CK	reg[27]/D
-0.123	35	CLK	CLK	YES	reg[0]/CK	reg[7]/D
-0.123	30	CLK	CLK	YES	reg[1]/CK	reg[8]/D
-0.013	28	CLK	CLK	NO	reg[2]/CK	reg[9]/D
-0.013	28	CLK	CLK	NO	reg[3]/CK	reg[0]/D

4. Report logic levels by using the `report_logic_levels` command. The following example generates a logic-level report of the clk_i path group:

```
de_shell> report_logic_levels -group clk_i
```

A logic-level report contains three sections: summary, logic-level distribution, and logic-level path report, as shown in the following example:

```
                          REQUIRED                NUM OF      MAX
      GROUP               PERIOD       WNS        PATHS      LEVEL
--------------------  ----------  ---------  ---------  --------
 All Path Groups             n/a        n/a      60862        26
 CLK                      0.9600    -0.3903      36153        26
 CLK_2                    0.9600    -0.2414      24709        20

Logic Level report for 'All Path Groups'
----------------------------------------

 Logic level distribution

    LOGIC         NUMBER      PERCENTAGE
    LEVELS      OF PATHS      OF PATHS
 -----------  ----------  ------------
 0 to 2              6          0%
 3 to 5            390          1%
 6 to 8           2841          5%
 9 to 11          8445         14%
 12 to 14        14351         24%
 15 to 17         7739         13%
 18 to 20        11979         20%
 21 to 23         9051         15%
 24 to 26         6060         10%

 Logic level report for top violating paths

              LOGIC
    WNS       LEVELS    THRESHOLD    START POINT     END POINT
 ----------  --------  -----------  -------------  -----------
  -0.3903        21          26      reg_1_/CK      latch/E
  -0.3902        21          26      reg_2_/CK      latch/E
  -0.3902        21          26      reg_3_/CK      reg_17_/D
  -0.3900        25          26      reg_4_/CK      reg_21_/D
  -0.3899        25          26      reg_5_/CK      reg_32_/D
```

See Also

- Analyzing the RTL

14 Using a Milkyway Database

You can write a Milkyway database within DC Explorer to use with other Synopsys Galaxy platform tools, such as IC Compiler. You do this by using the `write_milkyway` command. You can use a single Milkyway library across the entire Galaxy flow.

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 use a Milkyway database within DC Explorer, you must prepare a design library and a reference library and know about the following concepts and tasks:

- About the Milkyway Database
- Guidelines for Using the Milkyway Databases
- Creating a Milkyway Design Library
- 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 using a Milkyway database, you need to have the following required license and files:

- The Milkyway-Interface license

  DC Explorer provides this license, which is used to run the `write_milkyway` command.

- Source for logic libraries (.lib)
- Compiled databases
  - Logic libraries (.db), which contain 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 deleting unused versions of your design.

For information about using the Milkyway Environment tool, see the Milkyway documentation.

See Also

- Guidelines for Using the Milkyway Databases
- Limitations of Using Milkyway Format
- Writing the Milkyway Database

Guidelines for Using the Milkyway Databases

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_}]
  ```

- DC Explorer 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.

Creating a Milkyway Design 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.

To create a Milkyway design library,

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

   For example,

   ```
   de_shell> 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,

   ```
   de_shell> open_mw_lib $mw_design_library_name
   ```

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

   For example,

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

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

   For example,

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

See Also

- About the Milkyway Database

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 floorplan information from memory to 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.

```
de_shell> 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

```
de_shell> set_app_var mw_design_library testmw
de_shell> write_milkyway -output TOP
```

Example 14-1 shows a script to set up and write a Milkyway database.

Example 14-1: 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
```

See Also

- About the write_milkyway Command
- Guidelines for Using the Milkyway Databases
- Limitations of Using Milkyway Format
- Using Hierarchical Models

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 that is translated by using the `write_milkyway` command retains its hierarchy in the Milkyway database.

See Also

- Writing the Milkyway Database

Limitations of Using 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.

See Also

- About the Milkyway Database
- Writing the Milkyway Database

A Design File Management 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. Note that different design files can contain subdesigns with identical names.

This chapter contains the following sections:

Managing the Design Data

Use systematic organizational methods to manage the design data. Two basic elements of managing design data are

- Design Data Control
- Design Data Organization

Design Data Control

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

Table A-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

Design Data Organization

Establishing and adhering to a method of organizing data are more important than the method you choose. After you place the essential design data under a consistent set of controls, you can create a meaningful data organization. To simplify data exchanges and data searches, designers 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 A-1 shows directory structures based on the top-down compile strategy, and Figure A-2 shows the bottom-up compile strategy.

Figure A-1: Top-Down Compile Directory Structure

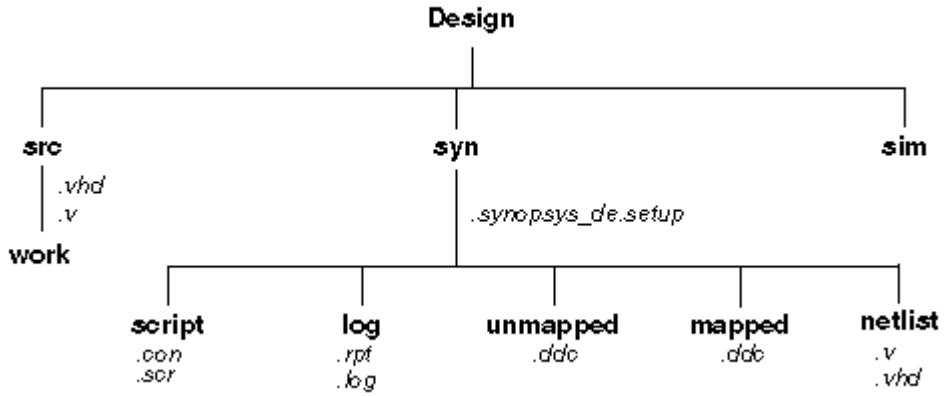
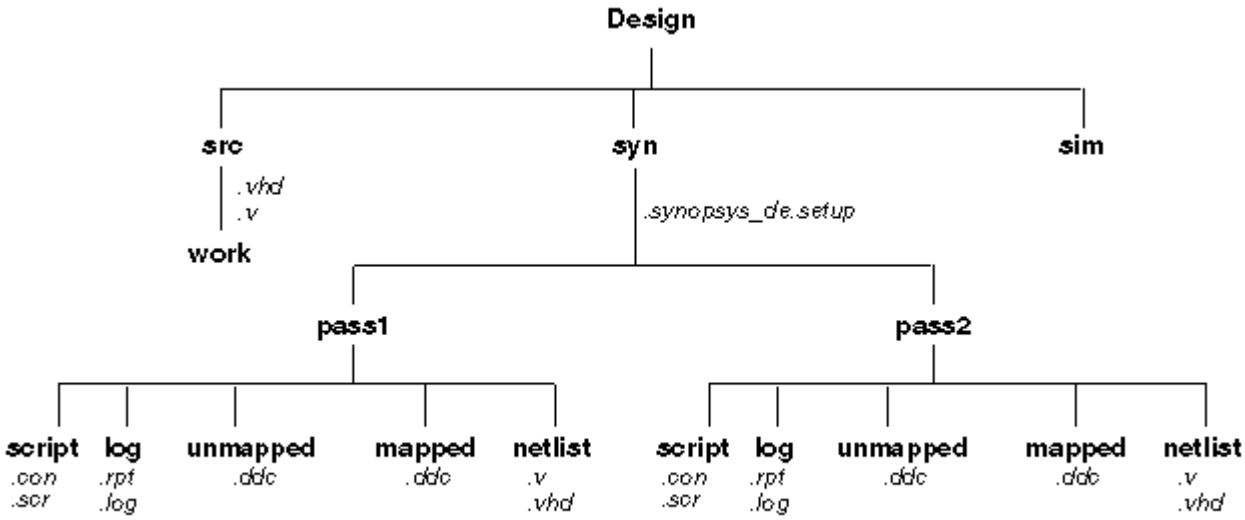


Figure A-2: Bottom-Up Compile Directory Structure



See Also

- Compile Strategies

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 DC Explorer 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:

- Partition for design reuse.
- Keep related combinational logic together.
- Register the block outputs.
- Partition by design goal.
- Partition by compile technique.
- Keep sharable resources together.
- Keep user-defined resources with the logic they drive.
- Isolate special functions, such as pads, clocks, boundary scans, and asynchronous logic.

The following sections describe each of these strategies.

Partitioning for Design Reuse

Design reuse decreases time-to-market by reducing the design, integration, and testing efforts.

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, DC Explorer 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, DC Explorer 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 DC Explorer with additional flexibility. Eliminating glue logic also reduces compile time because DC Explorer 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 A-3 shows poor partitioning of this design. Each of the combinational clouds occurs in a separate block, so DC Explorer cannot fully exploit its combinational optimization techniques.

Figure A-3: Poor Partitioning of Related Logic

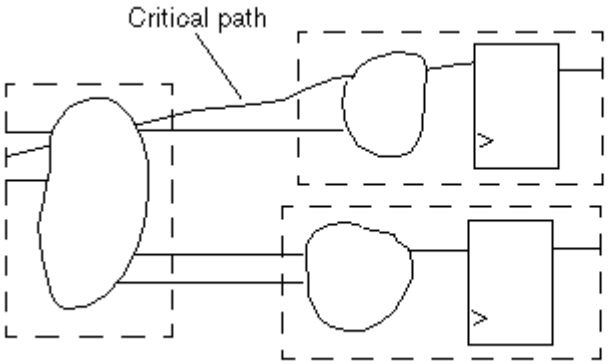
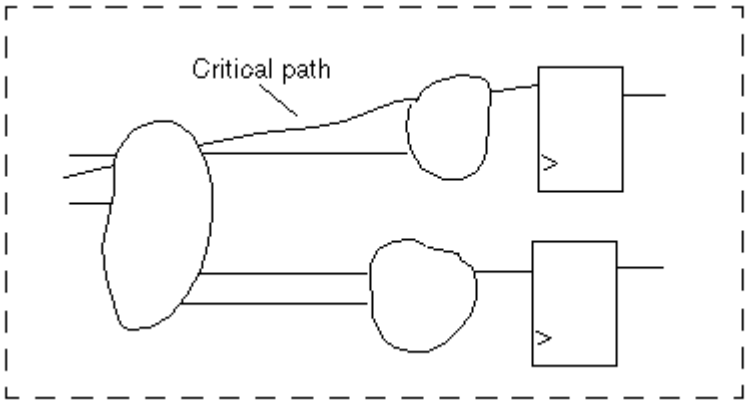


Figure A-4 shows the same design with no artificial boundaries. In this design, DC Explorer has the flexibility to combine related functions in the combinational clouds.

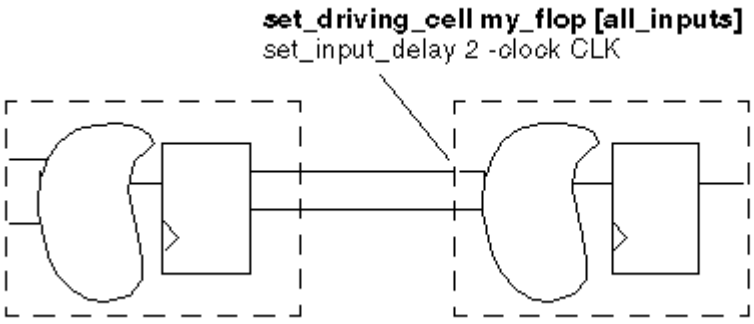
Figure A-4: Keeping Related Logic in the Same Block



Registering Block Outputs

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

Figure A-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 once in each clock cycle.

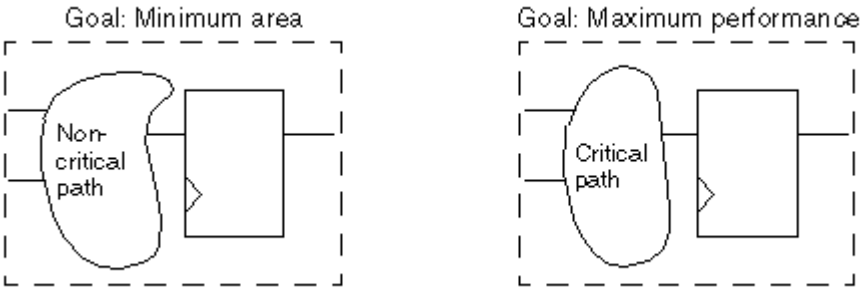
Partitioning by Design Goal

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

Figure A-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.

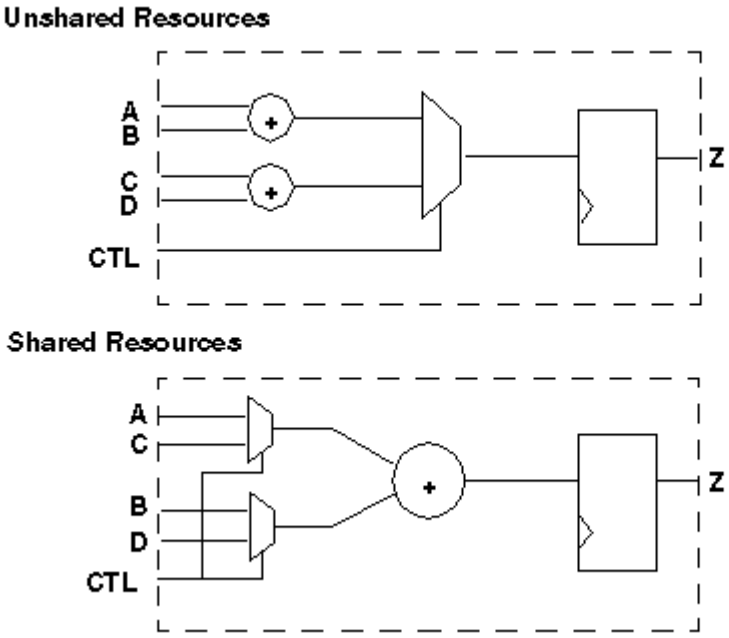
- Highly structured logic, such as error detection circuitry, which usually contains large exclusive OR trees, is better suited to structuring.
- Random logic is better suited to flattening.

Keeping Sharable Resources Together

DC Explorer 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 DC Explorer to share resources (using one adder instead of two) if the constraints allow sharing. Figure A-7 shows possible implementations of a logic example.

Figure A-7: Keeping Sharable Resources in the Same Process



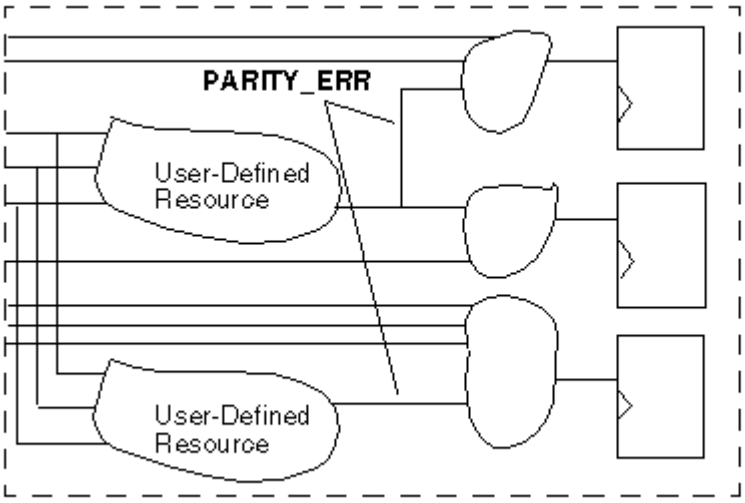
For more information about resource sharing, see the HDL Compiler documentation.

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. DC Explorer 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 A-8 illustrates splitting the load by multiple instantiation when the load on the signal PARITY_ERR is too large to meet constraints.

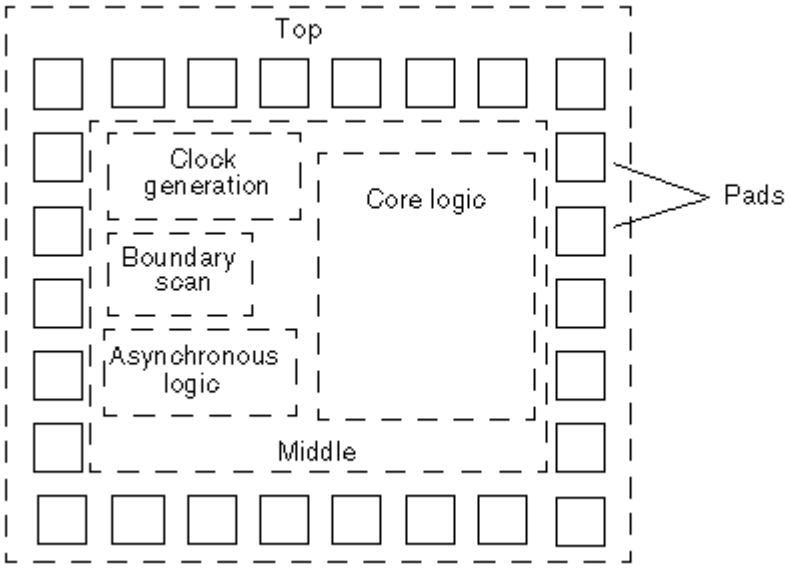
Figure A-8: Duplicating User-Defined Resources



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 A-9 shows the recommended partitioning for the top level of the design.

Figure A-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. This section provides information to help you write efficient code so that you can achieve your design target in the shortest possible time.

Topics include

- Writing Technology-Independent HDL
- Using HDL Constructs
- Writing Effective Code
- Instantiations of RTL PG Pins

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.

The following sections discuss various methods for keeping your HDL code technology independent.

- Inferring Components
- Designing State Machines

Inferring Components

HDL Compiler provides the capability to infer the following components:

- Multiplexers
- Registers

- Three-state drivers
- Multibit components

These inference capabilities are discussed in the following pages. For additional information and examples, see 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, DC Explorer maps the inferred MUX_OPs to multiplexer cells in the target logic library. DC Explorer determines the MUX_OP implementation during compile based on the design constraints.

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, DC Explorer 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 DC Explorer 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 datapath

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.

See Also

- The *Design Compiler User Guide*

Designing State Machines

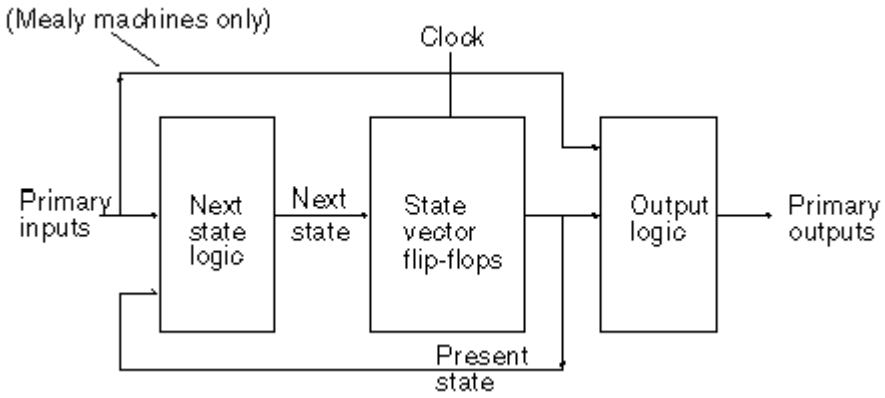
You can specify a state machine by using several different formats:

- Verilog
- VHDL
- State table
- PLA

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

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

Figure A-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.

See Also

- The *Design Compiler User Guide*

Using HDL Constructs

The following sections provide information and guidelines about the following specific HDL constructs:

- General HDL Constructs
- Verilog Macro Definitions
- 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 A-1: Incomplete Sensitivity List (Verilog)

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

Example A-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, DC Explorer creates a latch. The following examples show if statements that generate latches during synthesis.

Example A-3: Incorrect if Statement (Verilog)

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

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

Example A-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 myfile
module myfile (a, b, c);
   parameter WIDTH=128;
   input [WIDTH-1:0] a, b;
   output [WIDTH-1:0] c;
```

Example A-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;
```

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.

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

This section provides guidelines for writing efficient, readable HDL source code for synthesis. The guidelines cover

- Identifiers
- Expressions
- Functions
- 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 legibility 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 A-2 shows common suffixes and their meanings.

  Table A-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 global names within a function because the function might not be reevaluated if the global 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 global 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.

Instantiations of RTL PG Pins

DC Explorer can accept RTL designs containing a small number of PG pin connections on macros. 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 allow instantiations of PG pins in the RTL, set the `dc_allow_rtl_pg` variable to `true`, changing it from its default of `false`. To preserve the PG connections in a Verilog output, specify the `-pg -format verilog` options with the `write_file` command. To preserve the PG connections in a .ddc format output, specify the `-format ddc` option with the `write_file` command. When saving the design in .ddc format, you do not need to use the `-pg` option. When reading the .ddc file back into DC Explorer, ensure that the `dc_allow_rtl_pg` variable is set to `true`; otherwise, the tool issues a DDC-21 error message.

To preserve the PG connections in a Milkyway database, use the `write_milkyway` command. To pass the netlist to the IC Compiler, PrimeTime, or Formality tool, you can use a Verilog output, .ddc file, or Milkyway database.

To use PG pins in your RTL design, follow these guidelines:

- PG libraries are required
- FRAM must always have correct PG information
- The tool does not display the PG nets and pins; the `get_pins` command does 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 instantiate PG pins by name, such as `ref U1 (.pin(net), …);`
- PG nets can only connect to the following objects:
  - 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 marks cells that contain PG pins with the `dont_touch` attribute
- If your design contains UPF information, you must convert the PG connections from RTL to UPF by using the `convert_pg` command before you compile the design

The following example shows Verilog RTL code that instantiates two PG pins, my_vdd and my_vss:

```
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
```

B 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:

- Design Description
- Setup File
- Default Constraints File
- Read Script
- Compile Scripts

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 de_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 B-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 B-1: Block Diagram for the Design Example

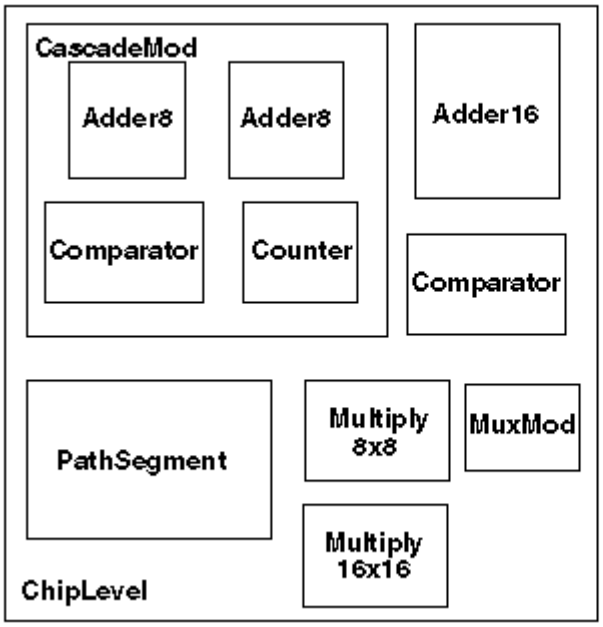
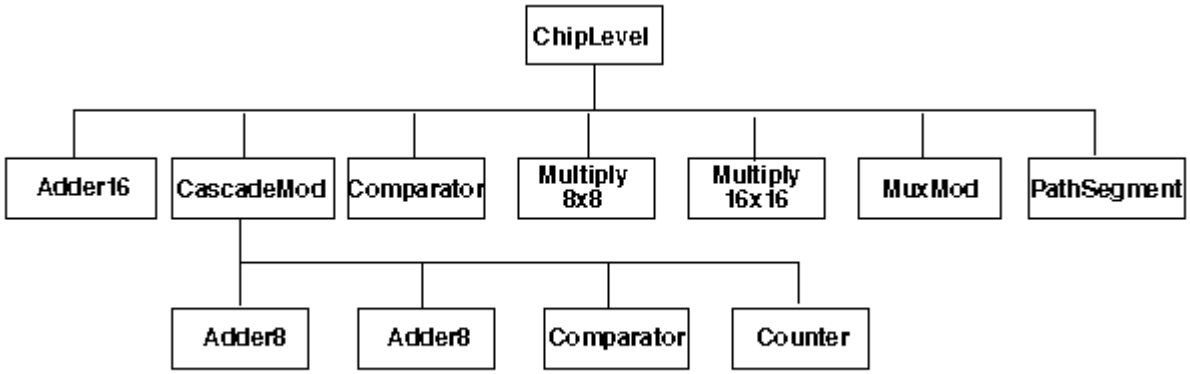


Figure B-2 shows the hierarchy for the design example.

Figure B-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.

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 B-1 through Example B-11 provide the Verilog source code for the ChipLevel design.

Example B-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 Reference
   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 B-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 B-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 B-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 B-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 B-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 B-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 B-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 B-9: def_macro.v

```
`define DATA 2'b00
`define REG 2'b01
`define STACKIN 2'b10
`define UPCOUT 2'b11
```

Example B-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 B-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 B-12 to your project working directory. This setup file is written in the Tcl subset and can be used in the dctcl command language.

Example B-12: .synopsys_dc.setup File

```
# Define the target library, symbol library,
# and link libraries
set target_library lsi_10k.db
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/"
```

See Also

- Setup Files
- *Using Tcl With Synopsys Tools*

Default Constraints File

The file shown in Example B-13 defines the default constraints for the design. In the scripts that follow, DC Explorer 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, DC Explorer uses the module-specific constraints.

Example B-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
```

Read Script

Example B-14 provides the dctcl script used to read in the ChipLevel design.

The `read.tcl` script reads design information from the specified Verilog files into memory.

Example B-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 B-15 through Example B-25 provide the dctcl scripts used to compile the ChipLevel design.

The compile script for each module is named for that module to ease recognition. The initial dctcl script files have the .tcl suffix. Scripts generated by the `write_script` command have the .wtcl suffix.

Example B-15: run.tcl

```
# Initial compile with estimated constraints
source "${script_path}initial_compile.tcl"

current_design ChipLevel
write_file -format ddc -hierarchy \
 -output "${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
write_file -format ddc -hierarchy \
 -output"${ddc_path}ChipLevel_final.ddc"
```

Example B-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 B-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

compile_exploration

write_file -format ddc -hierarchy -output "${ddc_path}${design}.ddc"

source "${script_path}report.tcl"
```

Example B-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

# Use compile-once, dont_touch approach for Comparator
set_dont_touch u12

compile_exploration

write_file -format ddc -hierarchy -output "${ddc_path}${design}.ddc"

source "${script_path}report.tcl"
```

Example B-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]

# Define design constraints
set_input_delay 1.35 -clock $clk_name {ain bin}
set_output_delay 5.1 -clock $clk_name {cp_out}

compile_exploration

write_file -format ddc -hierarchy -output "${ddc_path}${design}.ddc"

source "${script_path}report.tcl"
```

Example B-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}

compile_exploration

write_file -format ddc -hierarchy -output "${ddc_path}${design}.ddc"

source "${script_path}report.tcl"
```

Example B-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}

# 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_exploration

write_file -format ddc -hierarchy -output "${ddc_path}${design}.ddc"

source "${script_path}report.tcl"
report_timing_requirements -ignored \
   >> "${log_path}${rpt_file}"
```

Example B-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}

compile_exploration

write_file -format ddc -hierarchy -output "${ddc_path}${design}.ddc"

source "${script_path}report.tcl"
```

Example B-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}

# 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_exploration

write_file -format ddc -hierarchy -output "${ddc_path}${design}.ddc"

source "${script_path}report.tcl"
report_timing_requirements -ignored \
   >> "${log_path}${rpt_file}"
```

Example B-24: recompile.tcl

```
source "${script_path}read.tcl"

current_design ChipLevel
source "${script_path}defaults.con"

source "${script_path}adder16.wtcl"
compile_exploration
write_file -format ddc -hierarchy -output "${ddc_path}adder16_wtcl.ddc"
set rpt_file adder16_wtcl.rpt
source "${script_path}report.tcl"
```

```
source "${script_path}cascademod.wtcl"
dont_touch u12
compile_exploration
write_file -format ddc -hierarchy \
 -output "${ddc_path}cascademod_wtcl.ddc"
set rpt_file cascade_wtcl.rpt
source "${script_path}report.tcl"
source "${script_path}comp16.wtcl"
compile_exploration
write_file -format ddc -hierarchy -output "${ddc_path}comp16_wtcl.ddc"
set rpt_file comp16_wtcl.rpt
source "${script_path}report.tcl"

source "${script_path}mult8.wtcl"
compile_exploration
write_file -format ddc -hierarchy -output "${ddc_path}mult8_wtcl.ddc"
set rpt_file mult8_wtcl.rpt
source "${script_path}report.tcl"

source "${script_path}mult16.wtcl"
compile_exploration
write_file -format ddc -hierarchy -output "${ddc_path}mult16_wtcl.ddc"
set rpt_file mult16_wtcl.rpt
source "${script_path}report.tcl"
report_timing_requirements -ignored \
        >> "${log_path}${rpt_file}"

source "${script_path}muxmod.wtcl"
compile_exploration
write_file -format ddc -hierarchy -output "${ddc_path}muxmod_wtcl.ddc"
set rpt_file muxmod_wtcl.rpt
source "${script_path}report.tcl"

source "${script_path}pathseg.wtcl"
compile_exploration
write_file -format ddc -hierarchy -output "${ddc_path}pathseg_wtcl.ddc"
set rpt_file pathseg_wtcl.rpt
source "${script_path}report.tcl"
report_timing_requirements -ignored \
   >> "${log_path}${rpt_file}"
```

Example B-25: 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_constraint -all_violators -verbose \
   >> "${log_path}${rpt_file}"
report_timing -path_type end >> "${log_path}${rpt_file}"
report_timing -max_paths $maxpaths \
   >> "${log_path}${rpt_file}"
report_qor >> "${log_path}${rpt_file}"
```

C Basic Commands

This appendix lists the basic de_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.

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_latency`**, **`set_clock_latency`**, **`set_clock_latency`**

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_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_design_mismatch

Reports design mismatches that were circumvented to allow linking the design.

report_hierarchy

Lists the subdesigns of the current design.

report_missing_constraints

Reports missing constraints in 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.

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 postprocessing 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. DC Explorer 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.

DC Explorer 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 DC Explorer 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 dimensionless number that represents a numerical contribution to the total fanout. Fanout load is not the same as load, which is a capacitance value.

DC Explorer 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, DC Explorer 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 DC Explorer. 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 DC Explorer uses to resolve cell references. Link libraries can contain logic 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 DC Explorer 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 DC Explorer 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. DC Explorer 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 technology library in an `operating_conditions` group.

optimization

The step in the logic synthesis process in which DC Explorer 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 might 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, DC Explorer structures your design.

synthesis

A software process that generates an optimized gate-level netlist, which is based on a technology 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. DC Explorer uses symbol libraries to generate the design schematic. You can use Design Vision to view the design schematic.

target library

The logic library to which DC Explorer 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 DC Explorer. For DC Explorer 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.