Design Compiler®
User Guide

Version Z-2007.03, March 2007

Comments?
Send comments on the documentation by going to http://solvnet.synopsys.com, then clicking "Enter a Call to the Support Center."

SYNOPSYS®

Copyright Notice and Proprietary Information

Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

> "This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of _____ and its employees. This is copy number _____."

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.

Registered Trademarks (®)

Synopsys, AMPS, Cadabra, CATS, CRITIC, CSim, Design Compiler, DesignPower, DesignWare, EPIC, Formality, HSIM, HSPICE, iN-Phase, in-Sync, Leda, MAST, ModelTools, NanoSim, OpenVera, PathMill, Photolynx, Physical Compiler, PrimeTime, SiVL, SNUG, SolvNet, System Compiler, TetraMAX, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

Trademarks (™)

AFGen, Apollo, Astro, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, Columbia, Columbia-CE, Cosmos, CosmosEnterprise, CosmosLE, CosmosScope, CosmosSE, DC Expert, DC Professional, DC Ultra, Design Analyzer, Design Vision, DesignerHDL, Direct Silicon Access, Discovery, Encore, Galaxy, HANEX, HDL Compiler, Hercules, Hierarchical Optimization Technology, HSIM^{plus}, HSPICE-Link, i-Virtual Stepper, iN-Tandem, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Magellan, Mars, Mars-Xtalk, Milkyway, ModelSource, Module Compiler, Planet, Planet-PL, Polaris, Power Compiler, Raphael, Raphael-NES, Saturn, Scirocco, Scirocco-i, Star-RCXT, Star-SimXT, Taurus, TSUPREM-4, VCS Express, VCSi, VHDL Compiler, VirSim, and VMC are trademarks of Synopsys, Inc.

Service Marks (^{SM})

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.
ARM and AMBA are registered trademarks of ARM Limited.
Saber is a registered trademark of SabreMark Limited Partnership and is used under license.
All other product or company names may be trademarks of their respective owners.

Contents

Preface

This preface includes the following sections:

What's New in This Release

This section describes the new features, enhancements, and changes included in Design Compiler version Z-2007.03, which provides significant improvements in runtime and quality when compared to version Y-2006.06.

Enhancements in Topographical Technology

Design Compiler version Z-2007.03 delivers the following enhancements in topographical technology:

- Intelligent macro name matching in DEF

  By default, when the `extract_physical_constraints` command applies physical constraints in topographical mode, it uses an intelligent name matching algorithm to match macros and ports in the Design Exchange Format (DEF) file with in-memory objects, irrespective of the hierarchy separator or bus notation. You can use the `-exact` option to disable intelligent name matching. In addition, a new command, `set_fuzzy_query_options`, allows you to define the rules used by the intelligent name matching algorithm.

- Support for incremental Design Exchange Format (DEF) with the `extract_physical_constraints` command

  By default, the `extract_physical_constraints` command runs in incremental mode. That is, if you use the command to process multiple Design Exchange Format (DEF) files, the command preserves existing physical annotations on the design. You can use the `-no_incremental` option to disable incremental mode.

- Support for saving physical constraints in the .ddc format

  In topographical mode, user-specified physical constraints can be saved in the .ddc file, in addition to virtual placements. The `write_ddc` command saves the following physical constraints:

  - `set_placement_area`
  - `set_rectilinear_outline`
  - `set_port_location`
  - `set_cell_location`
  - `create_placement_keepout`
  - `create_voltage_area`
  - `set_utilization`
  - `set_aspect_ratio`
  - `set_port_side`

  Saved physical constraints can only be retrieved in topographical mode. Design Compiler wireload mode, JupiterXT, or IC Compiler cannot retrieve this information.

- Support for top-level design stitching

  In topographical mode, the `compile_ultra` command has a new option, `-top,` which enables you to stitch compiled physical blocks into the top-level design. Top-level design stitching in topographical mode is similar to the functionality provided by the `compile -top` command except that in topographical mode, the tool can map any unmapped top-level glue logic. The top-level design stitching capability also supports scan insertion (by using the `insert_dft` command) of top-level registers.

You can also use the `compile_ultra -top` command to stitch compiled logical blocks in Design Compiler wireload mode.

- Support for the `create_mw_lib` command

  In version Z-2007.03, topographical mode supports the `create_mw_lib` command so that there is a consistent user interface for Milkyway library support between topographical mode and IC Compiler.

  The `create_mw_design` command is still supported; however, it is recommended that you use the `create_mw_lib` command to create the Milkyway library. The `open_mw_lib` command is necessary to open the newly created Milkyway library. The following Milkyway library commands are also supported:

  - `copy_mw_lib`
  - `close_mw_lib`
  - `report_mw_lib`
  - `current_mw_lib`
  - `set_tlu_plus_files`
  - `check_tlu_plus_files`
  - `write_mw_lib_files`
  - `set_mw_lib_references`

- New black-box support

  In version Z-2007.03, topographical mode can create missing physical library cells for the following:

  - Logical library cells (leaf cells and macros)

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

Because black-box support is enabled by default, you do not need to set any environment variables such as `physopt_create_missing_physical_libcells`. The tool issues the following warning message when it creates physical library cells:

```
Warning: Created physical library cell for logical library
%s. (OPT-1413)
```

- DFT Flow in topographical mode

  In version 2007.03, topographical mode supports the following additional DFT features:

  - On-chip clocking support
  - Adaptive scan (regular and X-tolerant DFTMax)
  - Core wrapping
  - Hierarchical Flows
  - Boundary-scan design
  - Multivoltage flow

Improvement in Quality of Results

Design Compiler version Z-2007.03 provides significant improvements in runtime and quality when compared to version Y-2006.06 as a result of the following enhancements:

- Delay optimizations
  - Delay-based sequential optimization in the early phase of sequential mapping
  - New sequential mapping backend algorithm to remap sequential components for better area and delay
  - Improved library-analysis based mapping algorithm for delay
  - Cone-factor Shannon decomposition
- Area optimizations
  - Priority decoder optimization
  - Hierarchy-based incremental implementation selection for DesignWare
  - Improved autoungrouping strategy
  - Automatic shift register detection

    In version Z-2007.03, Design Compiler automatically identifies shift registers in a design and performs scan replacement only on the first flip-flop of the shift register. This enhancement improves sequential design area and reduces congestion by using fewer scan-signals for routing. You can use the `compile_seqmap_identify_shift_registers` variable (default: true) to control automatic identification of shift registers.
- Adaptive retiming

  The `compile_ultra` command supports the `-retime` option, which enables Design Compiler to automatically perform register moves during optimization. This capability—called adaptive retiming—enables the tool to make retiming moves to improve

timing. A new command, `set_dont_retime`, allows selective retiming. This feature is supported in both Design Compiler wireload mode and topographical mode.

Enhanced Usability

Design Compiler version Z-2007.03 improves ease-of-use in the following ways:

- New options to the `compile_ultra` command

  The `compile_ultra` command has six new options: `-incremental`, `-no_design_rule`, `-only_design_rule`, `-top`, `-retime`, `-gate_clock`. These enhancements make the `compile_ultra` interface consistent across Design Compiler wireload mode and topographical mode and improve usability of the command.Table 1 lists the supported options in both modes:

Table 1 Supported Options

Option	Wireload Mode	Topographical Mode
`-scan`	X	X
`-exact_map`	X	X
`-no_autoungroup`	X	X
`-no_boundary_optimization`	X	X
`-no_seq_output_inversion`	X	X
`-area_high_effort_script`	X	X
`-timing_high_effort_script`	X	X
`-incremental`	X (new)	X

Table 1 Supported Options (Continued)

Option	Wireload Mode	Topographical Mode
`-no_design_rule`	X (new)	X
`-only_design_rule`	X (new)	X
`-top`	X (new)	X (new)
`-retime`	X (new)	X (new)
`-gate_clock`	X (new)	X (new)
`-only_hold_time`	Not supported	Not supported

- Support for new netlist editing commands

  In version Z-2007.03, netlist editing commands similar to IC Compiler are available in both Design Compiler wireload mode and topographical mode. These commands are described below:

  - Resizing a cell

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

  - Inserting buffers or inverter pairs

    You can use the `insert_buffer` command to add a buffer at pins or ports. The `-inverter_pair` option allows you specify that a pair of inverting library cells is to be inserted instead of

a single non-inverting library cell. To retrieve a collection of all buffers and inverters from the library, you can use the `get_buffer` command.

- Inserting repeaters

  The `-no_of_cells` option of the `insert_buffer` command allows you to select a driver of a two-pin net and insert a chain of single-fanout buffers in the net driven by this driver.

- Removing buffers

  You can use the `remove_buffer` command to remove buffers.

- The `set_dont_touch_network` command has a new option, `-no_propagate`, which enables you to specify that the dont_touch network should not be propagated through logic gates.

- Starting version Z-2007.03, the `report_timing` command has the following enhancements:

  - The `-slack_greater_than` *`greater_than_slack_limit`* option to specify that only those paths with a slack greater than the *`greater_slack_limit`* are to be reported and the `-slack_lesser_than` *`lesser_than_slack_limit`* option to specify that only those paths with a slack less than the *`lesser_slack_limit`* are to be reported.

    You can use the two options together to report only those paths within or outside a given slack range.

  - You can use the `-through` option of the `report_timing` command multiple times to specify paths that traverse multiple points in the design.

For example, the following command returns a timing report for the path starting from A, passing through B or C, passing through D, and ending at E.

```
dc_shell-xg-t> report_timing -from A -through {B C} \
                    -through D –to E
```

- Support for the `set_clock_groups` command

  Starting from version Z-2007.03, you can use the `set_clock_groups` command to specify clock groups that are mutually exclusive or asynchronous with respect to each other so that paths between these clocks are not considered during timing analysis.

- Support for creating non-unate generated clocks

  The `create_generated_clock` has a new option, `-preinvert`, which enables you to create a generated clock based on the inverted clock signal. You can use this option when you are creating generated clocks based on a non-unate master clock.

- Support for the `set_clock_sense` command

  Starting from version Z-2007.03, you can use the `set_clock_sense` command to define the clock sense at non-unate points in the clock network.

- Improved support for verification

  Name changes in the design caused by the `rename_design` command are now recorded in the automated setup file (.svf), improving name matching and verification in Formality when the `rename_design` command is used. In addition, the `guide_environment` command allows a set of name/value pairs to be passed from Design Compiler to Formality.

- New Uniquification Variable in Bottom-Up Flow

  In version Z-2007.03, Design Compiler provides a new variable, `compile_keep_original_for_external_references`, which enables compile to keep the original design when there is an external reference to the design.

  By default, compile modifies the original copy of the designs in the current design. When the variable is set to true, the original design and its sub-designs are copied and preserved (before doing any modifications during compile) if there is an external reference to this design.

  Typically, you require this variable only when you are doing a bottom-up compile without setting a `dont_touch` attribute on all the sub-designs, especially those with boundary optimizations turned on. If there is a `dont_touch` attribute on any of the instances of the design or in the design, this variable has no effect.

- Datapath Extraction Across Registers

  In version Z-2007.03, a new variable, `hlo_dp_extract_across_registers`, enables datapath optimization across register banks for carry-save tree implementations. When the variable is set to `selective` (the default), the datapath is extracted across register banks for carry-save tree implementation. Such implementations might reduce area and timing by avoiding the need for a final adder in the driving datapath block.

Known Limitations and Resolved STARs

Information about known problems and limitations, as well as about resolved Synopsys Technical Action Requests (STARs), is available in the *Design Compiler Release Notes* in SolvNet.

To see the *Design Compiler Release Notes,*

1. Go to http://solvnet.synopsys.com/ReleaseNotes. (If prompted, enter your user name and password. If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.)
2. Click Design Compiler, then click the release you want in the list that appears at the bottom.

About This Manual

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

This manual does not cover asynchronous design, I/O pad synthesis, test synthesis, simulation, 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.

This manual supports version Z-2007.03 of the Synopsys synthesis tools, whether they are running under the UNIX operating system or the Linux operating system. The main text of this manual describes UNIX operation. This manual describes features available in XG mode, which is the default mode of operation.

Audience

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

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

Related Publications

For additional information about Design Compiler, see

- Synopsys Online Documentation (SOLD), which is included with the software for CD users or is available to download through the Synopsys electronic software transfer (EST) system
- Documentation on the Web, which is available through SolvNet at http://solvnet.synopsys.com/DocsOnWeb
- The Synopsys MediaDocs Shop, from which you can order printed copies of Synopsys documents, at http://mediadocs.synopsys.com

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

- Automated Chip Synthesis
- Design Budgeting
- Design Vision
- DesignWare components
- DFT Compiler
- PrimeTime
- Power Compiler
- HDL Compiler

Also see the following related documents:

- *Interface Logic Model User Guide*
- *Using Tcl With Synopsys Tools*
- *Synthesis Master Index*

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
`Courier`	Indicates command syntax.
`Courier italic`	Indicates a user-defined value in Synopsys syntax, such as *`object_name`*. (A user-defined value that is not Synopsys syntax, such as a user-defined value in a Verilog or VHDL statement, is indicated by regular text font italic.)
Courier bold	Indicates user input—text you type verbatim—in Synopsys syntax and examples. (User input that is not Synopsys syntax, such as a user name or password you enter in a GUI, is indicated by regular text font bold.)
[ ]	Denotes optional parameters, such as *`pin1`* [*`pin2 ... pinN`*]
|	Indicates a choice among alternatives, such as `low | medium | high` (This example indicates that you can enter one of three possible values for an option: low, medium, or high.)
_	Connects terms that are read as a single term by the system, such as `set_annotated_delay`
Control-c	Indicates a keyboard combination, such as holding down the Control key and pressing c.
\	Indicates a continuation of a command line.
/	Indicates levels of directory structure.
Edit > Copy	Indicates a path to a menu command, such as opening the Edit menu and choosing Copy.

Customer Support

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

Accessing SolvNet

SolvNet includes an electronic knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. SolvNet also gives you access to a wide range of Synopsys online services including software downloads, documentation on the Web, and “Enter a Call to the Support Center.”

To access SolvNet,

1. Go to the SolvNet Web page at http://solvnet.synopsys.com.
2. If prompted, enter your user name and password. (If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.)

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

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 call to your local support center from the Web by going to http://solvnet.synopsys.com (Synopsys user name and password required), then clicking “Enter a Call to the Support Center.”
- Send an e-mail message to your local support center.
  - E-mail support_center@synopsys.com from within North America.
  - Find other local support center e-mail addresses at http://www.synopsys.com/support/support_ctr.
- Telephone your local support center.
  - Call (800) 245-8005 from within the continental United States.
  - Call (650) 584-4200 from Canada.
  - Find other local support center telephone numbers at http://www.synopsys.com/support/support_ctr.

1

Introduction to Design Compiler

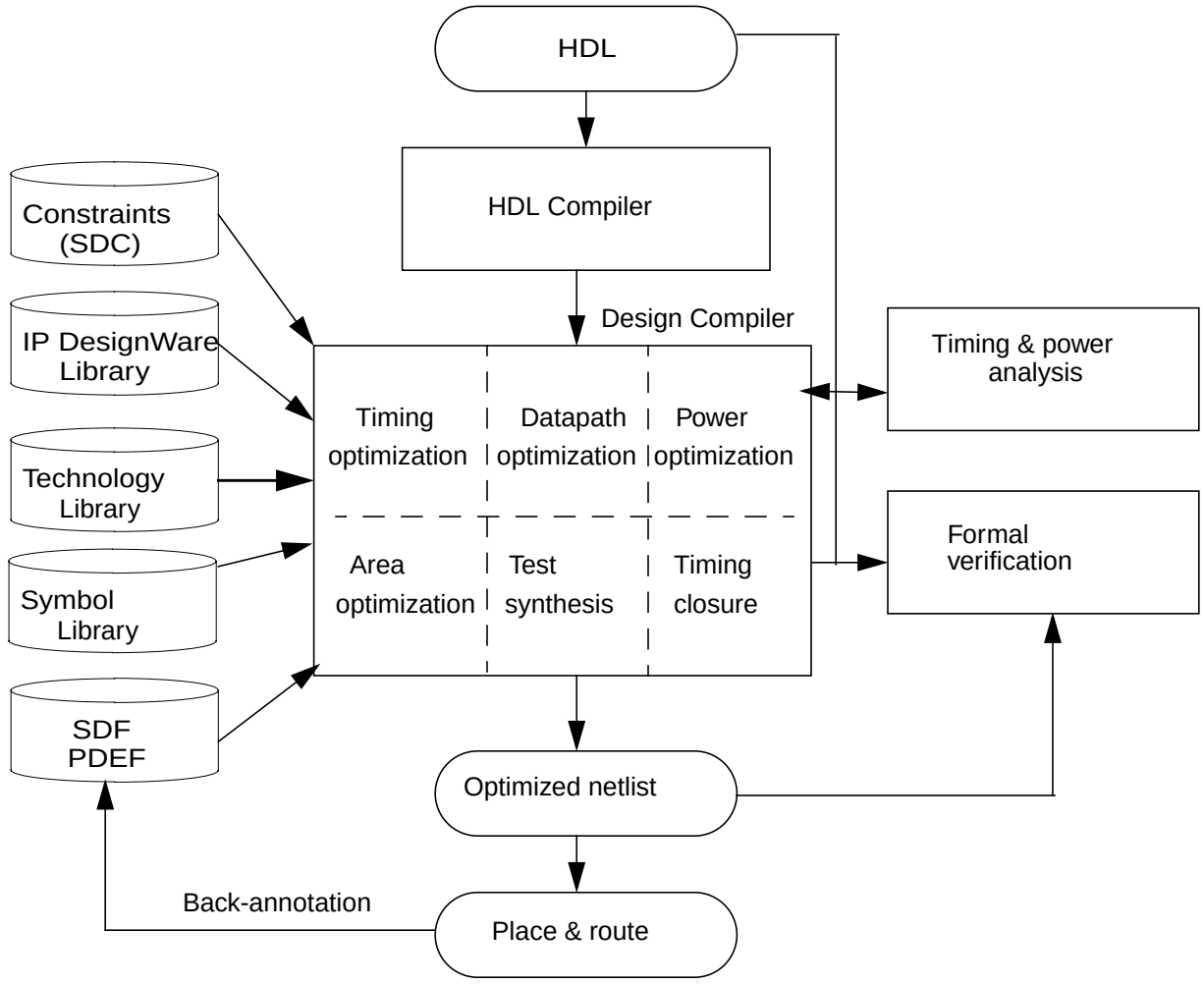
The Design Compiler tool is the core of the Synopsys synthesis products. Design Compiler optimizes designs to provide the smallest and fastest logical representation of a given function. It comprises tools that synthesize your HDL designs into optimized technology-dependent, gate-level designs. It supports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

This chapter includes the following sections:

Design Compiler and the Design Flow

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

Figure 1-1 Design Compiler and the Design Flow



You use Design Compiler for logic synthesis, which is the process of converting a design description written in a hardware description language such as Verilog or VHDL into an optimized gate-level netlist mapped to a specific technology library. The steps in the synthesis process are as follows:

1. The input design files for Design Compiler are often written using a hardware description language (HDL) such as Verilog or VHDL.

2. Design Compiler uses technology libraries, synthetic or DesignWare libraries, and symbol libraries to implement synthesis and to display synthesis results graphically.

   During the synthesis process, Design Compiler translates the HDL description to components extracted from the generic technology (GTECH) library and DesignWare library. The GTECH library consists of basic logic gates and flip-flops. The DesignWare library contains more complex cells such as adders and comparators. Both the GTECH and DesignWare libraries are technology independent, that is, they are not mapped to a specific technology library. Design Compiler uses the symbol library to generate the design schematic.

3. After translating the HDL description to gates, Design Compiler optimizes and maps the design to a specific technology library, known as the target library. The process is constraint driven. Constraints are the designer's specification of timing and environmental restrictions under which synthesis is to be performed.

4. After the design is optimized, it is ready for test synthesis. Test synthesis is the process by which designers can integrate test logic into a design during logic synthesis. Test synthesis enables designers to ensure that a design is testable and resolve any test issues early in the design cycle.

The result of the logic synthesis process is an optimized gate-level netlist, which is a list of circuit elements and their interconnections.

5. After test synthesis, the design is ready for the place and route tools, which place and interconnect cells in the design. Based on the physical routing, the designer can back-annotate the design with actual interconnect delays; Design Compiler can then resynthesize the design for more accurate timing analysis.

Design Compiler Family

Synopsys provides an integrated RTL synthesis solution. Using Design Compiler tools, you can

- Produce fast, area-efficient ASIC designs by employing user-specified gate-array, FPGA, or standard-cell libraries
- Translate designs from one technology to another
- Explore design tradeoffs involving design constraints such as timing, area, and power under various loading, temperature, and voltage conditions
- Synthesize and optimize finite state machines
- Integrate netlist inputs and netlist or schematic outputs into third-party environments while still supporting delay information and place and route constraints
- Create and partition hierarchical schematics automatically

DC Expert

At the core of the Synopsys' RTL synthesis solution is the DC Expert. DC Expert is applied to high-performance ASIC and IC designs.

DC Expert provides the following features:

- Hierarchical compile (top down or bottom up)
- Full and incremental compile techniques
- Sequential optimization for complex flip-flops and latches
- Time borrowing for latch-based designs
- Timing analysis
- Buffer balancing (within hierarchical blocks)
- Command-line interface and graphical user interface
- Budgeting, the process of allocating timing and environment constraints among blocks in a design
- Automated chip synthesis, a set of Design Compiler commands that fully automate the partitioning, budgeting, and distributed synthesis flow for large designs

DC Ultra

The DC Ultra tool is applied to high-performance deep submicron ASIC and IC designs, where maximum control over the optimization process is required.

In addition to the DC Expert capabilities, DC Ultra provides the following features:

- Additional high-effort delay optimization algorithms
- Advanced arithmetic optimization
- Integrated datapath partitioning and synthesis capabilities
- Finite state machine (FSM) optimization
- Advanced critical path resynthesis
- Register retiming, the process by which the tool moves registers through combinational gates to improve timing
- Support for advanced cell modeling, that is, the cell-degradation design rule
- Advanced timing analysis

HDL Compiler Tools

The HDL compiler reads HDL files and performs translation and architectural optimization of the designs. For more information about the HDL Compiler tools, see the HDL Compiler documentation.

DesignWare Library

A DesignWare library is a collection of reusable circuit-design building blocks (components) that are tightly integrated into the Synopsys synthesis environment. During synthesis, Design Compiler selects the right component with the best speed and area optimization from the DesignWare Library. For more information, see the DesignWare Library documentation.

DFT Compiler

The DFT Compiler tool is the Synopsys test synthesis solution. DFT Compiler provides integrated design-for-test capabilities, including constraint-driven scan insertion during compile. The DFT Compiler tool is applied to high-performance ASIC and IC designs that utilize scan test techniques. For more information, see the DFT Compiler documentation.

Power Compiler

The Power Compiler tool offers a complete methodology for power, including analyzing and optimizing designs for static and dynamic power consumption. For more information about these power capabilities, see the *Power Compiler User Guide*.

Design Vision

The Design Vision is a graphical user interface (GUI) to the Synopsys synthesis environment and an analysis tool for viewing and analysing designs at the generic technology (GTECH) level and gate level. Design Vision provides menus and dialog boxes for implementing Design Compiler commands. It also provides graphical displays, such as design schematics. For more information, see the *Design Vision User Guide* and *Design Vision Help*.

2

Design Compiler Basics

This chapter provides basic information about Design Compiler functions. The chapter presents both high-level and basic synthesis design flows. Standard user tasks, from design preparation and library specification to compile strategies, optimization, and results analysis, are introduced as part of the basic synthesis design flow presentation.

This chapter includes the following sections:

Note:

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

Synthesis is the process that generates a gate-level netlist for an IC design that has been defined using a Hardware Description Language (HDL). Synthesis includes reading the HDL source code and optimizing the design from that description.

Optimization is the step in the synthesis process that attempts to implement a combination of library cells that best meet the functional, timing, and area requirements of the design.

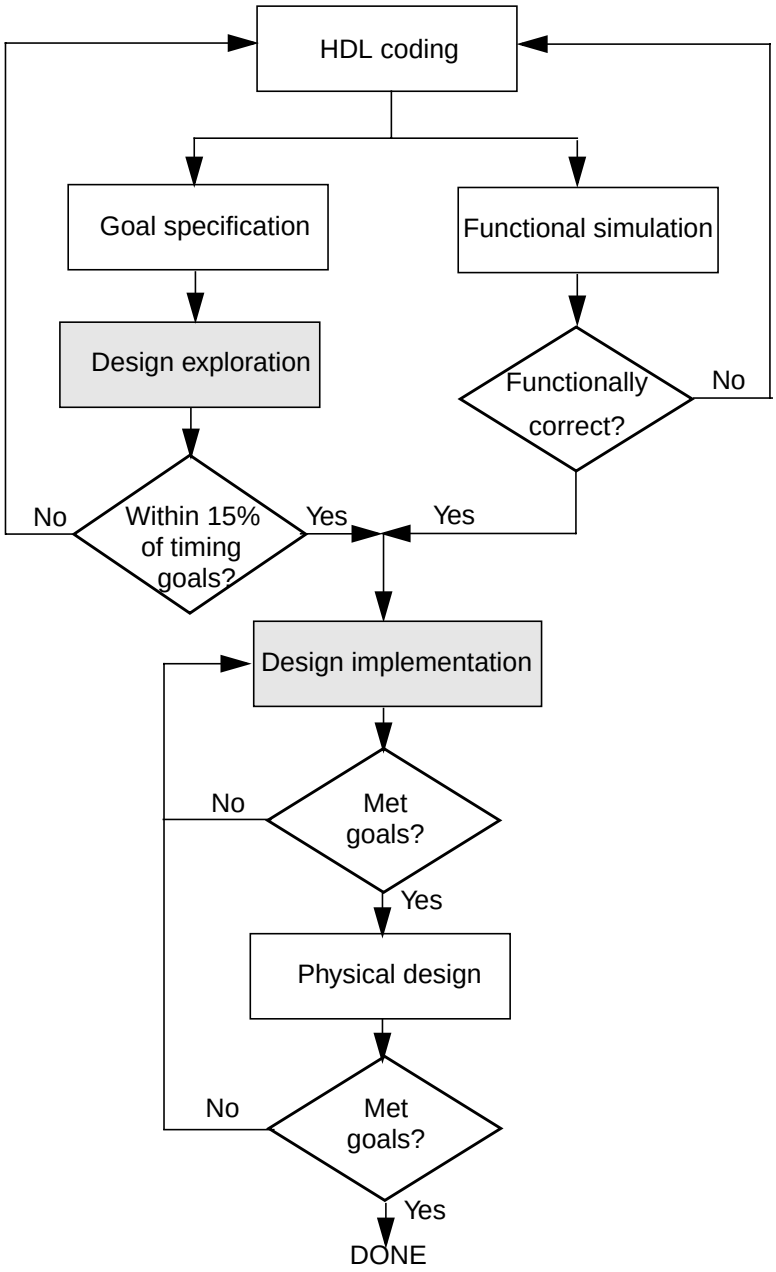
Compile is the Design Compiler command and process that executes the optimization step. After you read in the design and perform other necessary tasks, you invoke the `compile_ultra` command or `compile` command to generate a gate-level netlist for the design.

The High-Level Design Flow

In a basic high-level design flow, Design Compiler is used in both the design exploration stage and the final design implementation stage. In the exploratory stage, you use Design Compiler to carry out a preliminary, or default, synthesis. In the design implementation stage, you use the full power of Design Compiler to synthesize the design.

Figure 2-1 shows the high-level design flow. The shaded areas indicate where Design Compiler synthesis tasks occur in the flow.

Figure 2-1 Basic High-Level Design Flow



Using the design flow shown in Figure 2-1, you perform the following steps:

1. Start by writing an HDL description (Verilog or VHDL) of your design. Use good coding practices to facilitate successful Design Compiler synthesis of the design.
2. Perform design exploration and functional simulation in parallel.
    - In design exploration, use Design Compiler to (a) implement specific design goals (design rules and optimization constraints) and (b) carry out a preliminary, “default” synthesis (using only the Design Compiler default options).
    - If design exploration fails to meet timing goals by more than 15 percent, modify your design goals and constraints, or improve the HDL code. Then repeat both design exploration and functional simulation.
    - In functional simulation, determine whether the design performs the desired functions by using an appropriate simulation tool.
    - If the design does not function as required, you must modify the HDL code and repeat both design exploration and functional simulation.
    - Continue performing design exploration and functional simulation until the design is functioning correctly and is within 15 percent of the timing goals.
3. Perform design implementation synthesis by using Design Compiler to meet design goals.

After synthesizing the design into a gate-level netlist, verify that the design meets your goals. If the design does not meet your goals, generate and analyze various reports to determine the techniques you might use to correct the problems.

4. After the design meets functionality, timing, and other design goals, complete the physical design (either in-house or by sending it to your semiconductor vendor).

   Analyze the physical design's performance by using back-annotated data. If the results do not meet design goals, return to step 3. If the results meet your design goals, you are finished with the design cycle.

Running Design Compiler

This section provides the basic information you need to run Design Compiler. It includes the following sections:

- Design Compiler Mode of Operation
- Design Compiler Interfaces
- Setup Files
- Starting Design Compiler
- Exiting Design Compiler
- Getting Command Help
- Using Command Log Files
- Using Script Files
- Working with Licenses

Design Compiler Mode of Operation

The default mode of operation for Design Compiler is XG mode. XG mode uses optimized memory management techniques that increase the tool capacity and can reduce runtime. In XG mode, you use the dctcl command language to interact with Design Compiler. The Design Compiler documentation set describes the default XG mode.

Design Compiler Interfaces

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

You can interact with the Design Compiler shell by using dctcl, which is based on the tool command language (Tcl) and includes certain command extensions needed to implement specific Design Compiler functionality.

The dctcl command language provides capabilities similar to UNIX command shells, including variables, conditional execution of commands, and control flow commands. You can execute Design Compiler commands in the following ways:

- By entering single commands interactively in the shell
- By running one or more command scripts, which are text files of commands

- By typing single commands interactively on the console command line in the Design Vision window.

  You can use this approach to supplement the subset of Design Compiler commands available through the menu interface. For more information on Design Vision, see the *Design Vision User Guide* and Design Vision online Help.

Setup Files

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

Design Compiler 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 (the directory from which you invoke Design Compiler)

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 Design Compiler setup information for all users at your site. Only the system administrator can modify this file.
User-defined .synopsys_dc.setup file	User home directory	This file contains variables that define your preferences for the Design Compiler working environment. The variables in this file override the corresponding variables in the systemwide setup file.
Design-specific .synopsys_dc.setup file	Working directory from which you started Design Compiler	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 Design Compiler from this directory. Variables defined in this file override the corresponding variables in the user-defined and systemwide setup files.

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

Example 2-1 .synopsys_dc.setup File

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

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

Starting Design Compiler

To start Design Compiler, enter

dc_shell-xg-t

You can also enter one of the following commands:

dc_shell-xg_mode
dc_shell-tcl_mode -xg_mode

The resulting command prompt is

dc_shell-xg-t>

You can also include numerous options in these command lines, such as

- `-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 dc_shell prompt
- `-x` to include a dc_shell statement that is executed at startup

For a detailed list of options, see the *Design Compiler Command Line Interface Guide* and the man pages for dc_shell.

At startup, dc_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 `-x` and `-f` options, respectively, on the command line
4. Displays the program header and dc_shell prompt in the window from which you invoked dc_shell. The program header lists all features for which your site is licensed.

Exiting Design Compiler

You can exit Design Compiler at any time and return to the operating system.

Note:

By default, dc_shell saves the session information in the command.log file. However, if you change the name of the `sh_command_log_file` after you start the tool, session information might be lost.

Also, dc_shell does not automatically save the designs loaded in memory. If you want to save these designs before exiting, use the `write` command. For example,

```
dc_shell-xg-t> write -format ddc -hierarchy -output \
         my_design.ddc
```

To exit dc_shell, do one of the following:

- Enter quit.
- Enter exit.
- Press Control-d, if you are running Design Compiler in interactive mode and the tool is busy.

When you exit dc_shell, text similar to the following appears (the memory and the CPU numbers reflect your actual usage):

```
Memory usage for this session 14312 Kbytes.
CPU usage for this session 14 seconds.

Thank you ...
```

Getting Command Help

Design Compiler provides three levels of command help:

- A list of commands
- Command usage help
- Topic help

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

```
dc_shell-xg-t> help
```

To get help about a particular dc_shell command, enter the command name with the `-help` option. The syntax is

```
dc_shell-xg-t> command_name -help
```

To get topic help in dc_shell, enter

```
dc_shell-xg-t> man topic
```

where *topic* is the name of a shell command, variable, or variable group.

Using the `man` command, you can display the man pages for the topic while you are interactively running Design Compiler.

Using Command Log Files

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

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

Each Design Compiler 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

Using the Filename Log File

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

Using Script Files

You can create a command script file by placing a sequence of dc_shell commands in a text file. Any dc_shell command can be executed within a script file.

In dctcl, a “#” at the beginning of a line denotes a comment. For example,

```
# This is a comment
```

To execute a script file, use the `source` command.

For more information about script files, see the *Design Compiler Command-Line Interface Guide*.

Working With Licenses

In working with licenses, you need to determine what licenses are in use and know how to obtain and release licenses.

Listing the Licenses in Use

To view the licenses that are currently checked out, use the `list_license` command. To determine which licenses are already checked out, use the `license_users` command. For example,

```
dc_shell-xg-t> license_users
bill@eng1 Design-Compiler
matt@eng2 Design-Compiler, DC-Ultra-Opt
2 users listed.
```

Getting Licenses

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

If you know the tools and interfaces you need, you can use the `get_license` command to check out those licenses. This ensures that each license is available when you are ready to use it. For example,

```
dc_shell-xg-t> get_license HDL-Compiler
```

Once a license is checked out, it remains checked out until you release it or exit dc_shell.

Enabling License Queuing

Design Compiler has a license queuing functionality that allows your application to wait for licenses to become available if all licenses are in use. To enable this functionality, set the `SNPSLMD_QUEUE` environment variable to true. 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 two processes are waiting indefinitely for a license that the other process owns. Consider the following scenario:

- Two active processes, P1 and P2, are running Design Compiler
- You have the following licenses: Two Design Compiler licenses, a Power Compiler license, and an HDL Compiler license.

For P1 and P2 to be active, they should have checked out one Design Compiler license each. Assume that P1 has acquired the HDL Compiler license and P2 has acquired the Power Compiler license. If P1 and P2 now both attempt to acquire the license held by the other process, both processes wait indefinitely. The `SNPS_MAX_WAITTIME` environment variable and the `SNPS_MAX_QUEUETIME` environment variable help prevent such situations. You can use these variables only if the `SNPSLMD_QUEUE` environment variable is set to true.

The `SNPS_MAX_WAITTIME` variable specifies the maximum wait time for the first key license that you require, for example, starting dc_shell. Consider the following scenario:

You have two Design-Compiler licenses both of which are in use and are attempting to start a third dc_shell process. The queuing functionality places this last job in the queue for the specified wait time. The default wait time is 259,200 seconds (or 72 hours). If the license is still not available after the predefined time, you might see a message similar to the following:

```
Information: Timeout while waiting for feature ‘Design
Compiler’. (DCSH-17)
```

The SNPS_MAX_QUEUETIME variable specifies the maximum wait time for checking out subsequent licenses within the same dc_shell process. You use this variable after you have successfully checked out the first license to start dc_shell. Consider the following scenario:

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

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

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

```
Information: Successfully checked out feature
'DC-Ultra-Opt'. (DCSH-14)
Information: Started queuing for feature
'DC-Ultra-Features'. (DCSH-15)
Information: Still waiting for feature 'DC-Ultra-Features'.
(DCSH-16)
```

Releasing Licenses

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

```
dc_shell-xg-t> remove_license HDL-Compiler
```

Following the Basic Synthesis Flow

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

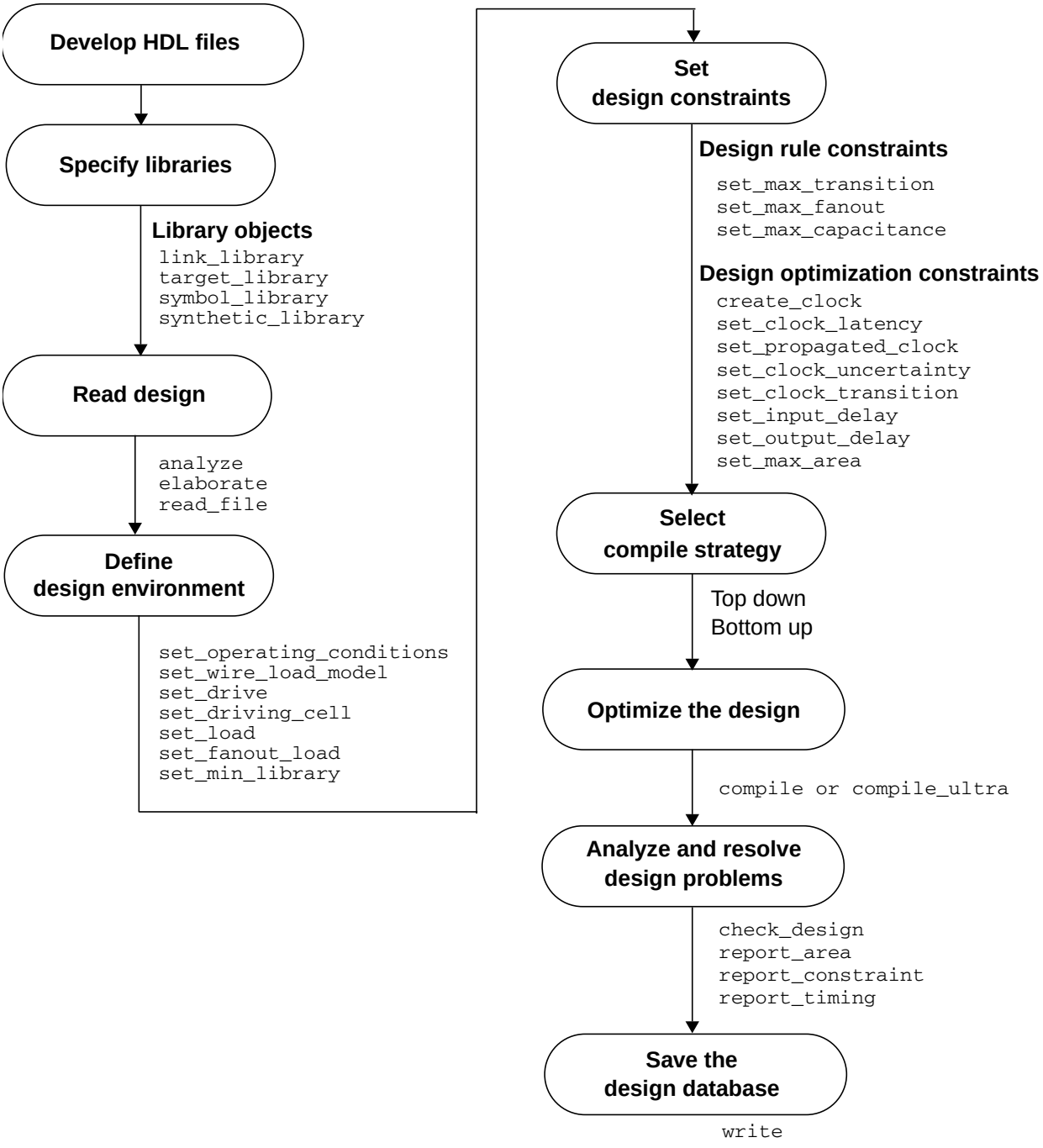
Also listed in Figure 2-2 are the basic dc_shell commands that are commonly used in each step of the basic flow. For example, the commands `analyze`, `elaborate`, and `read_file` are used in the step that reads design files into memory. All the commands shown in Figure 2-2 can take options, but no options are shown in the figure.

Note:

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

Following Figure 2-2 is a discussion of each step in the flow, including a reference to the chapter in this manual where you can find more information.

Figure 2-2 Basic Synthesis Flow



The basic synthesis flow consists of the following steps:

1. Develop HDL Files

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

Note:

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

See Chapter 3, “Preparing Design Files for Synthesis.”

2. Specify Libraries

You specify the link, target, symbol, and synthetic libraries for Design Compiler by using the `link_library`, `target_library`, `symbol_library`, and `synthetic_library` commands.

The link and target libraries are technology libraries that define the semiconductor vendor’s set of cells and related information, such as cell names, cell pin names, delay arcs, pin loading, design rules, and operating conditions.

The symbol library defines the symbols for schematic viewing of the design. You need this library if you intend to use the Design Vision GUI.

In addition, you must specify any specially licensed DesignWare libraries by using the `synthetic_library` command. (You do not need to specify the standard DesignWare library.)

See Chapter 4, “Working With Libraries.”

3. Read Design

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

Design Compiler provides the following ways to read design files:

- The `analyze` and `elaborate` commands
- The `read_file` command
- The `read_vhdl` and `read_verilog` commands. These commands are derived from the `read_file -format VHDL` and `read_file -format verilog` commands.

See Chapter 5, “Working With Designs in Memory.” For detailed information on the recommended reading methods, see the HDL Compiler documentation.

4. Define Design Environment

Design Compiler requires that you model the environment of the design to be synthesized. This model comprises the external operating conditions (manufacturing process, temperature, and voltage), loads, drives, fanouts, and wire load models. It directly

influences design synthesis and optimization results. You define the design environment by using the set commands listed under this step of Figure 2-2.

See Chapter 6, “Defining the Design Environment.”

5. Set Design Constraints

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

Optimization constraints define the design goals for timing (clocks, clock skews, input delays, and output delays) and area (maximum area). In the optimization process, Design Compiler attempts to meet these goals, but no design rules are violated by the process. You define these constraints by using commands such as those listed under this step in Figure 2-2. To optimize a design correctly, you must set realistic constraints.

Note:

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

See Chapter 7, “Defining Design Constraints.”

6. Select Compile Strategy

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

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

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

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

Note:

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

See Chapter 9, “Optimizing the Design.”

7. Optimize the Design

You use the `compile_ultra` command or `compile` command to invoke the Design Compiler synthesis and optimization processes. Several compile options are available with both commands. In particular, the `map_effort` option of the `compile` command can be set to medium, or high.

In a default compile, when you are performing design exploration, you use the medium `map_effort` option of the `compile` command. Because this option is the default, you do not need to specify `map_effort` in the `compile` command. In a final design implementation compile, you might want to set `map_effort` to high. You should use this option judiciously, however, because the resulting compile process is CPU intensive. Often setting `map_effort` to medium is sufficient.

For designs that have significantly tight timing constraints, you can invoke a single DC Ultra command, `compile_ultra`, for better quality of results (QoR). The command is a push-button solution for timing-critical, high performance designs and encapsulates DC Ultra strategies into a single command.

See Chapter 9, “Optimizing the Design.”

8. Analyze and Resolve Design Problems

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

See Chapter 11, “Analyzing and Resolving Design Problems.”

9. Save the Design Database

You use the `write` command to save the synthesized designs. Remember that Design Compiler does not automatically save designs before exiting.

You can also save in a script file the design attributes and constraints used during synthesis. Script files are ideal for managing your design attributes and constraints.

See the section “Exiting Design Compiler” on page 2-11 and see the chapter on using script files in the *Design Compiler Command-Line Interface Guide*.

A Design Compiler Session Example

Example 2-2 on page 2-27 shows a simple dctcl script that performs a top-down compile run. It uses the basic synthesis flow. The script contains comments that identify each of the steps in the flow. Some of the script command options and arguments have not yet been explained in this manual. Nevertheless, from the previous discussion

of the basic synthesis flow, you can begin to understand this example of a top-down compile. The remaining chapters will help you understand these commands in detail.

Note:

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

Example 2-2 Top-Down Compile Script

```
/* specify the libraries */
set target_library my_lib.db
set symbol_library my_lib.sdb
set link_library [list "*" $target_library]

/* read the design */
read_verilog Adder16.v

/* define the design environment */
set_operating_conditions WCCOM
set_wire_load_model "10x10"
set_load 2.2 sout
set_load 1.5 cout
set_driving_cell -lib_cell FD1 [all_inputs]
set_drive 0 clk

/* set the optimization constraints */
create_clock clk -period 10
set_input_delay -max 1.35 -clock clk {ain bin}
set_input_delay -max 3.5 -clock clk cin
set_output_delay -max 2.4 -clock clk cout
set_max_area 0

/* map and optimize the design */
compile

/* analyze and debug the design */
report_constraint -all_violators
report_area

/* save the design database */
write -format ddc -hierarchy -output Adder16.ddc
```

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

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

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

```
dc_shell-xg-t> source run.tcl
```

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

  For example, if the script is in a file called run.scr, you can invoke Design Compiler and execute the script file from the UNIX prompt by entering the following commands:

  ```
  % dc_shell-xg-t -f run.tcl
  ```

3

Preparing Design Files for Synthesis

Designs (that is, 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, however, 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 and data organization.

Controlling the Design Data

As new versions of your design are created, you must maintain some archival and record keeping method that provides a history of the design evolution and that lets you restart the design process if data is lost. Establishing controls for data creation, maintenance, overwriting, and deletion is a fundamental design management issue. Establishing file-naming conventions is one of the most important rules for data creation. Table 3-1 lists the recommended file name extensions for each design data type

Table 3-1 File Name Extensions

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

Organizing the Design Data

Establishing and adhering to a method of organizing data 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. The following figures show directory structures based on the top-down compile strategy (Figure 3-1) and the bottom-up compile strategy (Figure 3-2). For details about compile strategies, see “Selecting and Using a Compile Strategy” on page 9-6.

Figure 3-1 Top-Down Compile Directory Structure

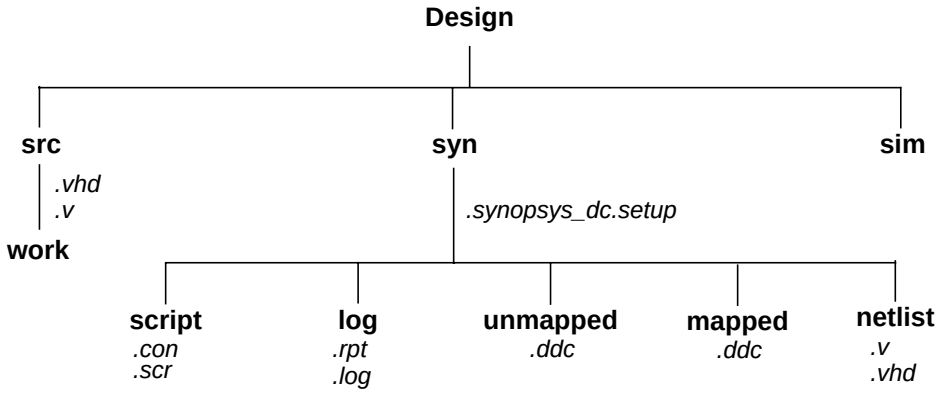
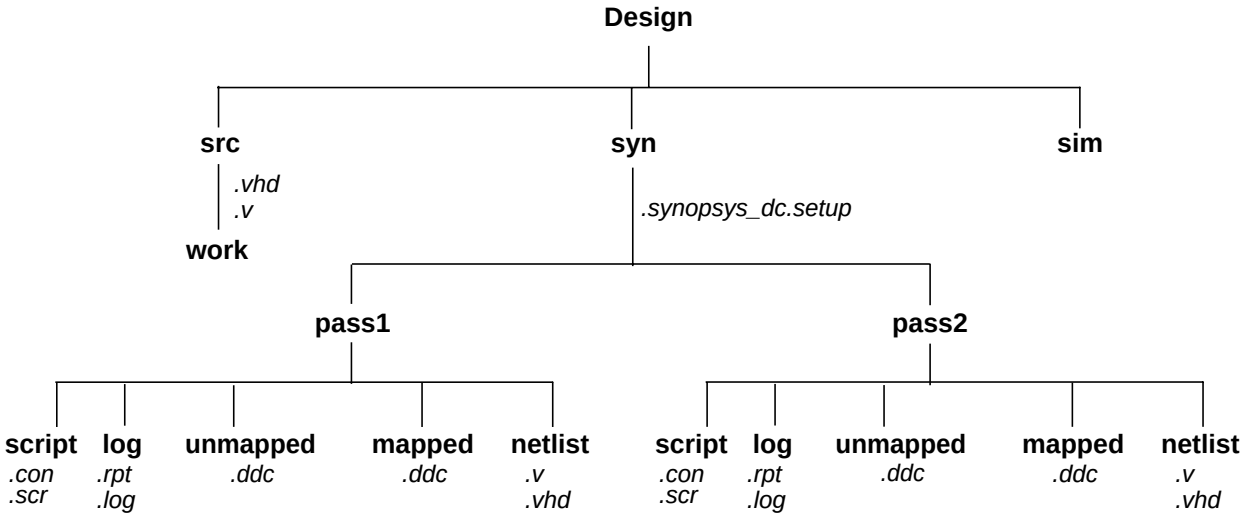


Figure 3-2 Bottom-Up Compile Directory Structure



Partitioning for Synthesis

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

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

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

- 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 effort.

When reusing existing designs, partition the design to enable instantiation of the designs.

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

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

Keeping Related Combinational Logic Together

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

For best results, apply these strategies:

- Group related combinational logic and its destination register together.

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

- Eliminate glue logic.

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

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

Figure 3-3 Poor Partitioning of Related Logic

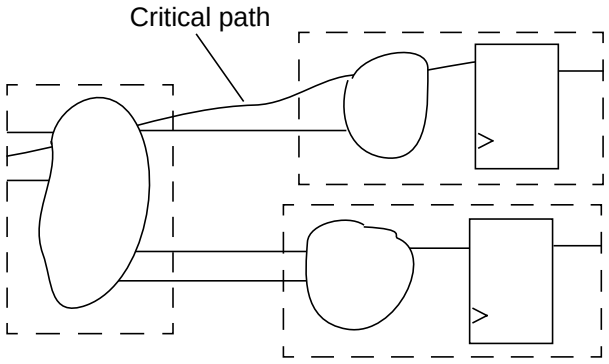
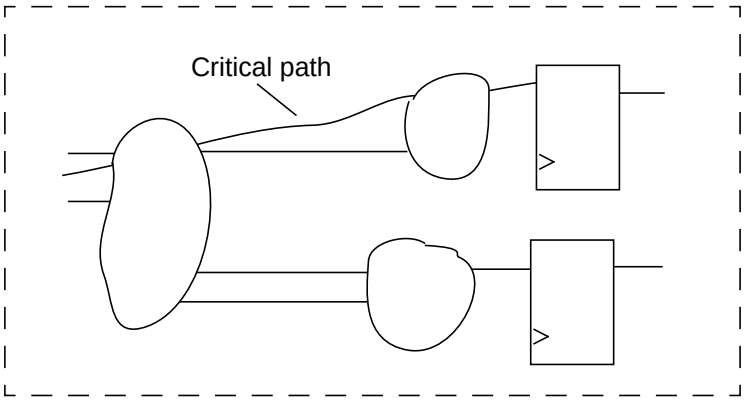


Figure 3-4 shows the same design with no artificial boundaries. In this design, Design Compiler has the flexibility to combine related functions in the combinational clouds.

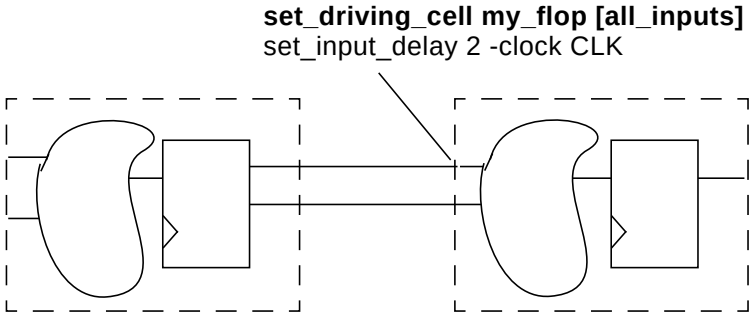
Figure 3-4 Keeping Related Logic in the Same Block



Registering Block Outputs

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

Figure 3-5 Registering All Outputs



This method enables you to constrain each block easily because

- The drive strength on the inputs to an individual block always equals the drive strength of the average input drive

- The input delays from the previous block always equal the path delay through the flip-flop

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

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

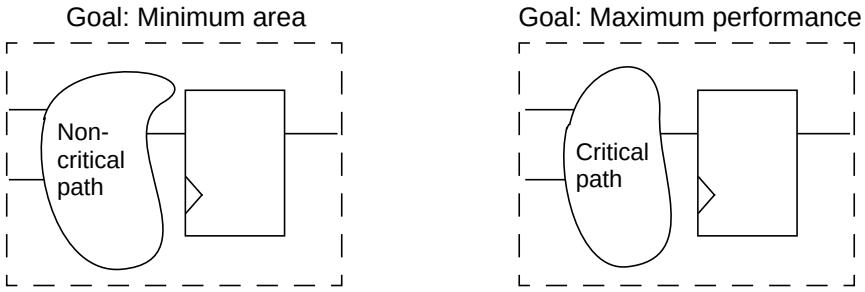
Partitioning by Design Goal

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

Figure 3-6 Blocks With Different Constraints



Partitioning by Compile Technique

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

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

For more information on structuring and flattening, see “Logic-Level Optimization” on page 9-3.

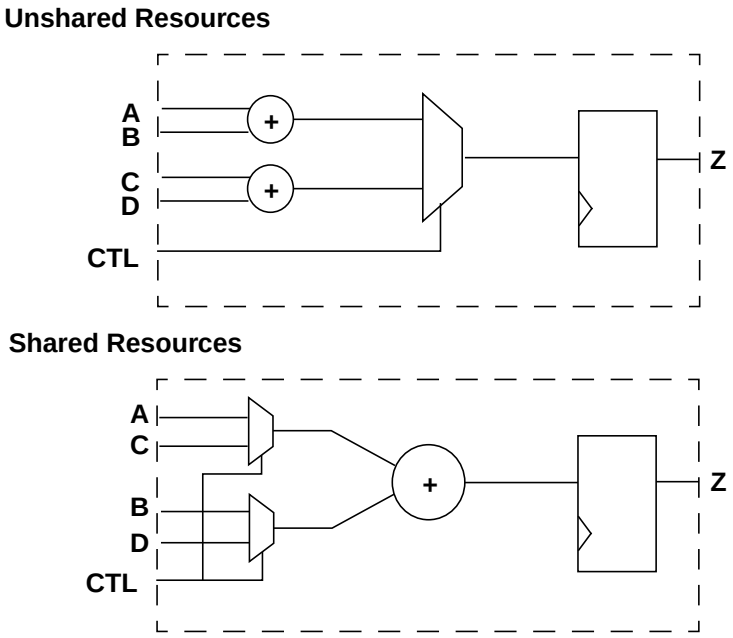
Keeping Sharable Resources Together

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

For example, if two separate adders have the same destination path and have multiplexed outputs to that path, keep the adders in one VHDL process or Verilog always block. This approach allows Design

Compiler to share resources (using one adder instead of two) if the constraints allow sharing. Figure 3-7 shows possible implementations of a logic example.

Figure 3-7 Keeping Sharable Resources in the Same Process



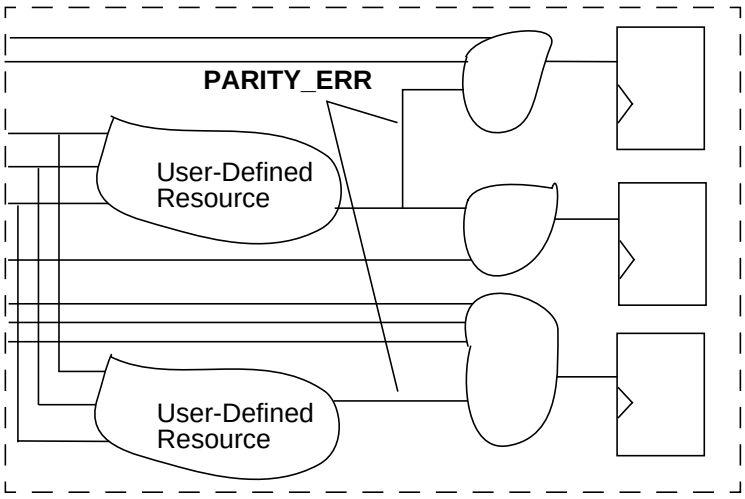
For more information about resource sharing, see the HDL Compiler documentation.

Keeping User-Defined Resources With the Logic They Drive

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

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

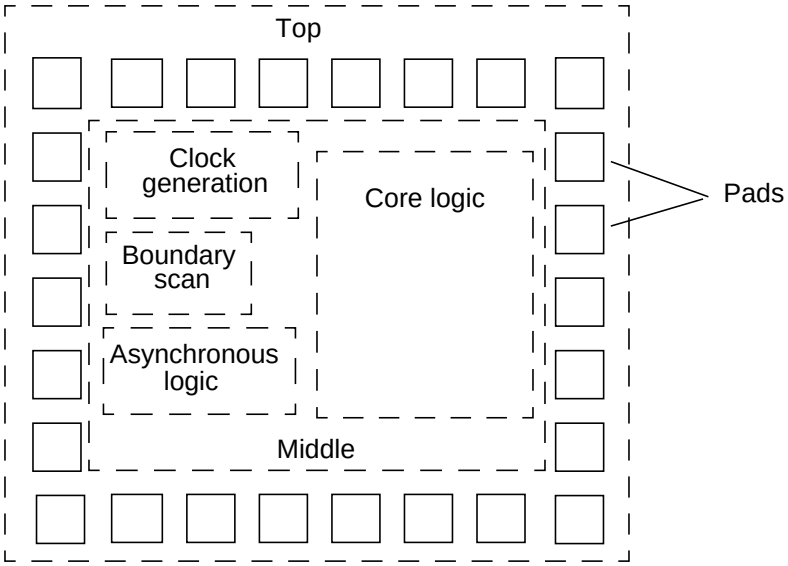
Figure 3-8 Duplicating User-Defined Resources



Isolating Special Functions

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

Figure 3-9 Recommended Top-Level Partitioning



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

HDL Coding for Synthesis

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

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

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 technology library contains at least a 2-to-1 multiplexer cell, Design Compiler maps the inferred MUX_OPs to multiplexer cells in the target technology library. Design Compiler determines the MUX_OP implementation during compile based on the design constraints. For information about how Design Compiler maps MUX_OPs to multiplexers, see the *Design Compiler Reference Manual: Optimization and Timing Analysis*.

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 technology library does not contain a falling-edge-triggered flip-flop, Design Compiler generates an inverter in the clock tree for the falling-edge clock.

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

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

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

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

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

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

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

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

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

Multibit components might not be efficient in the following instances:

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

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

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

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

Using Synthetic Libraries

To help you achieve optimal performance, Synopsys supplies a synthetic library. This library contains efficient implementations for adders, incrementers, comparators, and signed multipliers.

Design Compiler selects a synthetic component to meet the given constraints. After Design Compiler assigns the synthetic structure, you can always change to another type of structure by modifying your constraints. If you ungroup the synthetic cells or write the netlist to a text file, however, Design Compiler can no longer recognize the synthetic component and cannot perform implementation reselection.

The HDL Compiler documentation contains additional information about using compiler directives to control synthetic component use. The *DesignWare Foundation Library Databook* volumes contain additional information about synthetic libraries and provide examples of how to infer and instantiate synthetic components.

Example 3-1 and Example 3-2 use the label, ops, map_to_module, and implementation compiler directives to implement a 32-bit carry-lookahead adder.

Example 3-1 32-Bit Carry-Lookahead Adder (Verilog)

```
module add32 (a, b, cin, sum, cout);
   input [31:0] a, b;
   input cin;
   output [31:0] sum;
   output cout;
   reg [33:0] temp;

always @(a or b or cin)
   begin : add1
   /* synopsys resource r0:
         ops = "A1",
         map_to_module = "DW01_add",
         implementation = "cla"; */
   temp = ({1'b0, a, cin} + // synopsys label A1
           {1'b0, b, 1'b1});
   end

assign {cout, sum} = temp[33:1];

endmodule
```

Example 3-2 32-Bit Carry-Lookahead Adder (VHDL)

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

library synopsys;
use synopsys.attributes.all;

entity add32 is
   port (a,b : in std_logic_vector (31 downto 0);
         cin : in std_logic;
         sum : out std_logic_vector (31 downto 0);
         cout: out std_logic);
end add32;

architecture rtl of add32 is
   signal temp_signed : SIGNED (33 downto 0);
   signal op1, op2, temp : STD_LOGIC_VECTOR (33 downto 0);
   constant COUNT : UNSIGNED := "01";

begin
   infer: process ( a, b, cin )
      constant r0 : resource := 0;
      attribute ops of r0 : constant is "A1";
     attribute map_to_module of r0 : constant is "DW01_add";
      attribute implementation of r0 : constant is "cla";

   begin
        op1 <= '0' & a & cin;
        op2 <= '0' & b & '1';
       temp_signed <= SIGNED(op1) + SIGNED(op2); -- pragma
label A1
        temp <= STD_LOGIC_VECTOR(temp_signed);

        cout <= temp(33);
        sum <= temp(32 downto 1);
   end process infer;
 end rtl;
```

Designing State Machines

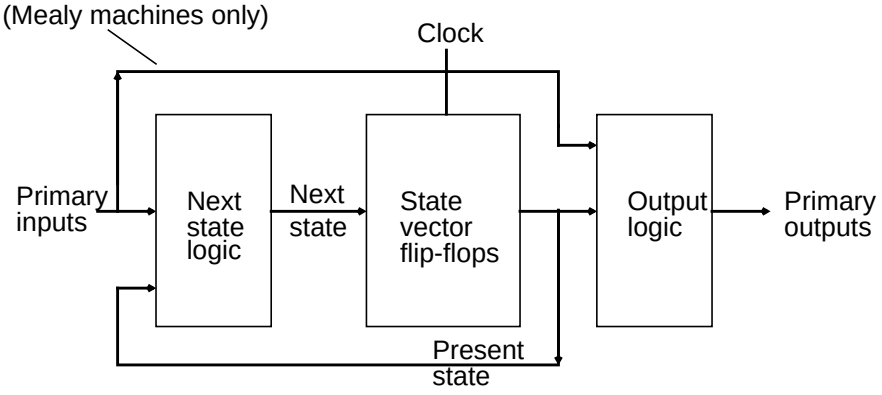
You can specify a state machine by using several different formats:

- Verilog
- VHDL
- State table
- PLA

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

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

Figure 3-10 Finite State Machine Architecture



Using an extracted state table provides the following benefits:

- State minimization can be performed.
- Tradeoffs between different encoding styles can be made.

- Don’t care conditions can be used without flattening the design.
- Don’t care state codes are automatically derived.

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

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 3-3 Incomplete Sensitivity List (Verilog)

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

Example 3-4 Incomplete Sensitivity List (VHDL)

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

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

For correct simulation results,

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

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

Example 3-5 Incorrect if Statement (Verilog)

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

Example 3-6 Incorrect if Statement (VHDL)

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

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

Example 3-7 Using the case Statement (Verilog)

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

Example 3-8 Using the case Statement (VHDL)

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

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

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

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

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

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

Example 3-9 shows a Verilog code fragment that includes a global `define statement and a local parameter. Example 3-10 shows a VHDL code fragment that includes a global constant and a local generic.

Example 3-9 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 foo
module foo (a, b, c);
   parameter WIDTH=128;
   input [WIDTH-1:0] a, b;
   output [WIDTH-1:0] c;
```

Example 3-10 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 foo
entity foo1 is
   port (a,b : in std_logic_vector(WIDTH-1 downto 0);
         c: out std_logic_vector(WIDTH-1 downto 0));
end foo;

-- Define and use local constant in entity foo
entity foo 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 foo;
```

Using Verilog Macro Definitions

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

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

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

- Do not use `define inside module definitions.

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

Using VHDL Port Definitions

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

- Use the STD_LOGIC and STD_LOGIC_VECTOR packages.

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

- Do not use the buffer port mode.

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

Writing Effective Code

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 readability of your HDL source code:

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

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

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

  Commonly used styles include C, Pascal, and Modula.

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

  - Pascal style capitalizes the first letter of the name and first letter of each word, for example, PacketAddr, DataIn, and FirstGrantEnable.

  - Modula style uses a lowercase letter for the first letter of the name and capitalizes the first letter of subsequent words, for example, packetAddr, dataIn, and firstGrantEnable.

    Choose one convention and apply it consistently.

- Avoid confusing characters.

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

- Avoid reserved words.

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

  Table 3-2 shows common suffixes and their meanings.

Table 3-2 Signal Name Suffixes and Their Meanings

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

Guidelines for Expressions

Observe the following guidelines for expressions:

- Use parentheses to indicate precedence.

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

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

- Replace repetitive expressions with function calls or continuous assignments.

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

Guidelines for Functions

Observe these guidelines for functions:

- Do not use global references within a function.

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

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

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

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

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

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

- Be careful when using component implication.

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

  The following functionality might require component instantiation or functional implication:

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

    This functionality includes asynchronous logic and asynchronous signals that are valid during certain states.
  - Data-path circuitry

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

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

Guidelines for Modules

Observe these guidelines for modules:

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

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

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

4

Working With Libraries

This chapter presents basic library information. Design Compiler uses technology, symbol, and synthetic or DesignWare libraries to implement synthesis and to display synthesis results graphically. You must know how to carry out a few simple library commands so that Design Compiler uses the library data correctly.

This chapter contains the following sections:

Selecting a Semiconductor Vendor

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

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

Understanding the Library Requirements

Design Compiler uses these libraries:

- Technology libraries
- Symbol libraries
- DesignWare libraries

This section describes these libraries.

Technology Libraries

Technology libraries contain information about the characteristics and functions of each cell provided in a semiconductor vendor's library. Semiconductor vendors maintain and distribute the technology libraries.

Cell characteristics include information such as cell names, pin names, area, delay arcs, and pin loading. The technology library also defines the conditions that must be met for a functional design (for example, the maximum transition time for nets). These conditions are called design rule constraints.

In addition to cell information and design rule constraints, technology libraries specify the operating conditions and wire load models specific to that technology.

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

Design Compiler uses technology libraries for these purposes:

- Implementing the design function

  The technology libraries that Design Compiler maps to during optimization are called target libraries. The 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. Design Compiler saves this information in the design's `local_link_library` attribute. For information about attributes, see "Working With Attributes" on page 5-55.

- Resolving cell references

  The technology libraries that Design Compiler uses to resolve cell references are called link libraries.

  In addition to technology libraries, link libraries can also include design files. The link libraries 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).

  For more information about resolving references, see "Linking Designs" on page 5-17.

- Calculating timing values and path delays

  The 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 Reference Manual*.

Symbol Libraries

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

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

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

DesignWare Libraries

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

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

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

Specifying Libraries

You use dc_shell variables to specify the libraries used by Design Compiler. Table 4-1 lists the variables for each library type as well as the typical file extension for the library.

Table 4-1 Library Variables

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

Specifying Technology Libraries

To specify technology libraries, you must specify the target library and link library.

Target Library

Design Compiler uses the target library to build a circuit. During mapping, Design Compiler selects functionally correct gates from the target library. It also calculates the timing of the circuit, using the vendor-supplied timing data for these gates.

Use the `target_library` variable to specify the target library.

The syntax is

```
set target_library my_tech.db
```

Link Library

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

During the linking process, Design Compiler uses the `link_library` system variable, the `local_link_library` attribute, and the `search_path` system variable to resolve references. These variables and attribute are described below:

- `link_library` variable

  The `link_library` variable specifies a list of libraries and design files that Design Compiler can use to resolve references. When you load a design into memory, Design Compiler also loads all libraries specified in the `link_library` variable.

Because the tool loads the libraries while loading the design, rather than during the link process, the memory usage and runtime required for loading the design might increase. However, the advantage is that you know immediately whether your design can be processed with the available memory.

An asterisk in the value of the `link_library` variable specifies that Design Compiler should search memory for the reference.

- `local_link_library` attribute

  The `local_link_library` attribute is a list of design files and libraries added to the beginning of the `link_library` variable during the linking process. Design Compiler searches files in the `local_link_library` attribute first when it resolves references.

- `search_path` variable

  If Design Compiler does not find the reference in the link libraries, it searches in the directories specified by the `search_path` variable, described in “Specifying a Library Search Path” on page 4-10. For more information on resolving references, see “Linking Designs” on page 5-17.

The syntax is

```
set link_library {* my_tech.db}
```

Note that you specify the same value for the target library and the link library (except when you are performing technology translation).

When you specify the files in the `link_library` variable, consider that Design Compiler searches these files from left to right when it resolves references, and it stops searching when it finds a reference. If you specify the link library as {"*" lsi_10k.db}, the designs in memory are searched before the lsi_10k library.

Design Compiler uses the first technology library found in the `link_library` variable as the main library. It uses the main library to obtain default values and settings used in the absence of explicit specifications for operating conditions, wire load selection group, wire load mode, and net delay calculation. Design Compiler obtains the following default values and settings from the main library:

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

If other libraries have units different from the main library units, Design Compiler converts all units to those that the main library uses.

Specifying DesignWare Libraries

You do not need to specify the standard synthetic library, standard.sldb, that implements the built-in HDL operators. The software automatically uses this library.

If you are using additional DesignWare libraries, you must specify these libraries by using the `synthetic_library` variable (for optimization purposes) and the `link_library` variable (for cell resolution purposes).

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

Specifying a Library Search Path

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

For example, assume that you have technology libraries named my_lib.db in both the lib directory and the vhdl directory. If the search path contains (in order) the lib directory, the vhdl directory, and the default search path, Design Compiler uses the my_lib.db file found in the lib directory, because it encounters the lib directory first.

You can use the `which` command to see which library files Design Compiler finds (in order).

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

Loading Libraries

Design Compiler uses binary libraries (.db format for technology libraries and .sdb format for symbol libraries) and automatically loads these libraries when needed.

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.

To manually load a binary library, use the `read_file` command.

```
dc_shell-xg-t> read_file my_lib.db
dc_shell-xg-t> read_file my_lib.sdb
```

Listing Libraries

Design Compiler refers to a library loaded in memory by its name. The library statement in the library source defines the library name.

To list the names of the libraries loaded in memory, use the `list_libs` command.

```
dc_shell-xg-t> 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

Use the `report_lib` command to report the contents of a library. The `report_lib` command can report the following data:

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

Specifying Library Objects

Library objects are the vendor-specific cells and their pins.

The Design Compiler naming convention for library objects is

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

file

The file name of a technology library followed by a colon (:). If you have multiple libraries loaded in memory with the same name, you must specify the file name.

library

The name of a library in memory, followed by a slash (/).

cell

The name of a library cell.

pin

The name of a cell's pin.

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

```
dc_shell-xg-t> set_dont_use my_lib/AND4
1
```

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

```
dc_shell-xg-t> set_disable_timing [get_pins my_lib/AND4/Z]
1
```

Directing Library Cell Usage

When Design Compiler maps a design to a technology library, it selects components (library cells) from that library. You can influence the choice of components (library cells) by

- Excluding cells from the target library
- Specifying cell preferences

Excluding Cells From the Target Library

Use the set_dont_use command to exclude cells from the target library. Design Compiler does not use these excluded cells during optimization.

This command affects only the copy of the library that is currently loaded into 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 disabled.

For example, to prevent Design Compiler from using the high-drive inverter INV_HD, enter

```
dc_shell-xg-t> set_dont_use MY_LIB/INV_HD
1
```

Use the `remove_attribute` command to reinclude excluded cells in the target library.

```
dc_shell-xg-t> remove_attribute MY_LIB/INV_HD dont_use
MY_LIB/INV_HD
```

Specifying Cell Preferences

Use the `set_prefer` command to indicate preferred cells. You can issue this command with or without the `-min` option.

Use the command without the `-min` option if you want Design Compiler to prefer certain cells during the initial mapping of the design.

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

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

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

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

```
dc_shell-xg-t> set_prefer MY_LIB/INV_LD
1
```

Use the `remove_attribute` command to remove cell preferences.

```
dc_shell-xg-t> remove_attribute MY_LIB/INV_LD preferred
MY_LIB/INV_LD
```

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

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

```
dc_shell-xg-t> set_prefer -min class/IV
1
```

Use the `remove_attribute` command to remove the `hold_preferred` cell attribute.

```
dc_shell-xg-t> remove_attribute class/IV hold_preferred
class/IV
```

Library-Aware Mapping and Synthesis

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

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

Generating the ALIB file

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

```
dc_shell-xg-t> set target_library "x.db"
dc_shell-xg-t> set link_library "x.db"
dc_shell-xg-t> set alib_library_analysis_path \
                  "/remote/libraries/alib"
dc_shell-xg-t> alib_analyze_libs
```

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

Using the ALIB library

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

```
dc_shell-xg-t> set alib_library_analysis_path \
                  "remote/libraries/alib"
```

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

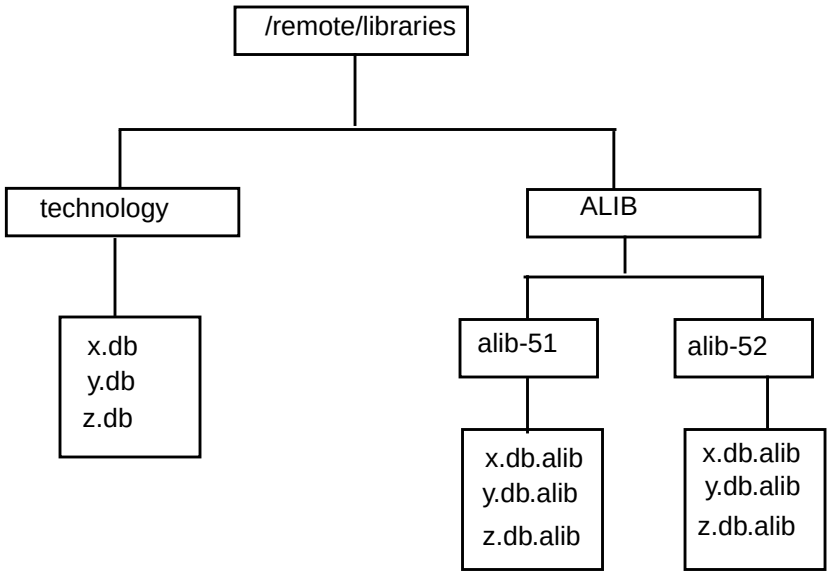
It is recommended that you generate the ALIB library for your target technology library when you install Design Compiler. Create a directory structure similar to the one you use for storing technology libraries as shown in Figure 4-1. If you re-install Design Compiler, you must regenerate the ALIB library. A new subdirectory is created within the /remote/libraries/alibs directory.

Note:

In version Y-2006.06-SP4 and later, Design Compiler reads ALIB files in the alib-52 directory. Design Compiler automatically generates the new ALIB files when you run the Y-2006.06-SP4 version for the first time.

If you are using the pre-built ALIB file in a central location, create new ALIB files before compiling.

Figure 4-1 Directory Structure for ALIB library



Removing Libraries From Memory

The `remove_design` command removes libraries from dc_shell memory. If you have multiple libraries with the same name loaded into memory, you must specify the path as well as 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.

5

Working With Designs in Memory

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

This chapter contains the following sections:

- Creating Designs
- Copying Designs
- Renaming Designs
- Changing the Design Hierarchy
- Editing Designs
- Translating Designs From One Technology to Another
- Removing Designs From Memory
- Saving Designs
- Working With Attributes

Design Terminology

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

About Designs

Designs are circuit descriptions that perform logical functions. Designs are described in various design formats, such as VHDL or Verilog HDL.

Logic-level designs are represented as sets of Boolean equations. Gate-level designs, such as netlists, are represented as interconnected cells.

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

Flat Designs

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

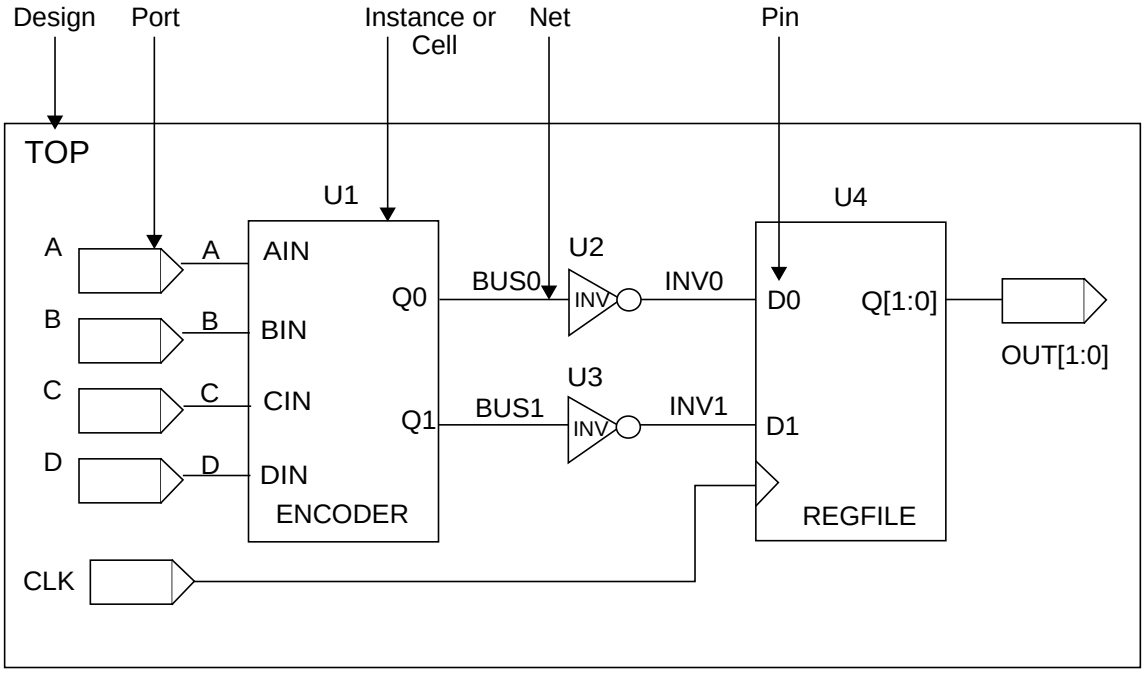
Hierarchical Designs

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

Design Objects

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

Figure 5-1 Design Objects



Design: {TOP, ENCODER, REGFILE}

Reference: {ENCODER, REGFILE, INV}

Instance: {U1, U2, U3, U4}

Design

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

Reference

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

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

Instance or Cell

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

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

Ports

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

Pins

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

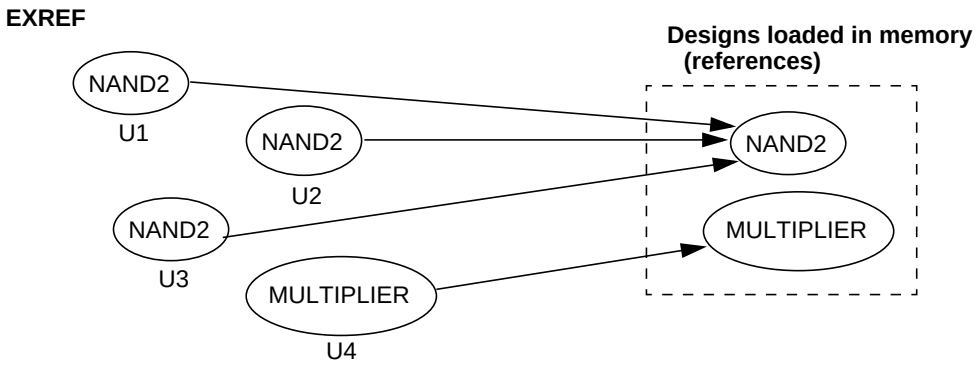
Nets

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

Relationship Between Designs, Instances, and References

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

Figure 5-2 Instances and References



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

For information about resolving references, see “Linking Designs” on page 5-17.

Reporting References

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

Using Reference Objects

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

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

```
dc_shell-xg-t> get_references AN2
{U2 U3 U4}
```

To see the reference names, use the following command:

```
dc_shell-xg-t> report_cell [get_references AN*]
```

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

Reading Designs

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

Table 5-1 Supported Input Formats

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

Commands for Reading Design Files

Design Compiler provides the following ways to read design files:

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

Using the analyze and elaborate Commands

The `analyze` command does the following:

- Reads an HDL source file
- Checks it for errors (without building generic logic for the design)

- Creates HDL library objects in an HDL-independent intermediate format
- Stores the intermediate files in a location you define

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

Use options to the `analyze` command as follows:

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

The `elaborate` command does the following:

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

Use options to the `elaborate` command as follows:

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

For more information about the `analyze` and `elaborate` commands, see the man pages and *HDL Compiler (Presto Verilog) Reference Manual* or the *HDL Compiler (Presto VHDL) Reference Manual*.

Using the read_file Command

The `read_file` command does the following:

- Reads several different formats
- Performs the same operations as `analyze` and `elaborate` in a single step
- Creates .mr and .st intermediate files for VHDL
- Does not execute the `link` command automatically (see “Linking Designs” on page 5-17)

- Does not create any intermediate files for Verilog (However, you can have the `read_file` command create intermediate files by setting the `hdlin_auto_save_templates` variable to true)

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

Use options to the `read_file` command as follows:

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

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

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

Table 5-2 read_file Versus analyze and elaborate Commands

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

Reading HDL Designs

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

- The `analyze` and `elaborate` commands

  Follow these steps:

  1. Analyze the top-level design and all subdesigns in bottom-up order (to satisfy any dependencies).
  2. Elaborate the top-level design and any subdesigns that require parameters to be assigned or overwritten.

For example, enter

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

- The `read_file` command

  For example, enter

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

- The `read_verilog` or `read_vhdl` command

  For example, enter

  ```
  dc_shell-xg-t> read_verilog RISC_CORE.v
  ```

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

Reading .ddc Files

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

```
dc_shell-xg-t> read_ddc design_file.ddc
```

Note:

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

Reading .db Files

Although you can use the .db format in XG mode, it is not recommended. To maximize the capacity and performance improvements offered in XG mode, use the .ddc format rather than the .db format.

To read in a .db file, use the `read_db` command or the `read_file -format db` command. For example,

```
dc_shell-xg-t> read_db design_file.db
```

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

Listing Designs in Memory

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

```
dc_shell-xg-t> list_designs
A (*)    B    C
1
```

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

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

```
dc_shell-xg-t> list_designs -show_file

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

/home/designer/dc/B.ddc
B       C
1
```

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

Setting the Current Design

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

- With the `read_file` command

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

  ```
  dc_shell-xg-t> read_file -format ddc MY_DESIGN.ddc
  Reading ddc file '/designs/ex/MY_DESIGN.ddc'
  Current design is 'MY_DESIGN'
  ```

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

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

  ```
  dc_shell-xg-t> current_design ANY_DESIGN
  Current design is 'ANY_DESIGN'.
  {ANY_DESIGN}
  ```

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

```
dc_shell-xg-t> printvar current_design
current_design = "test"
```

Using the current_design Command

It is recommended that you avoid writing scripts that use a large number of `current_design` commands, such as in a loop. Using a large number of `current_design` commands can increase runtime. For more information, see the *Design Compiler Command-Line Interface Guide*, chapter 5.

Additionally, several commands accept instance objects—that is, cells at a lower level of hierarchy. You can operate on hierarchical designs from any level in the design without using the `current_design` command. The enhanced commands are listed below:

- Netlist editing commands.

  For more information, see “Editing Designs” on page 5-43.

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

  For more information, see “Removing Levels of Hierarchy” on page 5-34 and “Uniquify Method” on page 9-19.

- The `set_size_only` command

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

- The `change_link` command

For more information, see “Changing Design References” on page 5-20.

Linking Designs

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

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

Design Compiler resolves references by carrying out the following steps:

1. It determines which library components and subdesigns are referenced in the current design and its hierarchy.
2. It searches the link libraries to locate these references.
   a. Design Compiler first searches the libraries and design files defined in the current design’s `local_link_library` attribute
   b. If an asterisk is specified in the value of the `link_library` variable, Design Compiler searches in memory for the reference.
   c. Design Compiler then searches the libraries and design files defined in the `link_library` variable.
3. If it does not find the reference in the link libraries, it searches in the directories specified by the `search_path` variable. See “Locating Designs by Using a Search Path” on page 5-19.

4. It links (connects) the located references to the design.

Note:

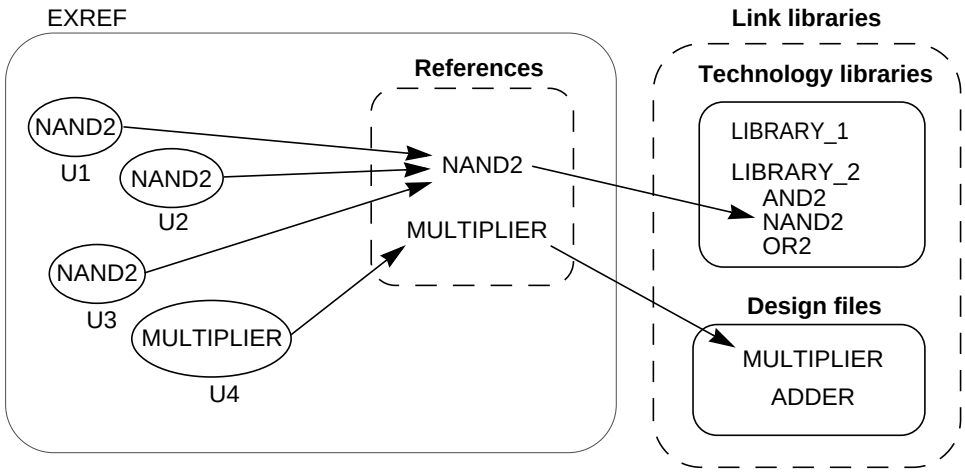
In a hierarchical design, Design Compiler considers only the top-level design's local link library. It ignores local link libraries associated with the subdesigns.

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

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

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

Figure 5-3 Resolving References



Locating Designs by Using a Search Path

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

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

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

```
dc_shell-xg-t> lappend search_path project
```

Changing Design References

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

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

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

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

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

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

Example 1

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

```
dc_shell-xg-t> copy_design ADDER MY_ADDER
dc_shell-xg-t> change_link {U1 U2} MY_ADDER
```

Example 2

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

```
dc_shell-xg-t> change_link top/sub_inst/U1 lsi_10k/AN3
```

Example 3

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

```
dc_shell-xg-t> change_link -all_instances \
    mid1/bot1/inv1 lsi_10k/AN3
Information: Changed link for all instances of cell 'inv1'
in subdesign 'bot'. (UID-193)

dc_shell-xg-t> get_cells -hierarchical \
         -filter "ref_name == AN3"
{mid1/bot1/inv1 mid1/bot2/inv1}
1
```

Listing Design Objects

Design Compiler provides commands for accessing various design objects. These commands refer to design objects located in the current design. Each command in Table 5-3 performs one of the following actions:

- List

  Provides a listing with minimal information.

- Display

  Provides a report that includes characteristics of the design object.

- Return

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

Table 5-3 lists the commands and the actions they perform.

Table 5-3 Commands to Access Design Objects

Object	Command	Action
Instance	`list_instances` `report_cell`	Lists instances and their references. Displays information about instances.
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.

Table 5-3 Commands to Access Design Objects (Continued)

Object	Command	Action
Clock	`report_clock` `all_clocks`	Displays information about clocks. Returns all clocks.
Register	`all_registers`	Returns all registers.

Note:

You can also use the `get_*` commands to create and list collections of cells, designs, libraries, library cells, library cell pins, nets, pins, and ports.

Specifying Design Objects

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

Using a Relative Path

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

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

```
dc_shell-xg-t> current_design Count_16
Current design is 'Count_16'.
{Count_16}
dc_shell-xg-t> set_dont_touch U1/U15
```

or

```
dc_shell-xg-t> current_design Count_16
Current design is 'Count_16'.
{Count_16}
dc_shell-xg-t> current_instance U1
Current instance is '/Count_16/U1'.
/Count_16/U1
dc_shell-xg-t> set_dont_touch U15
1
```

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

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

```
dc_shell-xg-t> current_instance
Current instance is the top-level of the design 'Count_16'
```

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

```
dc_shell-xg-t> printvar current_instance
current_instance = "Count_16/U1"
```

Using an Absolute Path

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

[*file*:]*design/object*

file

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

design

The name of a design in dc_shell memory.

object

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

To specify an object type, use the `get_*` command. For more information about these commands, see the *Design Compiler Command-Line Interface Guide.*

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

```
dc_shell-xg-t> set_dont_touch \
   /usr/designs/Count_16.ddc:Count_16/U1/U5
1
```

Creating Designs

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

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

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

/work_dir/my_design.db
my_design
1
```

Designs created with `create_design` contain no design objects. Use the appropriate create commands (such as `create_clock`, `create_cell`, or `create_port`) to add design objects to the new design. For information about these commands, see “Editing Designs” on page 5-43.

It is recommended that in XG mode, you store design data in the .ddc format rather than the .db format. After you have added design objects to your design, save your design in the .ddc format by using the `write -format ddc` command. See “Saving Designs in .ddc Format” on page 5-50.

Copying Designs

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

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

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

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

Enter the following commands to create unique instances:

```
dc_shell-xg-t> copy_design COMP COMP1
Information: Copying design /designs/COMP.ddc:COMP to
             designs/COMP.ddc:COMP1
1
dc_shell-xg-t> change_link U1 COMP1
Performing change_link on cell 'U1'.
1
dc_shell-xg-t> copy_design COMP COMP2
Information: Copying design /designs/COMP.ddc:COMP to
             designs/COMP.ddc:COMP2
1
dc_shell-xg-t> change_link U2 COMP2
Performing change_link on cell 'U2'.
1
```

Renaming Designs

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

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

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

```
dc_shell-xg-t> list_designs -show_file

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

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

dc_shell-xg-t> list_designs -show_file
```

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

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

```
dc_shell-xg-t> get_cells -hier \
               -filter "ref_name == D"
{b_in_a/c_in_b/d1_in_c b_in_a/c_in_b/d2_in_c}

dc_shell-xg-t> rename_design D -prefix NEW_ -update_links
Information: Renaming design /test_dir/D.ddc:D to
                        /test_dir/D.ddc:NEW_D. (UIMG-45)

dc_shell-xg-t> get_cells -h -filter "ref_name == D"
# no such cells!

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

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

Changing the Design Hierarchy

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

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

Design Compiler provides the following hierarchy manipulation capabilities:

- Adding levels of hierarchy
- Removing levels of hierarchy
- Merging cells from different subdesigns

The following sections describe these capabilities.

Adding Levels of Hierarchy

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

Grouping Cells Into Subdesigns

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

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

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

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

Note:

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

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

Example 1

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

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

Example 2

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

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

Example 3

In the following example, three cells— bot1, foo1, and j —are grouped into a new subdesign named SAMPLE, with an instance name U1. The cells are at a lower level in the hierarchy and at the same hierarchical level; the parent design is unique.

```
dc_shell-xg-t> group {mid1/bot1 mid1/foo1 mid1/j}\
               -cell_name U1 -design_name SAMPLE
```

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

```
dc_shell-xg-t> current_design mid
dc_shell-xg-t> group {bot1 foo1 j} -cell_name U1 \
               -design_name SAMPLE
```

Grouping Related Components Into Subdesigns

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

To do this	Use this
Specify one of the following component types:	
Bused gates	`-hdl_bussed`
Combinational logic	`-logic`

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

Note:

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

Example 1

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

```
dc_shell-xg-t> group -hdl_block ftj/bar -design_name \
                    new_block
```

Example 2

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

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

Removing Levels of Hierarchy

Design Compiler does not optimize across hierarchical boundaries; therefore, you might want to remove the hierarchy within certain designs. By doing so, you might be able to improve timing results.

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

Note:

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

Ungrouping Hierarchies Before Optimization

You use the `ungroup` command to ungroup one or more designs before optimization.

Use the following arguments and options to the `ungroup` command:

To do this	Use this
Specify a list of cells to be ungrouped When the parent design is unique, the list can include cells from a lower level in the hierarchy (that is, the `ungroup` command can accept instance objects)	Provide a list of cells as an argument to the `ungroup` command
Ungroup all cells in the current design or current instance	`-all`

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

Note:

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

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

Example 1

To ungroup a list of cells, enter

```
dc_shell-xg-t> ungroup {high_decoder_cell low_decoder_cell}
```

Example 2

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

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

Example 3

To completely collapse the hierarchy of the current design, enter

```
dc_shell-xg-t> ungroup -all -flatten
```

Example 4

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

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

Example 5

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

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

Example 6

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

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

Example 7

This example illustrates how the `ungroup` command can accept instance objects (cells at a lower level of hierarchy) when the parent design is unique. In the example, MID1/BOT1 is a unique instantiation of design BOT. The command ungroups the cells MID1/BOT1/FOO1 and MID1/BOT1/FOO2.

```
dc_shell-xg-t> ungroup {MID1/BOT1/FOO1 MID1/BOT1/FOO2}
```

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

```
dc_shell-xg-t> current_instance MID1/BOT1
dc_shell-xg-t> ungroup {FOO1 FOO2}
```

Ungrouping Hierarchies During Optimization

You can ungroup designs during optimization either explicitly or automatically.

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

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

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

```
dc_shell-xg-t> set_ungroup U1
dc_shell-xg-t> compile
```

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

```
dc_shell-xg-t> get_attribute object ungroup
```

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

```
dc_shell-xg-t> set_ungroup object false
```

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

```
dc_shell-xg-t> compile -ungroup_all
```

Ungrouping Hierarchies Automatically During Optimization. To automatically ungroup hierarchies during optimization, you can use the `compile_ultra` command or the `-auto_ungroup` option of the `compile` command. Design Compiler provides two options to automatically ungroup hierarchies: area-based auto-ungrouping and delay-based auto-ungrouping.

By default, the `compile_ultra` command performs delay-based auto-ungrouping. It 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. The tool estimates the area for unmapped hierarchies and removes small subdesigns; the goal is to improve area and timing quality of results.

To use the auto-ungrouping capability of the `compile` command, enter

```
compile -auto_ungroup area|delay
```

You can use only one argument at a time: either the `area` argument for area-based auto-ungrouping or the `delay` argument for delay-based auto-ungrouping.

Before ungrouping begins, the tool issues a message to indicate that the specified hierarchy is being ungrouped.

After auto-ungrouping, use the `report_auto_ungroup` command to get a report on the hierarchies that were ungrouped during cell-count-based auto-ungrouping or delay-based auto-ungrouping. This report gives instance names, cell names, and the number of instances for each ungrouped hierarchy. For more information on automatic ungrouping, see the *Design Compiler Reference Manual: Optimization and Timing Analysis.*

Preserving Hierarchical Pin Timing Constraints During Ungrouping

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

placed on hierarchical pins in different ways. The table below summarizes the effect that ungrouping has on timing constraints within different compile flows.

Table 5-4 Preserving Hierarchical Pin Timing Constraints

Compile flow	Effect on hierarchical pin timing constraints
Ungrouping a hierarchy before optimization by using `ungroup`	Timing constraints placed on hierarchical pins are preserved. In previous releases, timing attributes placed on the hierarchical pins of a cell were not preserved when that cell was ungrouped. If you want your current optimization results to be compatible with previous results, set the `ungroup_preserve_constraints` variable to false. The default for this variable is true, which specifies that timing constraints will be preserved.
Ungrouping a hierarchy during optimization by using `compile -ungroup_all` or `set_ungroup` followed by `compile`	Timing constraints placed on hierarchical pins are not preserved. To preserve timing constraints, set the `auto_ungroup_preserve_constraints` variable to true.
Automatically ungrouping a hierarchy during optimization, that is, by using the `compile_ultra` command or `compile -auto_ungroup area|delay`	Design Compiler does not ungroup the hierarchy. To make Design Compiler ungroup the hierarchy and preserve timing constraints, set the `auto_ungroup_preserve_constraints` variable to true.

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

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

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

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

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

- `set_disable_timing`
- `set_case_analysis`
- `create_clock`
- `create_generated_clock`
- `set_propagated_clock`
- `set_clock_latency`

Note:

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

Merging Cells From Different Subdesigns

To merge cells from different subdesigns into a new subdesign,

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

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

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

Editing Designs

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

Table 5-5 Design Editing Tasks and Commands

Object	Task	Command
Cells	Create a cell	`create_cell`
	Delete a cell	`remove_cell`
Nets	Create a net	`create_net`
	Connect a net	`connect_net`
	Disconnect a net	`disconnect_net`
	Delete a net	`remove_net`
Ports	Create a port	`create_port`
	Delete a port	`remove_port`
		`remove_unconnected_port`
Pins	Connect pins	`connect_pin`
Buses	Create a bus	`create_bus`
	Delete a bus	`remove_bus`

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

```
dc_shell-xg-t> create_cell mid1/foo my_lib/AND2
```

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

```
dc_shell-xg-t> get_pins U8/*
{"U8/A", "U8/Z"}
dc_shell-xg-t> all_connected U8/A
{"n66"}
dc_shell-xg-t> all_connected U8/Z
{"OUTBUS[10]"}
dc_shell-xg-t> remove_cell U8
Removing cell 'U8' in design 'top'.
1
dc_shell-xg-t> create_cell U8 IVP
Creating cell 'U8' in design 'top'.
1
dc_shell-xg-t> connect_net n66 [get_pins U8/A]
Connecting net 'n66' to pin 'U8/A'.
1
dc_shell-xg-t> connect_net OUTBUS[10] [get_pins U8/Z]
Connecting net 'OUTBUS[10]' to pin 'U8/Z'.
1
```

Note:

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

```
dc_shell-xg-t> change_link U8 IVP
```

Additional netlist editing commands similar to IC Compiler are available. These commands are described below:

- Resizing a cell

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

- Inserting buffers or inverter pairs

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

- Inserting repeaters

  The `-no_of_cells` option of the `insert_buffer` command allows you to select a driver of a two-pin net and insert a chain of single-fanout buffers in the net driven by this driver.

- Removing buffers

  You can use the `remove_buffer` command to remove buffers.

Translating Designs From One Technology to Another

You use the `translate` command to translate a design from one technology to another.

Designs are translated cell by cell from the original technology library to a new technology library, preserving the gate structure of the original design. The translator uses the functional description of

each existing cell (component) to determine the matching component in the new technology library (target library). If no exact replacement exists for a component, it is remapped with components from the target library.

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

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

Procedure to Translate Designs

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

To translate a design,

1. Read in your mapped design.

   ```
   dc_shell-xg-t> read_file design.ddc
   ```

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

   ```
   dc_shell-xg-t> set target_library target_lib.db
   ```

3. Invoke the `translate` command.

   ```
   dc_shell-xg-t> translate
   ```

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

Restrictions on Translating Between Technologies

Keep the following restrictions in mind when you translate a design from one technology to another:

- The `translate` command translates functionality logically but does not preserve drive strength during translation. It always uses the lowest drive strength version of a cell, which might produce a netlist with violations.
- When you translate CMOS three-state cells into FPGA, functional equivalents between the technologies might not exist.
- Buses driven by CMOS three-state components must be fully decoded (Design Compiler can assume that only one bus driver is ever active). If this is the case, bus drivers are translated into control logic. To enable this feature, set the `compile_assume_fully_decoded_three_state_buses` variable to true before translating.
- If a three-state bus within a design is connected to one or more output ports, translating the bus to a multiplexed signal changes the port functionality. Because `translate` does not change port functionality, this case is reported as a translation error.

Removing Designs From Memory

The `remove_design` command removes designs from dc_shell memory. For example, after completing a compilation session and saving the optimized design, you can use `remove_design` to delete the design from memory before reading in another design.

By default, the `remove_design` command removes only the specified design. To remove its subdesigns, specify the `-hierarchy` option. To remove all designs (and libraries) from memory, specify the `-all` option.

If you defined variables that reference design objects, Design Compiler removes these references when you remove the design from memory. This prevents future commands from attempting to operate on nonexistent design objects. For example,

```
dc_shell-xg-t> set PORTS [all_inputs]
{"A0", "A1", "A2", "A3"}
dc_shell-xg-t> query_objects $PORTS
PORTS = {"A0", "A1", "A2", "A3"}
dc_shell-xg-t> remove_design
Removing design 'top'
1
dc_shell-xg-t> query_objects $PORTS
Error: No such collection '_sel2' (SEL-001)
```

Saving Designs

You can save (write to disk) the designs and subdesigns of the design hierarchy at any time, using different names or formats. After a design is modified, you should manually save it. Design Compiler does not automatically save designs before it exits.

Table 5-6 lists the design file formats supported by Design Compiler.

Table 5-6 Supported Output Formats

Format	Description
.ddc	Synopsys internal database format
Milkyway	Format for writing a Milkyway database within Design Compiler
Verilog	IEEE Standard Verilog (see the HDL Compiler documentation)
VHDL	IEEE Standard VHDL (see the HDL Compiler documentation)

Commands to Save Design Files

Design Compiler provides the following ways to save design files:

- The `write` command
- The `write_milkyway` command

Using the write Command

You use the `write` command to convert designs in memory to a format you specify and save that representation to disk.

Use options to the `write` command as shown:

To do this	Use this
Specify a list of designs to save. The default is the current design	`-design_list`

To do this	Use this
Specify the format in which a design is saved.	`-format` You can specify any of the output formats listed in Table 5-6 (except the Milkyway format; use the `write_milkyway` command instead))
Specify that all designs in the hierarchy are saved	`-hierarchy`
Specify a single file into which designs are wriiten	`-output`
Specify that only modified designs are saved	`-modified`
Specify the name of the library in which the design is saved	`-library`

Using the write_milkyway Command

You use the `write_milkyway` command within dc_shell 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. For more information, see “Using a Milkyway Database” on page 10-1.

Saving Designs in .ddc Format

To save the design data in a .ddc file, use the `write -format ddc` command.

By default, the `write` command saves just the top-level design. To save the entire design, specify the `-hier` option. If you do not use the `-output` option to specify the output file name, the `write -format ddc` command creates a file called *top_design*.ddc, where *top_design* is the name of the current design.

Example 1

The following command writes out all designs in the hierarchy of the specified design:

```
dc_shell-xg-t> write -hierarchy -format ddc top
Writing ddc file 'top.ddc'
Writing ddc file 'A.ddc'
Writing ddc file 'B.ddc'
```

Example 2

The following command writes out multiple designs to a single file:

```
dc_shell-xg-t> write -format ddc -output test.ddc \
                    {ADDER MULT16}
Writing ddc file 'test.ddc'
```

Ensuring Name Consistency Between the Design Database and the Netlist

Before writing a netlist from within dc_shell, make sure that all net and port names conform to the naming conventions for your layout tool. Also ensure that you are using a consistent bus naming style.

Some ASIC and EDA vendors have a program that creates a .synopsys_dc.setup file that includes the appropriate commands to convert names to their conventions. If you need to change any net or port names, use the `define_name_rules` and `change_names` commands.

Naming Rules Section of the .synopsys_dc.setup File

Example 5-1 shows sample naming rules as created by a specific layout tool vendor. These naming rules do the following:

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

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

Example 5-1 Naming Rules Section of .synopsys_dc.setup File

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

Using the define_name_rules -map Command

Example 5-2 shows how to use the `-map` option with `define_name_rules` to avoid an error in the format of the string. If you do not follow this convention, an error appears.

Example 5-2 Using define_name_rules -map

```
define_name_rules naming_convention
     -map { {{string1, string2}} } -type cell
```

For example, to remove trailing underscores from cell names, enter

```
dc_shell-xg-t> define_name_rules naming_convention \
                   -map {{{_$, ""}} } -type cell
```

For more information about the `define_name_rules` command, see the man page.

Resolving Naming Problems in the Flow

You might encounter conflicts in naming conventions in design objects, input and output files, and tool sets. In the design database file, you can have many design objects (such as ports, nets, cells, logic modules, and logic module pins), all with their own naming conventions. Furthermore, you might be using several input and output file formats in your flow. Each file format is different and has its own syntax definitions. Using tool sets from several vendors can introduce additional naming problems.

To resolve naming issues, use the `change_names` command to ensure that all the file names match. Correct naming eliminates name escaping or mismatch errors in your design.

For more information about the `change_names` command, see the man page.

Methodology for Resolving Naming Issues

To resolve naming issues, 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 reoptimize your design as you normally would, using your standard set of scripts.

3. Apply name changes and resolve naming issues. Use the `change_names` command and its Verilog or VHDL switch before you write the design.

Important:

Always use the `change_names -rules -[verilog|vhdl] -hierarchy` command whenever you want to write out a Verilog or VHDL design, because naming in the design database file is not Verilog or VHDL compliant. For example, enter

```
change_names -rules verilog -hierarchy
```

4. Write files to disk. Use the `write -format verilog` command.

   Look for reported name changes, which indicate you need to repeat step 3 and refine your name rules.

5. If all the appropriate name changes have been made, your output files matches the design database file. Enter the following commands and compare the output.

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

6. Write the files for third-party tools.

If you need more specific naming control, use the `define_name_rules` command. See "Using the define_name_rules -map Command" on page 5-52.

Summary of Commands for Changing Names

Table 5-7 summarizes commands for changing names.

Table 5-7 Summary of Commands for Changing Names

To do this	Use this
Change the names of ports, cells, and nets in a design to be Verilog or VHDL compliant.	`change_names`

Table 5-7 Summary of Commands for Changing Names (Continued)

To do this	Use this
Show effects of `change_names` without making the changes.	`report_names`
Define a set of rules for naming design objects. Name rules are used by `change_names` and `report_names`.	`define_name_rules`
List available name rules.	`report_name_rules`

Working With Attributes

Attributes describe logical, electrical, physical, and other properties of objects in the design database. An attribute is attached to a design object and is saved with the design database.

Design Compiler uses attributes on the following types of objects:

- Entire designs
- Design objects, such as clocks, nets, pins, and ports
- Design references and cell instances within a design
- Technology libraries, library cells, and cell pins

An attribute has a name, a type, and a value. Attributes can have the following types: string, numeric, or logical (Boolean).

Some attributes are predefined and are recognized by Design Compiler; other attributes are user-defined. Appendix C lists the predefined attributes.

Some attributes are read-only. Design Compiler sets these attribute values and you cannot change them. Other attributes are read/write. You can change these attribute values at any time.

Most attributes apply to one object type; for example, the `rise_drive` attribute applies only to input and inout ports. Some attributes apply to several object types; for example, the `dont_touch` attribute can apply to a net, cell, port, reference, or design. You can get detailed information about the predefined attributes that apply to each object type by using the commands listed in Table 5-8.

Table 5-8 Commands to Get Attribute Descriptions

Object type	Command
All	`man attributes`
Designs	`man design_attributes`
Cells	`man cell_attributes`
Clocks	`man clock_attributes`
Nets	`man net_attributes`
Pins	`man pin_attributes`
Ports	`man port_attributes`
Libraries	`man library_attributes`
Library cells	`man library_cell_attributes`
References	`man reference_attributes`

Setting Attribute Values

To set the value of an attribute, use one of the following:

- An attribute-specific command
- The `set_attribute` command

Using an Attribute-Specific Command

Use an attribute-specific command to set the value of the command's associated attribute.

For example,

```
dc_shell-xg-t> set_dont_touch U1
```

Using the set_attribute Command

Use this command to set the value of any attribute or to define a new attribute and set its value.

For example, to set the `dont_touch` attribute on the lsi_10k/FJK3 library cell, enter

```
dc_shell-xg-t> set_attribute lsi_10K/FJK3 dont_use true
```

The `set_attribute` command enforces the predefined attribute type and generates an error if you try to set an attribute with a value of an incorrect type.

To determine the predefined type for an attribute, use the `list_attributes -application` command. This command generates a list of all application attributes and their types. To

generate a smaller report, you can use the `-class` attribute to limit the list to attributes that apply to one of the following classes: design, port, cell, clock, pin, net, lib, or reference.

For example, the `max_fanout` attribute has a predefined type of float. Suppose you enter the following command, Design Compiler displays an error message:

```
set_attribute lib/lcell/lpin max_fanout 1 -type integer
```

If an attribute applies to more than one object type, Design Compiler searches the database for the named object. For information about the search order, see “The Object Search Order” on page 5-61.

When you set an attribute on a reference (subdesign or library cell), the attribute applies to all cells in the design with that reference. When you set an attribute on an instance (cell, net, or pin), the attribute overrides any attribute inherited from the instance’s reference.

Viewing Attribute Values

To see all attributes on an object, use the `report_attribute` command.

```
dc_shell-xg-t> report_attribute -obj_type object
```

To see the value of a specific attribute on an object, use the `get_attribute` command.

For example, to get the value of the maximum fanout on port OUT7, enter

```
dc_shell-xg-t> get_attribute OUT7 max_fanout
Performing get_attribute on port 'OUT7'.
{3.000000}
```

If an attribute applies to more than one object type, Design Compiler searches the database for the named object. For information about the search order, see "The Object Search Order" on page 5-61.

Saving Attribute Values

Design Compiler does not automatically save attribute values when you exit dc_shell. Use the `write_script` command to generate a dc_shell script that re-creates the attribute values.

Note:

The `write_script` command does not support user-defined attributes.

By default, `write_script` prints to the screen. Use the redirection operator (>) to redirect the output to a file.

```
dc_shell-xg-t> write_script > attr.scr
```

Defining Attributes

The `set_attribute` command enables you to create new attributes. Use the `set_attribute` command described in "Using the set_attribute Command" on page 5-57.

If you want to change the value of an attribute, remove the attribute and then re-create it to store the desired type.

Removing Attributes

To remove a specific attribute from an object, use the `remove_attribute` command.

You cannot use the `remove_attribute` command to remove inherited attributes. For example, if a `dont_touch` attribute is assigned to a reference, remove the attribute from the reference, not from the cells that inherited the attribute.

For example, to remove the `max_fanout` attribute from port OUT7, enter

```
dc_shell-xg-t> remove_attribute OUT7 max_fanout
```

You can remove selected attributes by using the `remove_*` commands. Note that some attributes still require the `set_*` command with a `-default` option specified to remove the attribute previously set by the command. See the man page for a specific command to determine whether it has the `-default` option or uses a corresponding `remove` command.

To remove all attributes from the current design, use the `reset_design` command.

```
dc_shell-xg-t> reset_design
Resetting current design 'EXAMPLE'.
1
```

The `reset_design` command removes all design information, including clocks, input and output delays, path groups, operating conditions, timing ranges, and wire load models. The result of using `reset_design` is often equivalent to starting the design process from the beginning.

The Object Search Order

When Design Compiler searches for an object, the search order is command dependent. (Objects include designs, cells, nets, references, and library cells.)

If you do not use a `get` command, Design Compiler uses an implicit find to locate the object. Commands that can set an attribute on more than one type of object use this search order to determine the object to which the attribute applies.

For example, the `set_dont_touch` command operates on cells, nets, references, and library cells. If you define an object, *X,* with the `set_dont_touch` command and two objects (such as the design and a cell) are named *X,* Design Compiler applies the attribute to the first object type found. (In this case, the attribute is set on the design, not on the cell.)

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

You can override the default search order by using the dctcl `get_*` command to specify the object.

For example, assume that the current design contains both a cell and a net named critical. The following command sets the `dont_touch` attribute on the cell because of the default search order:

```
dc_shell-xg-t> set_dont_touch critical
1
```

To place the `dont_touch` attribute on the net instead, use the following command:

```
dc_shell-xg-t> set_dont_touch [get_nets critical]
1
```

6

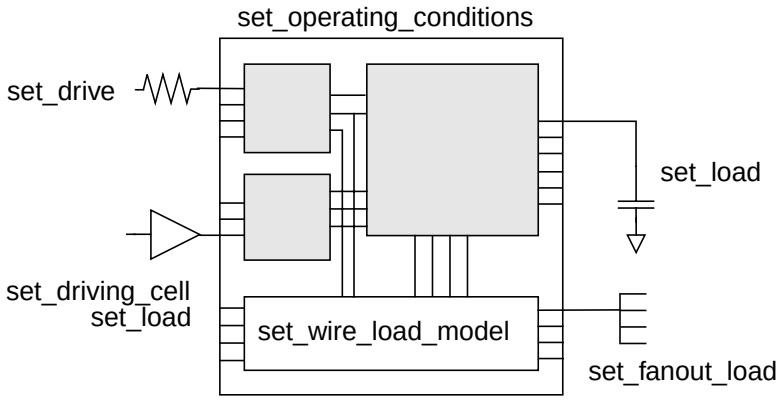
Defining the Design Environment

Before a design can be optimized, you must define the environment in which the design is expected to operate. You define the environment by specifying operating conditions, wire load models, and system interface characteristics.

Operating conditions include temperature, voltage, and process variations. Wire load models estimate the effect of wire length on design performance. System interface characteristics include input drives, input and output loads, and fanout loads. The environment model directly affects design synthesis results.

In Design Compiler, the model is defined by a set of attributes and constraints that you assign to the design, using specific dc_shell commands. Figure 6-1 illustrates the commands used to define the design environment.

Figure 6-1 Commands Used to Define the Design Environment



This chapter contains the following sections:

- Defining the Operating Conditions
- Defining Wire Load Models
- Modeling the System Interface

Defining the Operating Conditions

In most technologies, variations in operating temperature, supply voltage, and manufacturing process can strongly affect circuit performance (speed). These factors, called operating conditions, have the following general characteristics:

- Operating temperature variation

  Temperature variation is unavoidable in the everyday operation of a design. Effects on performance caused by temperature fluctuations are most often handled as linear scaling effects, but some submicron silicon processes require nonlinear calculations.

- Supply voltage variation

  The design's supply voltage can vary from the established ideal value during day-to-day operation. Often a complex calculation (using a shift in threshold voltages) is employed, but a simple linear scaling factor is also used for logic-level performance calculations.

- Process variation

  This variation accounts for deviations in the semiconductor fabrication process. Usually process variation is treated as a percentage variation in the performance calculation.

When performing timing analysis, Design Compiler must consider the worst-case and best-case scenarios for the expected variations in the process, temperature, and voltage factors.

Determining Available Operating Condition Options

Most technology libraries have predefined sets of operating conditions. Use the `report_lib` command to list the operating conditions defined in a technology library. The library must be loaded in memory before you can run the `report_lib` command. To see the list of libraries loaded in memory, use the `list_libraries` or the `list_libs` command.

For example, to generate a report for the library my_lib, which is stored in my_lib.db, enter the following commands:

```
dc_shell-xg-t> read_file my_lib.db
dc_shell-xg-t> report_lib my_lib
```

Example 6-1 shows the resulting operating conditions report.

Example 6-1 Operating Conditions Report

```
****************************************
Report : library
Library: my_lib
Version: X-2005.09
Date   : Mon Jan 13 10:56:49 2005
****************************************
...
Operating Conditions:

    Name       Library       Process     Temp    Volt    Interconnect Model
   -------------------------------------------------------------------------
    WCCOM      my_lib            1.50    70.00    4.75    worst_case_tree
    WCIND      my_lib            1.50    85.00    4.75    worst_case_tree
    WCMIL      my_lib            1.50   125.00    4.50    worst_case_tree
...
```

Specifying Operating Conditions

If the technology library contains operating condition specifications, you can let Design Compiler use them as default conditions. Alternatively, you can use the `set_operating_conditions` command to specify explicit operating conditions, which supersede the default library conditions.

For example, to set the operating conditions for the current design to worst-case commercial, enter

```
dc_shell-xg-t> set_operating_conditions WCCOM -lib my_lib
```

Use the `report_design` command to see the operating conditions defined for the current design.

Defining Wire Load Models

Wire load modeling allows you to estimate the effect of wire length and fanout on the resistance, capacitance, and area of nets. Design Compiler uses these physical values to calculate wire delays and circuit speeds.

Semiconductor vendors develop wire load models, based on statistical information specific to the vendors' process. The models include coefficients for area, capacitance, and resistance per unit length, and a fanout-to-length table for estimating net lengths (the number of fanouts determines a nominal length).

Note:

You can also develop custom wire load models. For more information about developing wire load models, see the Library Compiler documentation.

In the absence of back-annotated wire delays, Design Compiler uses the wire load models to estimate net wire lengths and delays. Design Compiler determines which wire load model to use for a design, based on the following factors, listed in order of precedence:

1. Explicit user specification
2. Automatic selection based on design area
3. Default specification in the technology library

If none of this information exists, Design Compiler does not use a wire load model. Without a wire load model, Design Compiler does not have complete information about the behavior of your target technology and cannot compute loading or propagation times for your nets; therefore, your timing information will be optimistic.

In hierarchical designs, Design Compiler must also determine which wire load model to use for nets that cross hierarchical boundaries. The tool determines the wire load model for cross-hierarchy nets based on one of the following factors, listed in order of precedence:

1. Explicit user specification
2. Default specification in the technology library
3. Default mode in Design Compiler

The following sections discuss the selection of wire load models for nets and designs.

Hierarchical Wire Load Models

Design Compiler supports three modes for determining which wire load model to use for nets that cross hierarchical boundaries:

- Top

  Design Compiler models nets as if the design has no hierarchy and uses the wire load model specified for the top level of the design hierarchy for all nets in a design and its subdesigns. The tool ignores any wire load models set on subdesigns with the `set_wire_load_model` command.

  Use top mode if you plan to flatten the design at a higher level of hierarchy before layout.

- Enclosed

  Design Compiler uses the wire load model of the smallest design that fully encloses the net. If the design enclosing the net has no wire load model, the tool traverses the design hierarchy upward until it finds a wire load model. Enclosed mode is more accurate than top mode when cells in the same design are placed in a contiguous region during layout.

  Use enclosed mode if the design has similar logical and physical hierarchies.

- Segmented

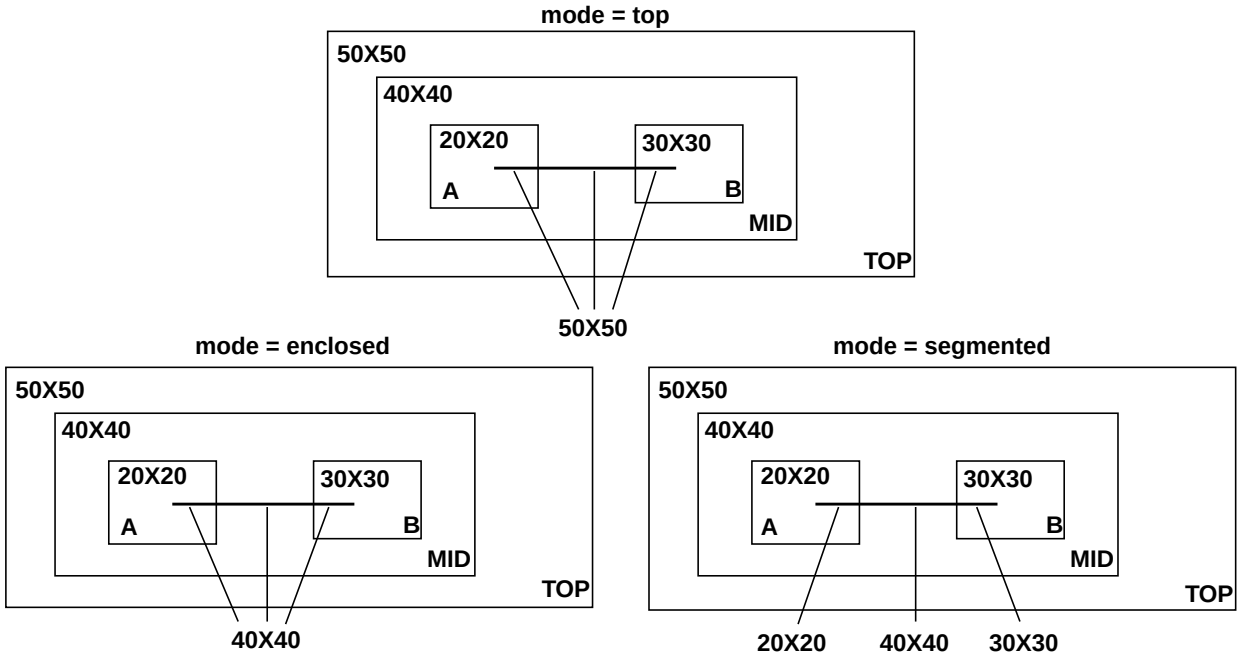
  Design Compiler determines the wire load model of each segment of a net by the design encompassing the segment. Nets crossing hierarchical boundaries are divided into segments. For each net segment, Design Compiler uses the wire load model of

the design containing the segment. If the design contains a segment that has no wire load model, the tool traverses the design hierarchy upward until it finds a wire load model.

Use segmented mode if the wire load models in your technology have been characterized with net segments.

Figure 6-2 shows a sample design with a cross-hierarchy net, cross_net. The top level of the hierarchy (design TOP) has a wire load model of 50x50. The next level of hierarchy (design MID) has a wire load model of 40x40. The leaf-level designs, A and B, have wire load models of 20x20 and 30x30, respectively.

Figure 6-2 Comparison of Wire Load Mode



In top mode, Design Compiler estimates the wire length of net cross_net, using the 50x50 wire load model. Design Compiler ignores the wire load models on designs MID, A, and B.

In enclosed mode, Design Compiler estimates the wire length of net cross_net, using the 40x40 wire load model (the net cross_net is completely enclosed by design MID).

In segmented mode, Design Compiler uses the 20x20 wire load model for the net segment enclosed in design A, the 30x30 wire load model for the net segment enclosed in design B, and the 40x40 wire load model for the segment enclosed in design MID.

Determining Available Wire Load Models

Most technology libraries have predefined wire load models. Use the `report_lib` command to list the wire load models defined in a technology library. The library must be loaded in memory before you run the `report_lib` command. To see a list of libraries loaded in memory, use the `list_libs` command.

The wire load report contains the following sections:

- Wire Loading Model section

  This section lists the available wire load models.

- Wire Loading Model Mode section

  This section identifies the default wire load mode. If a library default does not exist, Design Compiler uses top mode.

- Wire Loading Model Selection Group section

  The presence of this section indicates that the library supports automatic area-based wire load model selection.

To generate a wire load report for the my_lib library, enter

```
dc_shell-xg-t> read_file my_lib.db
```

```
dc_shell-xg-t> report_lib my_lib
```

Example 6-2 shows the resulting wire load models report. The library my_lib contains three wire load models: 05x05, 10x10, and 20x20. The library does not specify a default wire load mode (so Design Compiler uses top as the default wire load mode), and it supports automatic area-based wire load model selection.

Example 6-2 Wire Load Models Report

```
****************************************
Report : library
Library: my_lib
Version: Y-2006.06
Date   : Mon May 1 10:56:49 2006
****************************************
...
Wire Loading Model:

Name           :   05x05
Location       :   my_lib
Resistance     :   0
Capacitance    :   1
Area           :   0
Slope          :   0.186
Fanout   Length   Points Average Cap Std Deviation
------------------------------------------------------------------------
     1     0.39

Name           :   10x10
Location       :   my_lib
Resistance     :   0
Capacitance    :   1
Area           :   0
Slope          :   0.311
Fanout   Length   Points Average Cap Std Deviation
------------------------------------------------------------------------
     1     0.53
```

Example 6-2 Wire Load Models Report (Continued)

```
Name           :   20x20
Location       :   my_lib
Resistance     :   0
Capacitance    :   1
Area           :   0
Slope          :   0.547
Fanout   Length   Points Average Cap Std Deviation
------------------------------------------------------------------------
     1     0.86
Wire Loading Model Selection Group:

   Name           : my_lib

        Selection             Wire load name
    min area   max area
   -----------------------------------------
        0.00    1000.00          05x05
     1000.00    2000.00          10x10
     2000.00    3000.00          20x20
...
```

Specifying Wire Load Models and Modes

The technology library can define a default wire load model that is used for all designs implemented in that technology. The `default_wire_load` library attribute identifies the default wire load model for a technology library.

Some libraries support automatic area-based wire load selection. Design Compiler uses the library function `wire_load_selection` to choose a wire load model based on the total cell area. The wire load model selected the first time you compile is used in subsequent compiles.

For large designs with many levels of hierarchy, automatic wire load selection can increase runtime. To manage runtime, set the wire load manually.

You can turn off automatic selection of the wire load model by setting the `auto_wire_load_selection` variable to false. For example, enter the following commands:

```
dc_shell-xg-t> set auto_wire_load_selection false
```

The technology library can also define a default wire load mode. The `default_wire_load_mode` library attribute identifies the default mode. If the current library does not define a default mode, Design Compiler looks for the attribute in the libraries specified in the `link_library` variable. (To see the link library, use the `list` command.) In the absence of a library default (and an explicit specification), Design Compiler uses that top mode.

To change the wire load model or mode specified in a technology library, use the `set_wire_load_model` and `set_wire_load_mode` commands. The wire load model and mode you define override all defaults. Explicitly selecting a wire load model also disables area-based wire load model selection for that design.

For example, to select the 10x10 wire load model, enter

```
dc_shell-xg-t> set_wire_load_model "10x10"
```

To select the 10x10 wire load model and specify enclosed mode, enter

```
dc_shell-xg-t> set_wire_load_mode enclosed
```

The wire load model you choose for a design depends on how that design is implemented in the chip. Consult your semiconductor vendor to determine the best wire load model for your design.

Use the `report_design` or `report_timing` commands to see the wire load model and mode defined for the current design.

To remove the wire load model, use the `remove_wire_load_model` command with no model name.

Modeling the System Interface

Design Compiler supports the following ways to model the design's interaction with the external system:

- Defining drive characteristics for input ports
- Defining loads on input and output ports
- Defining fanout loads on output ports

The following sections discuss these tasks.

Defining Drive Characteristics for Input Ports

Design Compiler uses drive strength information to buffer nets appropriately in the case of a weak driver.

Note:

Drive strength is the reciprocal of the output driver resistance, and the transition time delay at an input port is the product of the drive resistance and the capacitance load of the input port.

By default, Design Compiler assumes zero drive resistance on input ports, meaning infinite drive strength. There are three commands for overriding this unrealistic assumption:

- `set_driving_cell`
- `set_drive`

- `set_input_transition`

Both the `set_driving_cell` and `set_input_transition` commands affect the port transition delay, but they do not place design rule requirements, such as `max_fanout` and `max_transition`, on input ports. However, the `set_driving_cell` command does place design rules on input ports if the driving cell has DRCs.

Note:

For heavily loaded driving ports, such as clock lines, keep the drive strength setting at 0 so that Design Compiler does not buffer the net. Each semiconductor vendor has a different way of distributing these signals within the silicon.

Both the `set_drive` and the `set_driving_cell` commands affect the port transition delay. The `set_driving_cell` command can place design rule requirements, such as `max_fanout` or `max_transition`, on input ports if the specified cell has input ports.

The most recently used command takes precedence. For example, setting a drive resistance on a port with the `set_drive` command overrides previously run `set_driving_cell` commands.

The set_driving_cell Command

Use the `set_driving_cell` command to specify drive characteristics on ports that are driven by cells in the technology library. This command is compatible with all the delay models, including the nonlinear delay model and piecewise linear delay model. The `set_driving_cell` command associates a library pin with an input port so that delay calculators can accurately model the drive capability of an external driver.

Use the `remove_driving_cell` command or `reset_design` command to remove driving cell attributes on ports.

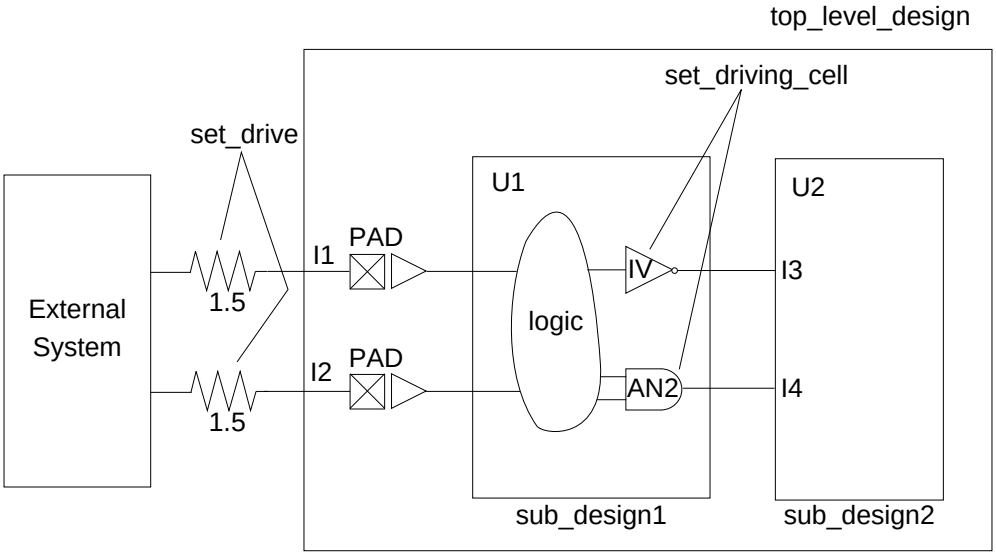
The set_drive and set_input_transition Commands

Use the `set_drive` or `set_input_transition` command to set the drive resistance on the top-level ports of the design when the input port drive capability cannot be characterized with a cell in the technology library.

You can use `set_drive` and the `drive_of` commands together to represent the drive resistance of a cell. However, these commands are not as accurate for nonlinear delay models as the `set_driving_cell` command is.

Figure 6-3 shows a hierarchical design. The top-level design has two subdesigns, U1 and U2. Ports I1 and I2 of the top-level design are driven by the external system and have a drive resistance of 1.5.

Figure 6-3 Drive Characteristics



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

1. Because ports I1 and I2 are not driven by library cells, use the `set_drive` command to define the drive resistance. Enter

```
dc_shell-xg-t> current_design top_level_design
dc_shell-xg-t> set_drive 1.5 {I1 I2}
```

2. To describe the drive capability for the ports on design sub_design2, change the current design to sub_design2. Enter

```
dc_shell-xg-t> current_design sub_design2
```

3. An IV cell drives port I3. Use the `set_driving_cell` command to define the drive resistance. Because IV has only one output and one input, define the drive capability as follows. Enter

```
dc_shell-xg-t> set_driving_cell -lib_cell IV {I3}
```

4. An AN2 cell drives port I4. Because the different arcs of this cell have different transition times, select the worst-case arc to define the drive. For checking setup violations, the worst-case arc is the slowest arc. For checking hold violations, the worst-case arc is the fastest arc.

   For this example, assume that you want to check for setup violations. The slowest arc on the AN2 cell is the B-to-Z arc, so define the drive as follows. Enter

```
dc_shell-xg-t> set_driving_cell -lib_cell AN2 -pin Z \
          -from_pin B {I4}
```

Defining Loads on Input and Output Ports

By default, Design Compiler assumes zero capacitive load on input and output ports. Use the `set_load` command to set a capacitive load value on input and output ports of the design. This information helps Design Compiler select the appropriate cell drive strength of an output pad and helps model the transition delay on input pads.

For example, to set a load of 30 on output pin out1, enter

```
dc_shell-xg-t> set_load 30 {out1}
```

Make the units for the load value consistent with the target technology library. For example, if the library represents the load value in picofarads, the value you set with the `set_load` command must be in picofarads. Use the `report_lib` command to list the library units.

Example 6-3 shows the library units for the library my_lib.

Example 6-3 Library Units Report

```
****************************************
Report : library
Library: my_lib
Version: 1999.05
Date   : Mon Jan 4 10:56:49 1999
****************************************

Library Type             : Technology
Tool Created             : 1999.05
Date Created             : February 7, 1992
Library Version          : 1.800000
Time Unit                : 1ns
Capacitive Load Unit     : 0.100000ff
Pulling Resistance Unit  : 1kilo-ohm
Voltage Unit             : 1V
Current Unit             : 1uA
...
```

Defining Fanout Loads on Output Ports

You can model the external fanout effects by specifying the expected fanout load values on output ports with the `set_fanout_load` command.

For example, enter

```
dc_shell-xg-t> set_fanout_load 4 {out1}
```

Design Compiler tries to ensure that the sum of the fanout load on the output port plus the fanout load of cells connected to the output port driver is less than the maximum fanout limit of the library, library cell, and design. (For more information about maximum fanout limits, see “Setting Design Rule Constraints” on page 7-3.)

Fanout load is not the same as load. Fanout load is a unitless value that represents a numerical contribution to the total fanout. Load is a capacitance value. Design Compiler uses fanout load primarily to measure the fanout presented by each input pin. An input pin normally has a fanout load of 1, but it can have a higher value.

7

Defining Design Constraints

In addition to specifying the design environment, you must set design constraints before compiling the design. There are two categories of design constraints:

- Design rule constraints
- Design optimization constraints

Design rule constraints are supplied in the technology library you specify. They are referred to as the *implicit* design rules. These rules are established by the library vendor, and, for the proper functioning of the fabricated circuit, they must not be violated. You can, however, specify stricter design rules if appropriate. The rules you specify are referred to as the *explicit* design rules.

Design optimization constraints define timing and area optimization goals for Design Compiler. These constraints are user-specified. Design Compiler optimizes the synthesis of the design, in

accordance with these constraints, but not at the expense of the design rule constraints. That is, Design Compiler attempts never to violate the higher-priority design rules.

Note:

In this chapter, setting explicit design rules and optimization constraints is discussed without reference to the particular compile strategy you choose. But the compile strategy you choose does influence your constraint settings.

This chapter contains the following sections:

- Setting Design Rule Constraints
- Setting Optimization Constraints
- Verifying the Precompiled Design

The task of setting timing constraints can be complicated (especially setting the timing exceptions) and includes the following tasks:

- Defining a Clock
- Specifying I/O Timing Requirements
- Specifying Combinational Path Delay Requirements
- Specifying Timing Exceptions

Setting Design Rule Constraints

This section discusses the most commonly specified design rule constraints:

- Transition time
- Fanout load
- Capacitance

Design Compiler also supports cell degradation and connection class constraints. For information about these constraints, see the *Design Compiler Reference Manual: Constraints and Timing*.

Design Compiler uses attributes assigned to the design's objects to represent design rule constraints. Table 7-1 provides the attribute name that corresponds to each design rule constraint.

Table 7-1 Design Rule Attributes

Design rule constraint	Attribute name
Transition time	max_transition
Fanout load	max_fanout
Capacitance	max_capacitance min_capacitance
Cell degradation	cell_degradation
Connection class	connection_class

Design rule constraints are attributes specified in the technology library and, optionally, specified by you explicitly.

If a technology library defines these attributes, Design Compiler implicitly applies them to any design using that library when it compiles the design or creates a constraint report. You cannot remove the design rule attributes defined in the technology library, because they are requirements for the technology, but you can make them more restrictive to suit your design.

If both implicit and explicit design rule constraints apply to a design or a net, the more restrictive value takes precedence.

Setting Transition Time Constraints

The transition time of a net is the time required for its driving pin to change logic values. This transition time is based on the technology library data. For the nonlinear delay model (NLDM), output transition time is a function of input transition and output load.

Design Compiler and Library Compiler model transition time restrictions by associating a `max_transition` attribute with each output pin on a cell. During optimization, Design Compiler attempts to make the transition time of each net less than the value of the `max_transition` attribute.

To change the maximum transition time restriction specified in a technology library, use the `set_max_transition` command. This command sets a maximum transition time for the nets attached to the identified ports or to all the nets in a design by setting the `max_transition` attribute on the named objects.

For example, to set a maximum transition time of 3.2 on all nets in the design adder, enter the following command:

```
dc_shell-xg-t> set_max_transition 3.2 [get_designs adder]
```

To undo a `set_max_transition` command, use the `remove_attribute` command. For example, enter the following command:

```
dc_shell-xg-t>remove_attribute [get_designs adder] \
          max_transition
```

Setting Fanout Load Constraints

The maximum fanout load for a net is the maximum number of loads the net can drive.

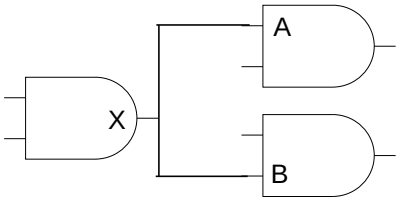
Design Compiler and Library Compiler model fanout restrictions by associating a `fanout_load` attribute with each input pin and a `max_fanout` attribute with each output (driving) pin on a cell.

The fanout load value does not represent capacitance; it represents the weighted numerical contribution to the total fanout load. The fanout load imposed by an input pin is not necessarily 1.0. Library developers can assign higher fanout load values to model internal cell fanout effects.

Design Compiler calculates the fanout of a driving pin by adding the `fanout_load` values of all inputs driven by that pin. To determine whether the pin meets the maximum fanout load restriction, Design Compiler compares the calculated fanout load value with the pin's `max_fanout` value.

Figure 7-1 shows a small circuit in which pin X drives two loads, pin A and pin B. If pin A has a `fanout_load` value of 1.0 and pin B has a `fanout_load` value of 2.0, the total fanout load of pin X is 3.0. If pin X has a maximum fanout greater than 3.0, say 16.0, the pin meets the fanout constraints.

Figure 7-1 Fanout Constraint Example



During optimization, Design Compiler attempts to meet the fanout load restrictions for each driving pin. If a pin violates its fanout load restriction, Design Compiler tries to correct the problem (for example, by changing the drive strength of the component).

The technology library might specify default fanout constraints on the entire library or fanout constraints for specific pins in the library description of an individual cell.

To determine whether your technology library is modeled for fanout calculations, you can search for the `fanout_load` attribute on the cell input pins by entering the following command:

```
dc_shell-xg-t> get_attribute [get_pins my_lib/*/*] \
               fanout_load
```

To set a more conservative fanout restriction than that specified in the technology library, use the `set_max_fanout` command on the design or on an input port. (Use the `set_fanout_load` command to set the expected fanout load value for output ports.)

The `set_max_fanout` command sets the maximum fanout load for the specified input ports or for all the nets in a design by setting the `max_fanout` attribute on the specified objects. For example, to set a `max_fanout` requirement of 16 on all nets in the design adder, enter the following commands:

```
dc_shell-xg-t> set_max_fanout 16 [get_designs adder]
```

If you use the `set_max_fanout` command and a library `max_fanout` attribute exists, Design Compiler tries to meet the smaller (more restrictive) fanout limit.

To undo a `set_max_fanout` command, use the `remove_attribute` command. For example, enter the following command:

```
dc_shell-xg-t> remove_attribute [get_designs adder]\
                                    max_fanout
```

Setting Capacitance Constraints

The transition time constraints do not provide a direct way to control the actual capacitance of nets. If you need to control capacitance directly, use the `set_max_capacitance` command to set the maximum capacitance constraint on input ports or designs. This constraint is completely independent, so you can use it in addition to the transition time constraints.

Design Compiler and Library Compiler model capacitance restrictions by associating the `max_capacitance` attribute with the output ports or pins of a cell. Design Compiler calculates the capacitance on a net by adding the wire capacitance of the net to the capacitance of the pins attached to the net. To determine whether a net meets the capacitance constraint, Design Compiler compares the calculated capacitance value with the `max_capacitance` value of the pin driving the net.

For example, to set a maximum capacitance of 3 for all nets in the design adder, enter the following command:

```
dc_shell-xg-t> set_max_capacitance 3 [get_designs adder]
```

To undo a `set_max_capacitance` command, use the `remove_attribute` command. For example, enter the following command:

```
dc_shell-xg-t> remove_attribute [get_designs adder] \
          max_capacitance
```

You can also use the `set_min_capacitance` command to define the minimum capacitance for input ports or pins. Design Compiler attempts to ensure that the load seen at the input port does not fall below the specified capacitance value, but it does not specifically optimize for this constraint.

Setting Optimization Constraints

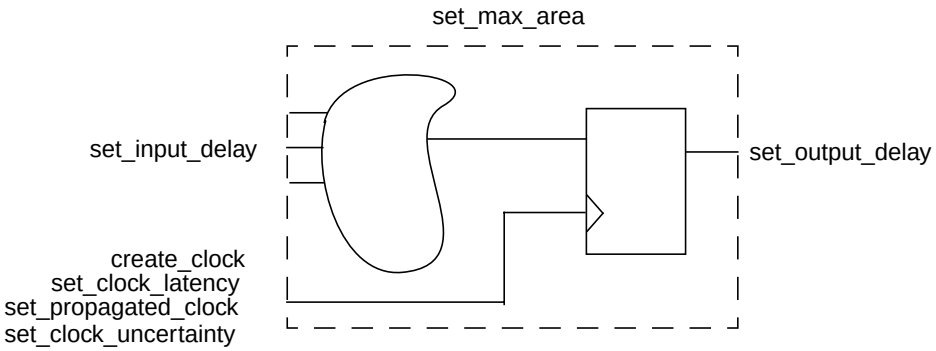
This section discusses the most commonly specified optimization constraints:

- Timing constraints
- Area constraints

Design Compiler also supports power constraints. For information about power constraints, see the *Power Compiler Reference Manual*.

Figure 7-2 illustrates some of the common commands used to define the optimization constraints.

Figure 7-2 Commands Used to Define the Optimization Constraints for Sequential Blocks



Setting Timing Constraints

Timing constraints specify the required performance of the design. To set the timing constraints,

1. Define the clocks.
2. Specify the I/O timing requirements relative to the clocks.
3. Specify the combinational path delay requirements.
4. Specify the timing exceptions.

Table 7-2 lists the most commonly used commands for these steps.

Table 7-2 Commands to Set Timing Constraints

Command	Description
`create_clock`	Defines the period and waveform for the clock.

`set_clock_latency`
`set_propagated_clock`
`set_clock_uncertainty`
	Defines the clock delay.

Table 7-2 Commands to Set Timing Constraints (Continued)

Command	Description
`set_input_delay`	Defines the timing requirements for input ports relative to the clock period.
`set_output_delay`	Defines the timing requirements for output ports relative to the clock period.
`set_max_delay`	Defines maximum delay for combinational paths. (This is a timing exception command.)
`set_min_delay`	Defines minimum delay for combinational paths. (This is a timing exception command.)
`set_false_path`	Specifies false paths. (This is a timing exception command.)
`set_multicycle_path`	Specifies multicycle paths. (This is a timing exception command.)

The following sections describe these steps in more detail.

Defining a Clock

For synchronous designs, the clock period is the most important constraint because it constrains all register-to-register paths in the design.

Defining the Period and Waveform for the Clock. Use the `create_clock` command to define the period (`-period` option) and waveform (`-waveform` option) for the clock. If you do not specify the clock waveform, Design Compiler uses a 50 percent duty cycle.

Use the `create_clock` command on a pin or a port. For example, to specify a 25-megahertz clock on port clk with a 50 percent duty cycle, enter

```
dc_shell-xg-t> create_clock clk -period 40
```

When your design contains multiple clocks, pay close attention to the common base period of the clocks. The common base period is the least common multiple of all the clock periods. For example, if you have clock periods of 10, 15, and 20, the common base period is 60.

Define your clocks so that the common base period is a small integer multiple of each of the clock periods. The common base period requirement is qualitative; no hard limit exists. If the base period is more than 10 times larger than the smallest period, however, long runtimes and greater memory requirements can result.

As an extreme case, if you have a register-to-register path where one register has a period of 10 and the other has a period of 10.1, the common base period is 1010.0. The timing analyzer calculates the setup requirement for this path by expanding both clocks to the common base period and determining the tightest single-cycle relationship for setup. Internally, for extreme cases such as this, the timing analyzer only approximates the setup requirement because the paths are not really synchronous.

You can work around this problem by specifying a clock period without a decimal point and adjusting the clock period by inserting clock uncertainty.

```
dc_shell-xg-t> create_clock -period 10 clk1
dc_shell-xg-t> create_clock -period 10 clk2
dc_shell-xg-t> set_clock_uncertainty -setup 0.1 clk2
```

Use the `report_clock` command to show information about all clock sources in your design.

Use the `remove_clock` command to remove a clock definition.

Creating a Virtual Clock. In some cases, a system clock might not exist in a block. You can use the `create_clock -name` command to create a virtual clock for modeling clock signals present in the system but not in the block. By creating a virtual clock, you can represent delays that are relative to clocks outside the block.

```
dc_shell-xg-t> create_clock -period 30 -waveform {10 25} \
          -name sys_clk
```

Specifying Clock Network Delay. By default, Design Compiler assumes that clock networks have no delay (ideal clocks). Use the `set_clock_latency` and `set_clock_uncertainty` commands to specify timing information about the clock network delay. You can use these commands to specify either estimated or actual delay information.

Use the `set_propagated_clock` command to specify that you want the clock latency to propagate through the clock network. For example,

```
dc_shell-xg-t> set_propagated_clock clk
```

Use the `-setup` or `-hold` options of the `set_clock_uncertainty` command to add some margin of error into the system to account for variances in the clock network resulting from layout. For example, on the 20-megahertz clock mentioned previously, to add a 0.2 margin on each side of the clock edge, enter

```
dc_shell-xg-t> set_clock_uncertainty -setup 0.2 clk
dc_shell-xg-t> set_clock_uncertainty -hold 0.2 clk
```

Use the `-skew` option of the `report_clock` command to show clock network skew information. Design Compiler uses the clock information when determining whether a path meets setup and hold requirements.

Specifying I/O Timing Requirements

If you do not assign timing requirements to an input port, Design Compiler responds as if the signal arrives at the input port at time 0. In most cases, input signals arrive at staggered times. Use the `set_input_delay` command to define the arrival times for input ports. You define the input delay constraint relative to the system clock and to the other inputs.

If you do not assign timing requirements to an output port, Design Compiler does not constrain any paths which end at an output port. Use the `set_output_delay` command to define the required output arrival time. You define the output delay constraint relative to the system clock.

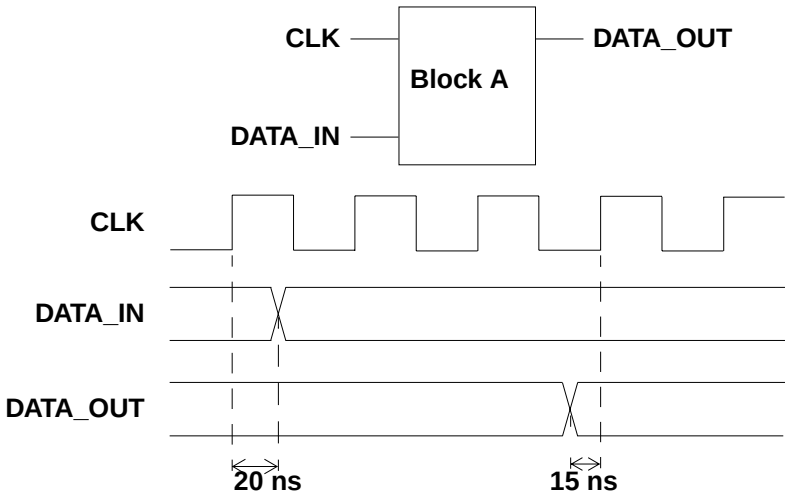
If an input or output port has multiple timing requirements (because of multiple paths), use the `-add_delay` option to specify the additional timing requirements.

Use the `report_port` command to list input or output delays associated with ports.

Use the `remove_input_delay` command to remove input delay constraints. Use the `remove_output_delay` command to remove output delay constraints.

Figure 7-3 shows the timing relationship between the delay and the active clock edge (the rising edge in this example).

Figure 7-3 Relationship Between Delay and Active Clock Edge



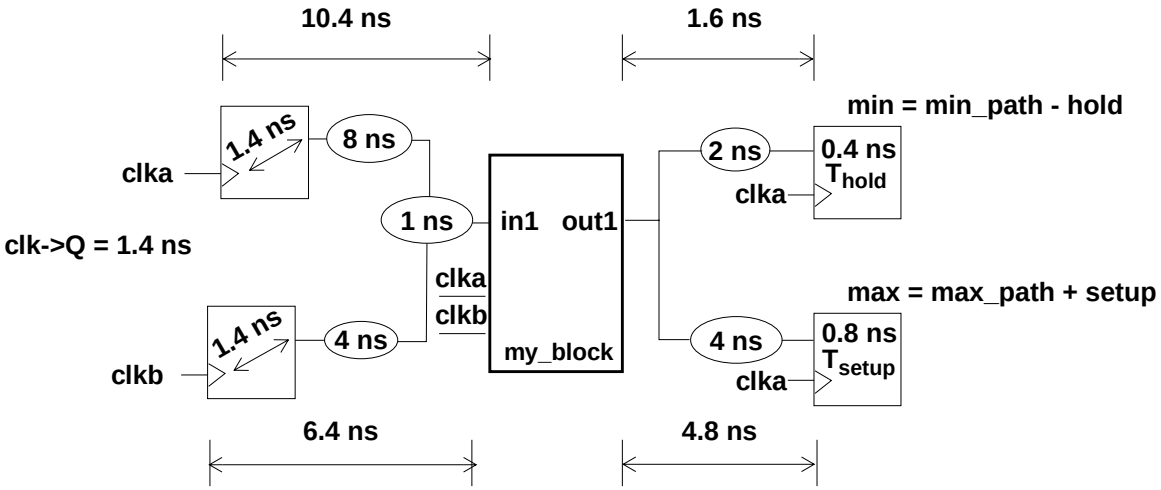
In the figure, block A has an input DATA_IN and an output DATA_OUT. From the waveform diagram, DATA_IN is stable 20 ns after the clock edge, and DATA_OUT needs to be available 15 ns before the clock edge.

After you set the clock constraint by using the `create_clock` command, use the `set_input_delay` and `set_output_delay` commands to specify these additional requirements. For example, enter

```
dc_shell-xg-t> set_input_delay 20 -clock CLK DATA_IN
dc_shell-xg-t> set_output_delay 15 -clock CLK DATA_OUT
```

Figure 7-4 illustrates the timing requirements for the constrained design block my_block. Example 7-1 shows the script used to specify these timing requirements.

Figure 7-4 Timing Requirements for my_block



Example 7-1 Timing Constraints for my_block

```
create_clock -period 20 -waveform {5 15} clka
create_clock -period 30 -waveform {10 25} clkb
set_input_delay 10.4 -clock clka in1
set_input_delay 6.4 -clock clkb -add_delay in1
set_output_delay 1.6 -clock clka -min out1
set_output_delay 4.8 -clock clka -max out1
```

Specifying Combinational Path Delay Requirements

For purely combinational delays that are not bounded by a clock period, use the `set_max_delay` and `set_min_delay` commands to define the maximum and minimum delays for the specified paths.

A common way to produce this type of asynchronous logic in HDL code is to use asynchronous sets or resets on latches and flip-flops. Because the reset signal crosses several blocks, constrain this signal at the top level.

For example, to specify a maximum delay of 5 on the RESET signal, enter

```
dc_shell-xg-t> set_max_delay 5 -from RESET
```

To specify a minimum delay of 10 on the path from IN1 to OUT1, enter

```
dc_shell-xg-t> set_min_delay 10 -from IN1 -to OUT1
```

Use the `report_timing_requirements` command to list the minimum delay and maximum delay requirements for your design.

Specifying Timing Exceptions

Timing exceptions define timing relationships that override the default single-cycle timing relationship for one or more timing paths. Use timing exceptions to constrain or disable asynchronous paths or paths that do not follow the default single-cycle behavior.

Note:

Specifying numerous timing exceptions can increase the compile runtime. Nevertheless, some designs can require many timing exceptions.

Design Compiler recognizes only timing exceptions that have valid reference points.

- The valid startpoints in a design are the primary input ports and the clock pins of sequential cells.
- The valid endpoints are the primary output ports of a design and the data pins of sequential cells.

Design Compiler does not generate a warning message if you specify invalid reference points. You must use the `-ignored` option of the `report_timing_requirements` command to find timing exceptions ignored by Design Compiler.

You can specify the following conditions by using timing exception commands:

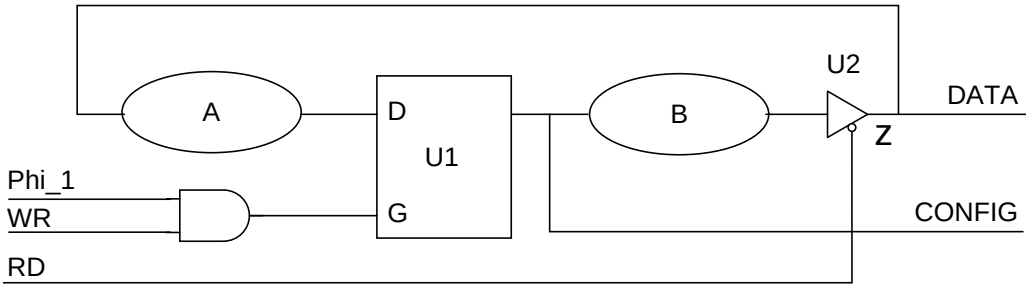
- False paths (`set_false_path`)
- Minimum delay requirements (`set_min_delay`)
- Maximum delay requirements (`set_max_delay`)
- Multicycle paths (`set_multicycle_path`)

Use the `report_timing_requirements` command to list the timing exceptions in your design.

Specifying False Paths. Design Compiler does not report false paths in the timing report or consider them during timing optimization. Use the `set_false_path` command to specify a false path. Use this command to ignore paths that are not timing critical, that can mask other paths that must be considered during optimization, or that never occur in normal operation.

For example, Figure 7-5 shows a configuration register that can be written and read from a bidirectional bus (DATA) in a chip.

Figure 7-5 Configuration Register



The circuit has these timing paths:

1. DATA to U1/D

2. RD to DATA
3. U1/G to CONFIG (with possible time borrowing at U1/D)
4. U1/G to DATA (with possible time borrowing at U1/D)
5. U1/G to U1/D (through DATA, with possible time borrowing)

The first four paths are valid paths. The fifth path (U1/G to U1/D) is a functional false path because normal operation never requires simultaneous writing and reading of the configuration register. In this design, you can disable the false path by using this command:

```
dc_shell-xg-t> set_false_path -from U1/G -to U1/D
```

To undo a `set_false_path` command, use the `reset_path` command with similar options. For example, enter

```
dc_shell-xg-t> set_false_path -setup -from IN2 -to FF12/D
dc_shell-xg-t> reset_path -setup -from IN2 -to FF12/D
```

Creating a false path differs from disabling a timing arc. Disabling a timing arc represents a break in the path. The disabled timing arc permanently disables timing through all affected paths. Specifying a path as false does not break the path; it just prevents the path from being considered for timing or optimization.

Specifying Minimum and Maximum Delay Requirements. You can use the `set_min_delay` and `set_max_delay` commands, described earlier in this chapter, to specify path delay requirements that are more conservative than those derived by Design Compiler based on the clock timing.

To undo a `set_min_delay` or `set_max_delay` command, use the `reset_path` command with similar options.

Register-to-Register Paths. Design Compiler uses the following equations to derive constraints for minimum and maximum path delays on register-to-register paths:

$$\texttt{min\_delay} = (T_{capture} - T_{launch}) + \texttt{hold}$$
$$\texttt{max\_delay} = (T_{capture} - T_{launch}) - \texttt{setup}$$

You can override the derived path delay ($T_{capture} – T_{launch}$) by using the `set_min_delay` and `set_max_delay` commands.

For example, assume that you have a path launched from a register at time 20 that arrives at a register where the next active edge of the clock occurs at time 35.

```
dc_shell-xg-t> create_clock -period 40 waveform {0 20} clk1
dc_shell-xg-t> create_clock -period 40 -waveform {15 35} clk2
```

Design Compiler automatically derives a maximum path delay constraint of (35 – 20) – (library setup time of register at endpoint). To specify a maximum path delay of 10, enter

```
dc_shell-xg-t> set_max_delay 10 -from reg1 -to reg2
```

Design Compiler calculates the maximum path delay constraint as 10 – (library setup time of register at endpoint), which overrides the original derived maximum path delay constraint.

Register-to-Port Paths. Design Compiler uses the following equations to derive constraints for minimum and maximum path delays on register-to-port paths:

```
min_delay = period - output_delay
max_delay = period - output_delay
```

If you use the set_min_delay or set_max_delay commands, the value specified in these commands replaces the period value in the constraint calculation. For example, assume you have a design with a clock period of 20. Output OUTPORTA has an output delay of 5.

```
dc_shell-xg-t> create_clock -period 20 CLK
dc_shell-xg-t> set_output_delay 5 -clock CLK OUTPORTA
```

Design Compiler automatically derives a maximum path delay constraint of 15 (20 – 5). To specify that you want a maximum path delay of 10, enter

```
dc_shell-xg-t> set_max_delay 10 -to OUTPORTA
```

Design Compiler calculates the maximum path delay constraint as 5 (10 – 5), which overrides the original derived maximum path delay constraint.

Asynchronous Paths. You can also use the set_max_delay and set_min_delay commands to constrain asynchronous paths across different frequency domains. For example,

```
dc_shell-xg-t> set_max_delay 17.1 -from [get_clocks clk1] \
     -to [get_clocks clk2]

dc_shell-xg-t> set_max_delay 23.5 -from
     [get_clocks clk2] -to [get_clocks clk3]

dc_shell-xg-t> set_max_delay 31.6 -from [get_clocks clk3] \
     -to [get_clocks clk1]
```

Setting Multicycle Paths. The multicycle path condition is appropriate when the path in question is longer than a single cycle or when data is not expected within a single cycle. Use the

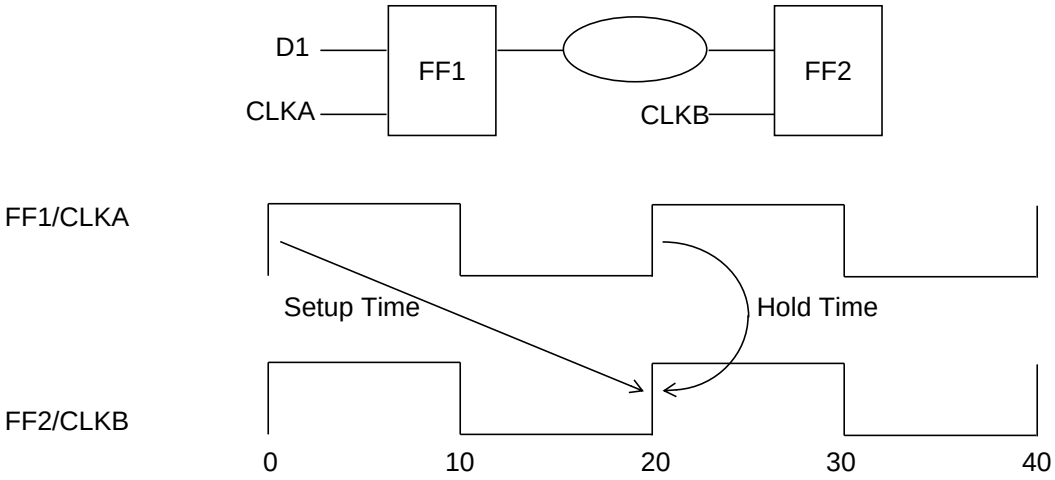
`set_multicycle_path` command to specify the number of clock cycles Design Compiler should use to determine when data is required at a particular endpoint.

You can specify this cycle multiplier for setup or hold checks. If you do not specify the `-setup` or `-hold` option with the `set_multicycle_path` command, Design Compiler applies the multiplier value only to setup checks.

By default, setup is checked at the next active edge of the clock at the endpoint after the data is launched from the startpoint (default multiplier of 1). Hold data is launched one clock cycle after the setup data but checked at the edge used for setup (default multiplier of zero).

Figure 7-6 shows the timing relationship of setup and hold times.

Figure 7-6 Setup and Hold Timing



The timing path starts at the clock pin of FF1 (rising edge of CLKA) and ends at the data pin of FF2. Assuming that the flip-flops are rising-edge-triggered, the setup data is launched at time 0 and

checked 20 time units later at the next active edge of CLKB at FF2. Hold data is launched one (CLKA) clock cycle (time 20) and checked at the same edge used for setup checking (time 20).

The `-setup` option of the `set_multicycle_path` command moves the edge used for setup checking to before or after the default edge. For the example shown in Figure 7-6,

- A setup multiplier of zero means that Design Compiler uses the edge at time zero for checking
- A setup multiplier of 2 means that Design Compiler uses the edge at time 40 for checking

The `-hold` option of the `set_multicycle_path` command launches the hold data at the edge before or after the default edge, but Design Compiler still checks the hold data at the edge used for checking setup. As shown in Figure 7-6 (assuming a default setup multiplier),

- A hold multiplier of 1 means that the hold data is launched from CLKA at time 40 and checked at CLKB at time 20
- A hold multiplier of -1 means that the hold data is launched from CLKA at time 0 and checked at CLKB at time 20

To undo a `set_multicycle_path` command, use the `reset_path` command with similar options.

Using Multiple Timing Exception Commands. A specific timing exception command refers to a single timing path. A general timing exception command refers to more than one timing path. If you execute more than one instance of a given timing exception command, the more specific commands override the more general ones.

The following rules define the order of precedence for a given timing exception command:

- The highest precedence occurs when you define a timing exception from one pin to another pin.
- A command using only the `-from` option has a higher priority than a command using only the `-to` option.
- For clocks used in timing exception commands, if both `-from` and `-to` are defined, they override commands that share the same path defined by either the `-from` or the `-to` option.

This list details the order of precedence (highest at the top) defined by these precedence rules:

1. *command* -from *pin* -to *pin*
2. *command* -from *clock* -to *pin*
3. *command* -from *pin* -to *clock*
4. *command* -from *pin*
5. *command* -to *pin*
6. *command* -from *clock* -to *clock*
7. *command* -from *clock*
8. *command* -to *clock*

For example, in the following command sequence, paths from A to B are treated as two-cycle paths because specific commands override general commands:

```
dc_shell-xg-t> set_multicycle_path 2 -from A -to B
dc_shell-xg-t> set_multicycle_path 3 -from A
```

The following rules summarize the interaction of the timing exception commands:

- General `set_false_path` commands override specific `set_multicycle_path` commands.
- General `set_max_delay` commands override specific `set_multicycle_path` commands.
- Specific `set_false_path` commands override specific `set_max_delay` or `set_min_delay` commands.
- Specific `set_max_delay` commands override specific `set_multicycle_path` commands.

Setting Area Constraints

The `set_max_area` command specifies the maximum area for the current design by placing a `max_area` attribute on the current design. Specify the area in the same units used for area in the technology library.

For example, to set the maximum area to 100, enter

```
dc_shell-xg-t> set_max_area 100
```

Design area consists of the areas of each component and net. The following components are ignored when Design Compiler calculates design area:

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

Cell (component) area is technology dependent; Design Compiler obtains this information from the technology library.

When you specify both timing and area constraints, Design Compiler attempts to meet timing goals before area goals. To prioritize area constraints over total negative slack (but not over worst negative slack), use the `-ignore_tns` option when you specify the area constraint.

```
dc_shell-xg-t> set_max_area -ignore_tns 100
```

To optimize a small area, regardless of timing, remove all constraints except for maximum area. You can use the `remove_constraint` command to remove constraints from your design. Be aware that this command removes *all* optimization constraints from your design.

Verifying the Precompiled Design

Before compiling your design, verify that

- The design is consistent

  Use the `check_design` command to verify design consistency. For information about the `check_design` command, see “Checking for Design Consistency” on page 11-3.

- The attributes and constraints are correct

  Design Compiler provides many commands for reporting the attributes and constraints. For information about these commands, see “Analyzing Design Problems” on page 11-8 and “Analyzing Timing” on page 11-9.

8

Using Design Compiler Topographical Technology

Topographical technology enables you to accurately predict post-layout timing, area, and power during RTL synthesis without the need for wireload 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. This new technology is built in as part of the DC Ultra feature set and is available only by using the `compile_ultra` command in topographical mode.

Design Compiler topographical mode requires a DC Ultra license and a DesignWare license. A Milkyway-Interface license is also necessary if the Milkyway flow is used.

This chapter includes the following sections:

- Starting Design Compiler Topographical Mode
- Inputs and Outputs in Design Compiler Topographical Mode
- Specifying Libraries
- Using TLUPlus for RC Estimation
- Multivoltage Designs
- Using Floorplan Physical Constraints
- Performing an Incremental Compile
- DFT Insertion Flow in Topographical Mode
- Power Compiler Flow in Topographical Mode
- Performing Top-Level Design Stitching
- Supported Commands, Command Options, and Variables
- Sample Scripts

Topographical Technology

Topographical technology includes these principal features:

- Topographical technology leverages the Synopsys physical implementation solution to drive an accurate timing prediction within the RTL synthesis engine.Timing and synthesis are based on a virtual layout analysis. This ensures excellent correlation with the physical design.
- Topographical technology predicts and uses real net capacitances.
  - Wire load models are not needed (they are ignored if present).
  - Capacitances are updated as synthesis progresses.
- Design Compiler topographical technology is shared with IC Compiler.
- Topographical technology supports multivoltage designs.
- Topographical technology supports all synthesis flows, including the following:
  - Test-ready compile flow (basic scan and DFT MAX adaptive scan)
  - Clock-gating flow
  - Register retiming
- The feature supports a top-down compile flow (recommended).

Note:

When you use the `insert_buffer` command and `remove_buffer` command described in “Editing Designs” on page 5-43, the `report_timing` command does not report placement-based timing for the edited cells. To update timing, run the `compile_ultra -inc` command.

Starting Design Compiler Topographical Mode

To use the Design Compiler topographical features, you must run dc_shell in topographical mode. At the prompt, enter

```
dc_shell-xg-t -topographical
```

You can abbreviate the switch to as short as `-to`. In topographical mode, the dc_shell command-line prompt appears as

```
dc_shell-topo>
```

To query the mode, run the `shell_is_in_topographical_mode` command. The command returns 1 if the topographical mode is active; otherwise it returns 0. The topographical mode is supported only in XG mode and is Tcl based.

When you run the `compile_ultra` command in this mode, the Design Compiler topographical features are automatically used. All `compile_ultra` command options are supported.

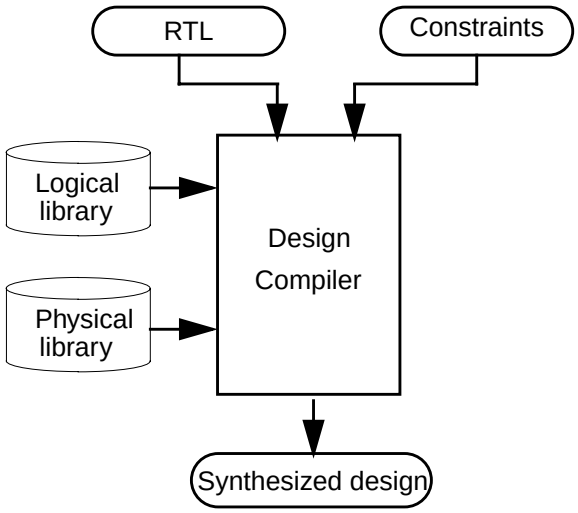
Note:

In the Z-2007.03 release, only a subset of dc_shell commands, command options, and variables is supported in topographical mode. For more information, see “Supported Commands, Command Options, and Variables.”

Inputs and Outputs in Design Compiler Topographical Mode

Figure 8-1 shows the inputs and outputs in Design Compiler topographical mode. The sections that follow provide more information.

Figure 8-1 Inputs and Outputs in Design Compiler Topographical Mode



Required Inputs in Design Compiler Topographical Mode

The following inputs are required to run topographical synthesis:

- RTL design or an existing gate-level netlist
- Timing and optimization constraints

  Note that if you specify wire load models, the tool ignores them.

- Liberty format logic library (.lib and .db)

- Physical library
  - Milkyway format (the default)
  - .pdb library format.

Outputs From Topographical Synthesis

You can output the synthesized design in .ddc format, Milkyway format, or as a Verilog netlist. Be aware of the following conditions when choosing an output format.

Design in .ddc or Milkyway Format

- Contains back-annotated net delays and parasitic data based on Design Compiler topographical technology.
- Does not contain any placement data.

Design as a Verilog Netlist

- Recommended only if you intend to use a third-party tool for place and route operations.
- Back-annotated net delays and parasitics estimated on the basis of topographical technology are lost.
- Delay and parasitic data are saved by using
  - `write_sdf` command
  - `write_parasitics` command

Specifying Libraries

In topographical mode, Design Compiler requires both logical libraries and physical libraries.

Specifying Logical Libraries

Design Compiler topographical mode uses the same logical libraries as the Design Compiler wire load mode. Use the following commands to set up logical libraries:

```
set search_path "search_path ./libraries"
set link_library "* max_lib.db"
set target_library "max_lib.db"
```

For more information on logical libraries, see “Working With Libraries” on page 4-1.

Note:

Because multivoltage designs use power domains, these designs usually require that certain library cells are marked as always-on cells and certain library cell pins are marked as always-on library cell pins. These always-on attributes are necessary to establish any always-on relationships between power domains. For more information, see the *Power Compiler User Guide, Chapter 11, Using Multivoltage Designs*.

Specifying Physical Libraries

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

The following steps provide an overview of how to create a Milkyway library; for more information, see “Using a Milkyway Database” on page 10-1.

1. Use the `create_mw_lib` command to create the Milkyway library. For example,

   ```
   create_mw_lib -technology $mw_tech_file \
                 -mw_reference_library \
                 $mw_reference_library $mw_lib_name
   ```

2. Use the `open_mw_lib` command to open the Milkyway library that you created. For example, enter

   ```
   open_mw_lib $mw_lib_name
   ```

3. (Optional) Use the `set_tlu_plus` command to attach TLU+ files. For example,

   ```
   set_tlu_plus_files -max_tluplus $prs_tlu_file \
            -tech2itf_map $prs_map_file
   ```

   For more information, see “Using TLUPlus for RC Estimation” on page 8-10.

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

   ```
   open_mw_lib $mw_lib_name
   set_tlu_plus_files -max_tluplus $prs_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`

- `check_tlu_plus_files`
- `write_mw_lib_files`
- `set_mw_lib_references`

Note:

You can also use the `create_mw_design` variable to create the Milkyway library; however, it is recommended that you use the `create_mw_lib` command. If you are using the `create_mw_design` variable, you must set the reference library and design library by using the `mw_reference_library` and `mw_design_library` variables as follows:

```
set mw_reference_library "./milkyway/max_lib"
set mw_design_library my_design.mw
create_mw_design -tech_file ./max_lib.tf \
     -max_tluplus rc_worst.tlup \
     -tf2itf_map rcxt_tf2itf_map
```

You can also use the .pdb format to specify physical libraries. In this case, use the following commands:

```
set use_pdb_lib_format true
set physical_library "max_lib.pdb"
```

Using TLUPlus for RC Estimation

If TLUPlus files are available, you can use them for RC estimation in Design Compiler topographical mode. It is not mandatory to specify TLUPlus files as long as resistance and capacitance models are present in the vendor technology physical library. However, TLUPlus files provide more accurate capacitance and resistance data, thereby improving correlation with back-end results.

You use the `set_tlu_plus_files` command to specify TLUPlus files. In addition, use the `-tech2itf_map` option to specify a map file, which maps layer names between the Milkyway technology file and the process Interconnect Technology Format (ITF) file. For example,

```
set_tlu_plus_files -max_tlu_plus -tech2itf_map design.map
```

For more information on the map file, see the Milkyway documentation. To ensure that you are using the TLUPlus files, check the `compile_ultra` log for the following message:

```
****************************************************
Information: TLU Plus based RC computation is enabled.
(RCEX-141)
****************************************************
```

Multivoltage Designs

Design Compiler topographical mode supports the use of multivoltage designs, multivoltage design features, and related commands. In particular, power domains can be defined and level shifter and isolation cells can be used as needed to adjust voltage differences between power domains and to isolate shut-down power domains.

A power domain is defined as a logic grouping of one or more hierarchical blocks in a design that share the following:

- Primary voltage states or voltage range (that is, the same operating voltage)
- Power net hookup requirements
- Power-down control and acknowledge signals (if any)

- Power switching style
- Same process, voltage, and temperature (PVT) operating condition values (all cells of the power domain except level shifters)
- Same set or subset of nonlinear delay model (NLDM) target libraries

Principle power domain commands include `create_power_domain`, `infer_power_domain`, `connect_power_domain`, `create_power_net_info`, `connect_power_net_info`, as well as a number of other commands.

Note:

Power domains are not voltage areas. A power domain is a grouping of logic hierarchies, whereas the corresponding voltage area is a physical placement area into which the cells of the power domain's hierarchies are placed. This correspondence is not automatic. You are responsible for correctly aligning the hierarchies to the voltage areas.

Level shifter cells are needed to connect drive and load pins operating at different voltages across the power domains. They are used to step up or step down the voltage from their input side to their output side. These cells are modeled either as simple buffers or as buffer cells with an enable pin. The first type of cell is referred to as a buffer-type level shifter and the second type as an enable-type level shifter. Enable-type level shifters are used when a power domain must be selectively shut down for some duration of the design's operation.

Isolation cells are used when a power domain must be selectively isolated from other power domains at certain times during the design's operation but stepping the voltage from the input side to the output side of the cell is not necessary. Their function is equivalent to that of an enable-type level shifter that connects two power domains operating at the same voltage. (Because an enable-type level shifter is basically a buffer, it can connect drive and load pins operating at the same voltage as well as pins not operating at the same voltage.)

Level shifters and isolation cells are not usually part of the original design description and must be inserted during the logic synthesis flow. Different methods are used to add these special cells to a design. Buffer-type level shifters can be inserted either automatically as part of running the `compile_ultra` (or `compile`) command or manually by using the `insert_level_shifters` command. Isolation cells and enable-type level shifters can be instantiated at the RTL level of the design description or inserted manually by using the `insert_isolation_cell`, which can be easier to carry out.

The compile flow for multivoltage designs can involve using a number of commands not discussed here. It is recommended that you refer to the *Power Compiler User Guide, Chapter 11, Using Multivoltage Designs* for more information on how to handle multivoltage designs.

Using Floorplan Physical Constraints

In topographical mode, Design Compiler can read physical constraints (floorplan information) from a Tcl-based script of physical commands. The principal reason for using floorplan constraints in topographical mode is to accurately represent the placement area

and to improve timing correlation with the post-place-and-route design. Also, using floorplan information is recommended if the floorplan is very complicated, for example, if the design contains many macros or a large number of blockages and keepout regions, or if the core area contains unusual shapes.

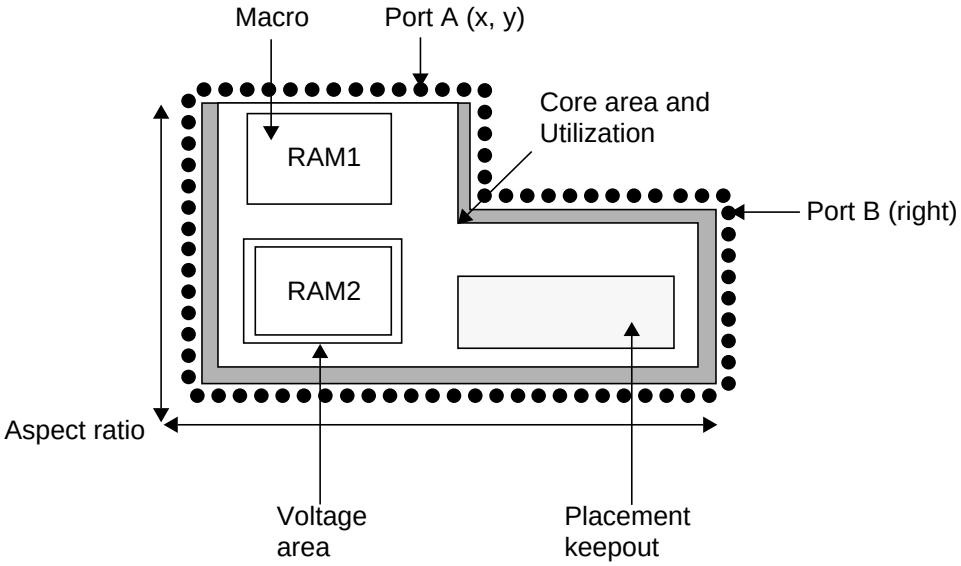
Physical information can be provided to Design Compiler topographical mode in the following ways:

- It can be derived from Jupiter XT floorplan data. See “Deriving Physical Constraints From Jupiter XT” on page 8-21.
- It can be extracted from an existing Design Exchange Format (DEF) file. See “Extracting Physical Constraints From a Design Exchange Format File” on page 8-21.
- It can be created manually, described next in “Supported Physical Constraints.

Supported Physical Constraints

As shown in Figure 8-2, only the high-level physical constraints that determine core area and shape, port location, macro location and orientation, voltage areas, and placement blockages are supported.

Figure 8-2 Supported Physical Constraints



The following sections describe the commands you can use to explicitly define the physical constraints when you cannot obtain this information from a Jupiter XT run (described in Figure on page 8-21) or from a DEF file (described in “Extracting Physical Constraints From a Design Exchange Format File” on page 8-21.

The commands described in the following sections allow you to specify relative constraints or exact constraints. Use the relative constraint specifications when your floorplan information is not exact but you have reliable, approximate information regarding port locations and placement area.

If you do not provide any physical constraints or read them from Jupiter XT script or a DEF file, the tool uses the following default physical constraints:

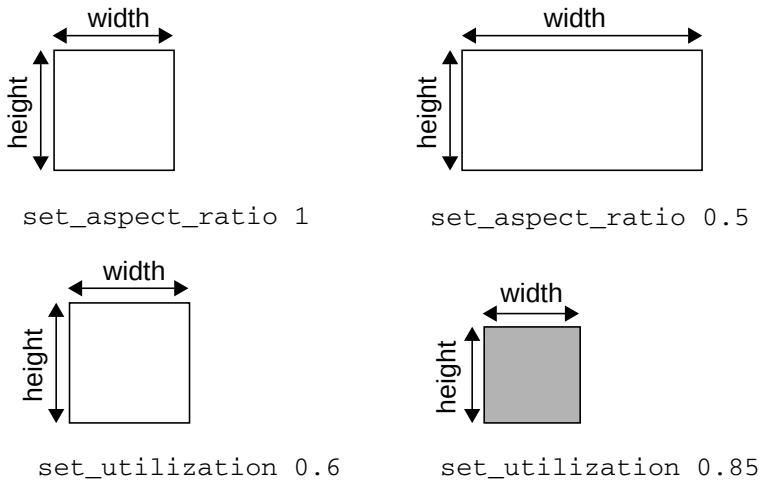
- Aspect ratio of 1.0
- Utilization of 0.6

- Virtual placement of cells and pins in optimal locations to minimize wire length

Defining Core Area And Shape

You can define relative core area and shape by using the `set_aspect_ratio` and `set_utilization` commands as shown in Figure 8-3. 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 8-3 Defining Relative Core Area and Shape



Note:

For multivoltage designs, you define core area and shape by using the `set_placement_area` command.

You can define the exact rectangular block shape by setting the following command:

```
set_placement_area -coordinate {X1 Y1 X2 Y2}
```

where `{X1 Y1 X2 Y2}` specifies the lower left and upper right coordinates of the core area.

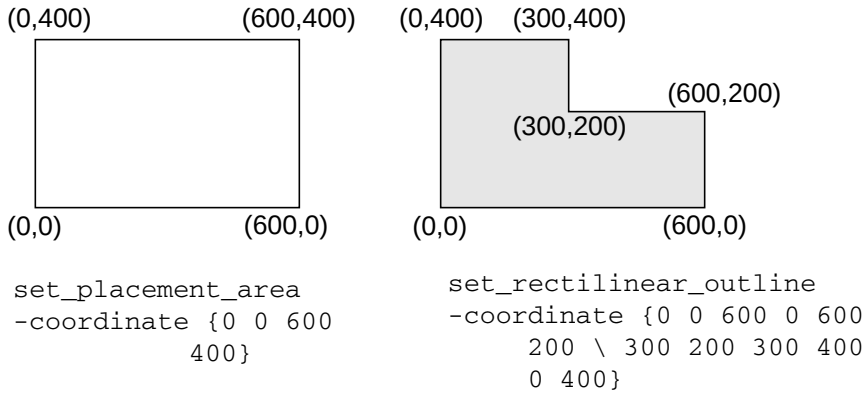
You can define the exact rectilinear block shape by using the following command:

```
set_rectilinear_outline -coordinate {X1 Y1 X2 Y2
...}
```

where the points of the rectilinear shape are not closed; that is, the first and last points are not the same, and the points are in counterclockwise order.

Figure 8-4 shows how you define exact core area and shape by using the `set_placement_area` command and the `set_rectilinear_outline` command.

Figure 8-4 Defining Exact Core Area And Shape



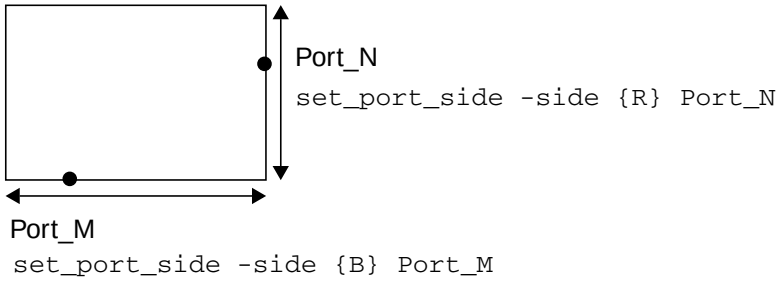
Defining Port Location

You can define relative port locations by using the following conmmand:

```
set_port_side port_name -side {L|R|T|B}
```

Figure 8-5 shows how you define port sides by using the `set_port_side` command; use the command to specify which side a port is placed. Valid sides are left (L), right (R), top(T), or bottom(B). A port can be placed at any location along the specified side. This constraint is used only for a design with a rectangular core area.

Figure 8-5 Defining Relative Port Sides



You can define exact port locations by setting the following command:

```
set_port_location port_name -coordinate {x y}
```

where `-coordinate` is the lower left point of the port shape. Figure 8-6 on page 8-20 shows how you can use the `set_port_location` command to define the exact port locations.

Defining Macro Location And Orientation

You can define exact macro locations by setting the following command:

```
set_cell_location cell_name -coordinate {x y}
-orientation {N|S|E|W|FN|FS|FE|FW} -fixed
```

where `-coordinate` 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 listed in the following table.

Orientation	Rotation
N (default)	Nominal orientation, 0 degree rotation (north)
S	180 degrees (south)
E	270 degrees counterclockwise (east)
W	90 degrees counterclockwise (west)
FN	Reflection in the y-axis (flipped north)
FS	180 degrees, followed by reflection in the y-axis (flipped south)
FE	270 degrees counterclockwise, followed by reflection in the y-axis (flipped east)
FW	90 degrees counterclockwise, followed by reflection in the y-axis (flipped west)

Figure 8-6 on page 8-20 shows how you use the `set_cell_location` command to define exact port locations.

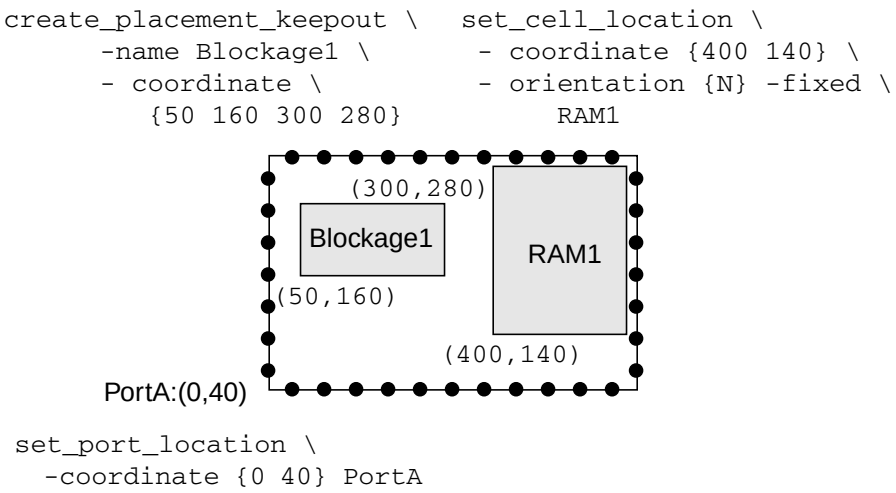
Defining Placement Keepout

You define the exact placement keepouts (blockages) by setting the following command.

```
create_placement_keepout -name name_string
-coordinate {llx lly urx ury -type {soft|hard}
```

where the `-name name_string` specifies the name of the keepout (blockage) and the `-type` option specifies the type of keepout to be created (soft or hard). Figure 8-6 on page 8-20 shows how you use the `create_placement_keepout` command to define placement blockages.

Figure 8-6 Defining Exact Ports, Macros, and Blockages



Defining Voltage Area

To define voltage area, use the `create_voltage_area` command. Note that the guard band option is not used in topographical mode. In addition to the `create_voltage_area` command, the following commands are supported:

- `remove_voltage_area`
- `report_voltage_area`

Deriving Physical Constraints From Jupiter XT

You can generate a Tcl-based script of physical commands from Jupiter XT floorplan data by using the `derive_physical_constraints` command (in Jupiter XT). After you have derived the physical constraints, use the `source` command to apply the command script in topographical mode.

Note:

If voltage areas are required (multivoltage designs), they are usually defined in Jupiter XT as part of the floorplan. However, if the voltage areas are missing, you can use the `create_voltage_area` command.

For more information on using the `derive_physical_constraints` command, see the man page and Jupiter XT documentation.

Extracting Physical Constraints From a Design Exchange Format File

To extract floorplan information from a Design Exchange Format (DEF) file, use the `extract_physical_constraints` command. The command extracts physical information from the specified DEF file and applies these constraints to the design. For example, the following command extracts physical constraints from the in.def DEF file and writes the constraints to the out.tcl file:

```
dc_shell-xg-t > extract_physical_constraints in.def \
                        -output output.tcl
```

Table 8-1 lists the physical constraints extracted by the `extract_physical_constraints` command and included in the output file.

Table 8-1 Physical Constraints

Physical information	Commands
Core area	`set_placement_area -coordinate {x1 y1 x2 y2}`
Port location	`set_port_location -coordinate {x y}`
Macro location or orientation	`set_cell_location -coordinate {x y} \` `-orientation {N|S|E|W|FN|FS|FE|FW} -fixed`
Placement keepout	`create_placement_keepouts -name` *`name_string`* `\` `-coordinate {x1 y1 x2 y2} -type hard|soft`

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.

By default, the `extract_physical_constraints` command runs in incremental mode. That is, if you use the command to process a different DEF file, the command preserves existing physical annotations on the design. Any conflicts are resolved in the following manner:

- 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.

Use the `-no_incremental` option to disable incremental mode.

Matching Names of Macros and Ports

By default, when the `extract_physical_constraints` command applies physical constraints in topographical mode, it uses an intelligent name matching algorithm to match macros and ports in the DEF file with macros and ports in memory. The command uses the intelligent name matching capability when it does not find an exact match.

The `extract_physical_constraints` command usually reads from DEF files generated from a netlist that has a different naming style from the netlist in memory. Therefore, there could be name mismatches caused by automatic ungrouping and the `change_names` command; typically, hierarchy separators and bus notations are sources of these mismatches.

For example, automatic ungrouping by the `compile_ultra` command followed by `change_names` might result in the forward slash (/) separator being replaced with an underscore (_) character. Therefore, a macro named *a/b/c/macro_name* in the RTL might be named *a/b_c_macro_name* in the mapped netlist, which is the input to the back-end tool. When extracting physical constraints from the DEF file, the `extract_physical_constraints` command automatically resolves these name differences by using an intelligent name matching algorithm.

To disable intelligent name matching, you can use the `-exact` option of the `extract_physical_constraints` command. The option allows you to specify that the objects in the netlist in memory be matched exactly with the corresponding objects in the DEF file. When you use the `-verbose` option of the `extract_physical_constraints` command, the tool displays an informational message:

```
Information: Fuzzy match cell %s in netlist with instance
%s in DEF.
```

By default, the following characters are considered equivalent:

- Hierarchical separators { / _ . }

  For example, a cell named *a.b_c/d_e* is automatically matched with the string *a/b_c.d/e* in the DEF file.

- Bus notations { [ ] __ ( ) }

  For example, a cell named *a [4] [5]* is automatically matched with the string *a_4__5_* in the DEF file.

To define the rules used by the intelligent name matching algorithm, use the `set_fuzzy_query_options` command. The syntax is

```
set_fuzzy_query_options
     [-hierarchical_separators char_list]
     [-bus_name_notations notation_list]
     [-class list_of_object_class]
     [-reset]
     [-show]
```

Use options to the `set_fuzzy_query_options` command as follows:

- `-hierarchical_separators` to define a list of equivalent hierarchical separators. Each item of the *`char_list`* list can only be a single character. When this option is used, there must be at least 2 items in the list.

  The default list value is `{/ _ .}`. Because "/" is the default hierarchical separator for the in-memory netlist, a warning is issued if "/" is not in the hierarchical separator list.

  The following characters are not supported as hierarchical separators: 0-9, a-z, A-Z, *, ?, \, +, ^, [,], (,), <, >, {, }.

- `-bus_name_notations` to define a list of equivalent bus notation characters. The length of each item in the *`notation_list`* list must be 2. When this option is used, there must be at least 2 items in the list. The first character of each item is the opening bus notation character, and the second character of each item is the closing bus notation character.

  The default list value is `{[] __ ()}`. Because "%s[%d+]" is the default in-memory bus naming style, a warning is issued if "[]" is not in the bus name notation list.

  The following characters are not supported as opening or closing bus name characters: 0-9, a-z, A-Z, *, ?, \, +, ^, /.

  Bracket bus notations must be paired. For example, `{[)}` is not a properly paired bracket pair so it is not supported. For bus names that do not include brackets notations, the opening and closing bus name characters must be the same. For example, `{-_}` is not a valid bus name notation.

- `-class` to spcify the object class to which the Intelligent Name Matching rules will be applied. Only cell, port, pin, and net classes are supported.

  The default list value is `{cell pin port net}`.

- `-reset` to reset options of the `set_fuzzy_query_options` command to the default. This option will be exclusive of other options except for `-show`.

- `-show` to report the current Intelligent Name Matching rules. When this option is used with other options that define new rules, the new rules will be set first and then displayed.

The following example has the same effect as using the `-reset` option:

```
dc_shell-xg-t > set_fuzzy_query_options \
                  -hierarchical_separators {/ _ . @} \
                  -bus_name_notations {[] __ ()} \
                  -class {cell pin port net}
```

Saving Physical Constraints

After you run `compile_ultra`, you might want to output the physical information annotated on the design netlist to a constraints file by using the `write_physical_constraints` command.

A physical constraints file can be checked, edited, and sourced for later use; that is, you can read the physical constraints from the file into Design Compiler in the topographical mode to further optimize a synthesized .ddc design or netlist. To report the physical constraints, use the `report_physical_constraints` command.

You can also save the design in .ddc format by using the `write_ddc` command. In addition to physical constraints that you have specified, the tool also saves virtual placements. The following physical constraints are saved:

- `set_placement_area`
- `set_rectilinear_outline`
- `set_port_location`
- `set_cell_location`
- `create_placement_keepout`
- `create_voltage_area`
- `set_utilization`
- `set_aspect_ratio`
- `set_port_side`

The tool applies these constraints automatically when you read in the .ddc file by using the `read_ddc` command in a new topographical mode session. Note that these physical constraints can be retrieved in topographical mode only; Design Compiler wire load mode, JupiterXT, or IC Compiler cannot retrieve this information.

Performing an Incremental Compile

The `-incremental` option of the `compile_ultra` command allows you to employ a second-pass, incremental compile strategy. The main goal for `compile_ultra -incremental` is to enable topographical-based optimization for post-topographical-based

synthesis flows such as retiming, design-for-test (DFT), DFT MAX, and minor netlist edits. The primary focus in Design Compiler topographical mode is to maintain QoR correlation; therefore, only limited changes to the netlist can be made.

Use the incremental compile strategy to meet the following goals:

- Improve design QoR
- Fix the netlist after manual netlist edits or constraint changes
- Fix the netlist after various synthesis steps have been performed on the compiled design, for example, after `insert_dft` or register retiming
- Control design rule fixing by using the `-no_design_rule` or `-only_design_rule` option in combination with the `-incremental` option

Note that applying `compile_ultra -incremental` to a topographical netlist results in placement-based optimization only. This compile should not be thought of as an incremental mapping.

When using the `-incremental` option, keep the following in mind:

- Marking library cells with the `dont_use` attribute does not work for an incremental flow when it is applied to a topographical netlist. Make sure to apply any `set_dont_use` attributes before the first pass of a topographical-based synthesis.
- If you intend to use boundary optimization and scan insertion, apply them to the first pass of a topographical-based synthesis.
- Avoid significant constraint changes in the incremental pass.

  Note:

  Physical constraint changes are not supported.

DFT Insertion Flow in Topographical Mode

Topographical mode supports test synthesis. The flow is similar to the one in Design Compiler wire load mode except that the `insert_dft` command is used for stitch-only. Topographical mode supports basic scan and adaptive scan.

To perform scan insertion within topographical mode, use a script similar to the following:

```
dc_shell-xg-t -topo
read_ddc top_elaborated.ddc
insert_clock_gating
source top_constraint.sdc
source physical_constraints.tcl
compile_ultra -scan

## Provide DFT specifications

create_test_protocol
dft_drc
preview_dft
insert_dft
dft_drc

compile_ultra -incremental -scan
write_scan_def -output dft.scandef
```

For more information on scan insertion, see the DFT Compiler documentation.

Power Compiler Flow in Topographical Mode

In topographical mode, Design Compiler can perform power correlation. The `set_power_prediction` command enables Design Compiler to correlate post-synthesis power numbers with

those after place and route. Optionally, you can use the `-ct_references` option to specify clock tree references to improve correlation.

In addition, power prediction is on by default if any gate-level power optimizations are enabled by using the following commands:

- `set_max_dynamic_power`
- `set_max_leakage_power`
- `set_max_total_power`
- `set_power_gating_style`
- `compile_ultra -gate_clock`

To perform power correlation within topographical mode, run a script similar to the following:

```
read_ddc top_elaborated.ddc

## Define clock gating style

set_clock_gating_style -pos {integrated} \
    -control_point before
insert_clock_gating
propagate_constraints -gate_clock
hookup_testports
source top_constraint.sdc
source physical_constraints.tcl

## Set power optimization constraints

set_max_dynamic_power 0 mW
compile_ultra -scan
write_ddc -output synthesized.ddc
```

For more information on power correlation, see the Power Compiler documentation.

Performing Top-Level Design Stitching

In topographical mode, the `-top` option of the `compile_ultra` command enables you to stitch compiled physical blocks into the top-level design.

The recommended strategy is a top-level compile flow. However, you might need to use the top-level design stitching capability in certain cases. For example,

- The chip top level might contain unmapped glue logic that needs to be mapped, timed, and optimized.
- Paths that traverse multiple physical blocks might require the complete timing context at the chip level.

The `-top` option of the `compile_ultra` option enables you to perform chip-level mapping, optimization, and stitching. Top-level design stitching in topographical mode is similar to the functionality provided by the `compile -top` command except that in topographical mode, the tool can map any unmapped top-level glue logic. The tool automatically back-annotates the top-level design with virtual placement-based delays of subblocks while preserving the logic structure of the subblocks.

When you perform top-level design stitching, keep the following points in mind:

- Each subblock must be a fully mapped netlist. Otherwise, the following error message is issued:

  ```
  Error: Using compile -top with unmapped logic (OPT-1304)
  ```

  The subblock must have been compiled in topographical mode. Otherwise, the following warning message is issued:

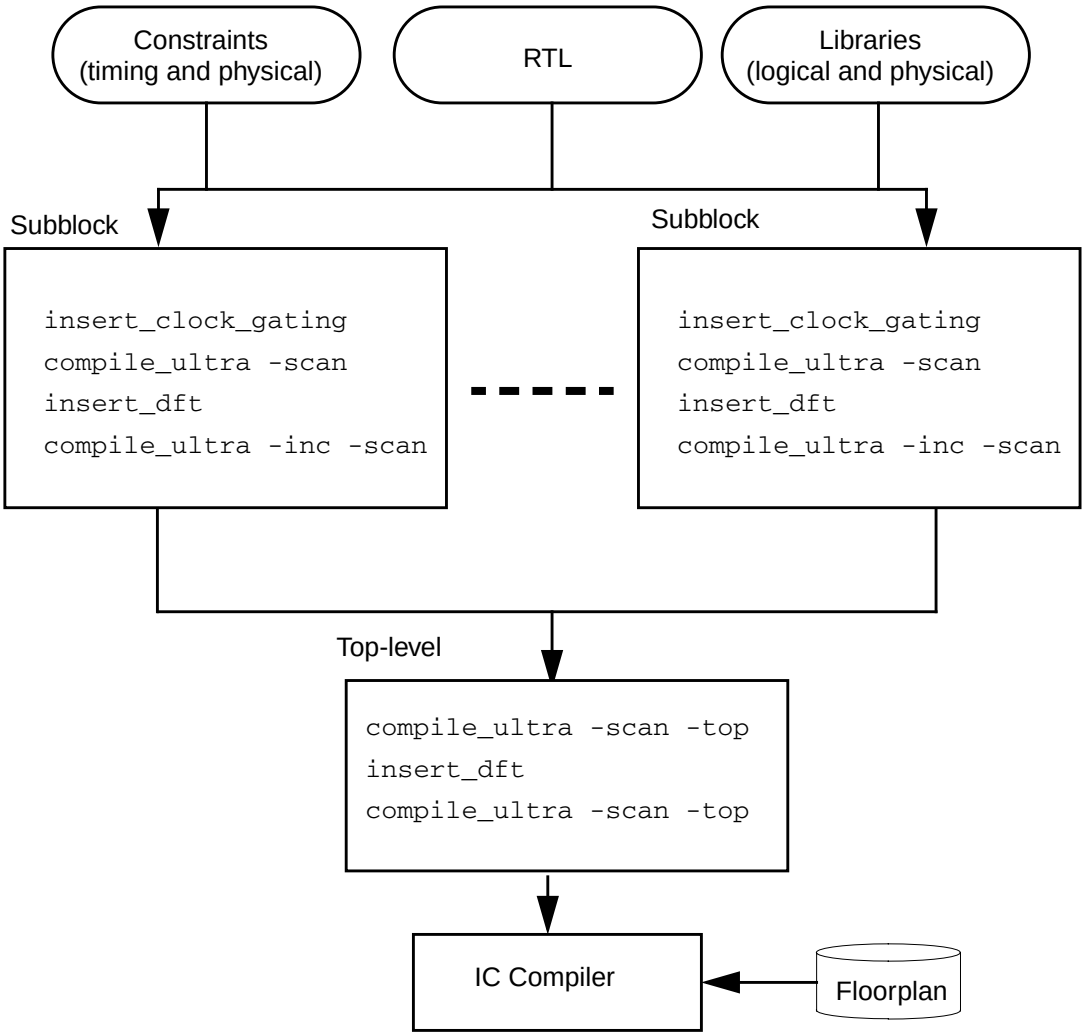
```
Warning: Subblock '%s' was not created using DC
topographical technology. (OPT-1422)
```

- If the subblock is to be included in the top-level scan chain, use the `insert_dft` command to insert scan chains in the subblock.

Steps in the Top-Level Design Stitching Flow

Figure 8-7 shows the steps in the top-level design stitching flow.

Figure 8-7 Top-Level Design Stitching Flow



As shown in Figure 8-7, the top-level design stitching flow requires these steps:

1. Set up the design and libraries.
2. Read in the top-level design.
3. Compile each subblock by performing the following steps:
   a. Set the current design to the sub-block.
   b. Apply timing constraints and power constraints.
   c. (Optional) Provide physical constraints as described in “Using Floorplan Physical Constraints” on page 8-13.
   d. Perform clock gating by using the `insert_clock_gating` command.
   e. Perform test-ready compile by using the `compile_ultra -scan` command.
   f. If the subblock is to be included in the top-level scan chain, use the `insert_dft` command to insert scan chains.
   g. Run the `compile_ultra -scan -incremental` command.
   h. Use the `report_timing` command to check timing.
4. Set the current design to the top-level design, link the design, and apply the top-level timing constraints.
5. Perform clock gating at the top level by using the `insert_clock_gating` command.
6. Run the `compile_ultra -scan -top` command.

   You use these options because top-level glue logic can contain both unmapped sequential cells and combinational cells. The `-top` option maps the top-level logic and back-annotates the

top-level design with timing information from the lower-level blocks.The `-scan` option enables the tool to map sequential cells to appropriate scan flip-flops.

7. Run the `insert_dft` command to insert scan chains at the top level, followed by `compile_ultra -scan -top` to map any additional unmapped logic that might have been introduced.

Note:

The `compile_ultra -top` command performs boundary optimizations that could change interface logic between blocks; therefore, the back-annotated timing of subblocks might be updated even though the logic configuration hasn't changed.

The top-level design stitching feature does not support clock tree estimation; therefore, the `report_power` command cannot report any correlated power at the top level.

Supported Commands, Command Options, and Variables

In topographical mode, if an unsupported command or variable is encountered in a script, an error message is issued; however, the script *continues*.

For example,

```
ERROR: Command set_wire_load_mode is not supported in DC
Topographical mode. (OPT-1406)

WARNING: Variable hdlin_enable_vpp is not supported in DC
Topographical mode and will be ignored. (OPT-1407)
```

For unsupported command options (except wire load model options), an error message is issued, and the script stops executing. Wire load model options are ignored, and the script continues.

Also, variables are read-only in topographical mode as indicated in error messages such as the following:

```
ERROR: Can’t set hdlin_enable_vpp: it is a read-only
variable. (CMD-024)
```

Look for additional supported commands, command options, and variables in subsequent releases. You should check your scripts and update them as needed. See the appropriate man pages to determine the current status in topographical mode.

Sample Scripts

The scripts shown in Example 8-1 and Example 8-2 suggest how to set up your libraries and use Design Compiler topographical technology to synthesize your design.

Example 8-1 Default Flow for Topographical Synthesis

```
dc_shell-xg-t -topo

set search_path "search_path ./libraries"
set link_library "* max_lib.db"
set 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_ddc top_elaborated.ddc
source top_constraint.sdc
compile_ultra
report_timing
change_names -rules verilog -h
write -format ddc -hierarchical \
    -output top_synthesized.ddc
write -f verilog -h -o netlist.v
write_sdf sdf_file_name
write_parasitics -o parasitics_file_name
write_sdc sdc_file_name
```

Example 8-2 Flow With Physical Constraints

```
dc_shell-xg-t -topo

set search_path "search_path ./libraries"
set link_library "* max_lib.db"
set target_library "max_lib.db"

read_ddc top_elaborated.ddc
source top_constraint.sdc
extract_physical_contraints def_file_name
compile_ultra
report_timing
change_names -rules verilog -h
write -format ddc -hierarchical \
    -output top_synthesized.ddc
write -f verilog -h -o netlist.v
write_sdf sdf_file_name
write_parasitics -o parasitics_file_name
write_sdc sdc_file_name
write_physical_constraints -o \
       phys_cstr_file_name.tcl
```

9

Optimizing the Design

Optimization is the Design Compiler synthesis step that maps the design to an optimal combination of specific target library cells, based on the design's functional, speed, and area requirements. You use the `compile_ultra` command or the `compile` command to compile a design. Design Compiler provides options that enable you to customize and control optimization. Several of the many factors affecting the optimization outcome are discussed in this chapter. For detailed information, see the *Design Compiler Reference Manual:Optimization and Timing Analysis*.

This chapter has the following sections:

- Understanding the Compile Cost Function
- Performing Design Exploration
- Performing Design Implementation

The Optimization Process

Design Compiler performs the following three levels of optimization:

- Architectural optimization
- Logic-level optimization
- Gate-level optimization

The following sections describe these processes.

Architectural Optimization

Architectural optimization works on the HDL description. It includes such high-level synthesis tasks as

- Sharing common subexpressions
- Sharing resources
- Selecting DesignWare implementations
- Reordering operators
- Identifying arithmetic expressions for data-path synthesis (DC Ultra only).

Except for DesignWare implementations, these high-level synthesis tasks occur only during the optimization of an unmapped design. DesignWare selection can recur after gate-level mapping.

High-level synthesis tasks are based on your constraints and your HDL coding style. After high-level optimization, circuit function is represented by GTECH library parts, that is, by a generic, technology-independent netlist.

For more information about how your coding style affects architectural optimization, see Chapter 3, “Preparing Design Files for Synthesis”.

Logic-Level Optimization

Logic-level optimization works on the GTECH netlist. It consists of the following two processes:

- Structuring

  This process adds intermediate variables and logic structure to a design, which can result in reduced design area. Structuring is constraint based. It is best applied to noncritical timing paths.

  During structuring, Design Compiler searches for subfunctions that can be factored out and evaluates these factors, based on the size of the factor and the number of times the factor appears in the design. Design Compiler turns the subfunctions that most reduce the logic into intermediate variables and factors them out of the design equations.

- Flattening

  The goal of this process is to convert combinational logic paths of the design to a two-level, sum-of-products representation. Flattening is carried out independently of constraints. It is useful for speed optimization because it leads to just two levels of combinational logic.

  During flattening, Design Compiler removes all intermediate variables, and therefore all its associated logic structure, from a design.

Gate-Level Optimization

Gate-level optimization works on the generic netlist created by logic synthesis to produce a technology-specific netlist. It includes the following processes:

- Mapping

  This process uses gates (combinational and sequential) from the target technology libraries to generate a gate-level implementation of the design whose goal is to meet timing and area goals. You can use the various options of the `compile_ultra` command or the `compile` command to control the mapping algorithms used by Design Compiler.

- Delay optimization

  The process goal is to fix delay violations introduced in the mapping phase. Delay optimization does not fix design rule violations or meet area constraints.

- Design rule fixing

  The process goal is to correct design rule violations by inserting buffers or resizing existing cells. Design Compiler tries to fix these violations without affecting timing and area results, but if necessary, it does violate the optimization constraints.

- Area optimization

  The process goal is to meet area constraints after the mapping, delay optimization, and design rule fixing phases are completed. However, Design Compiler does not allow area recovery to introduce design rule or delay constraint violations as a means of meeting the area constraints.

You can change the priority of the constraints by using the `set_cost_priority` command. Also, you can disable design rule fixing by specifying the `-no_design_rule` option when you run the `compile_ultra` command or `compile` command. However, if you use this option, your synthesized design might violate design rules.

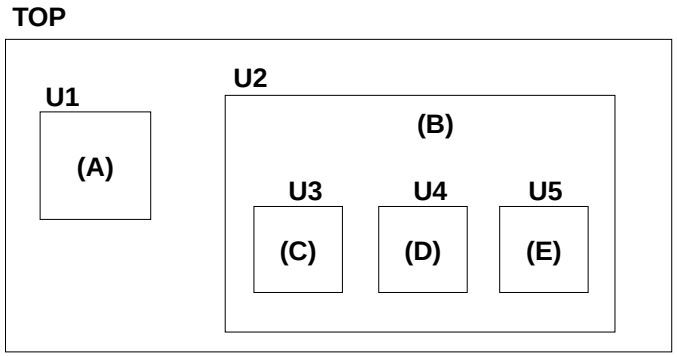
Selecting and Using a Compile Strategy

You can use various strategies to compile your hierarchical design. The basic strategies are

- Top-down compile, in which the top-level design and all its subdesigns are compiled together
- Bottom-up compile, in which the individual subdesigns are compiled separately, starting from the bottom of the hierarchy and proceeding up through the levels of the hierarchy until the top-level design is compiled
- Mixed compile, in which the top-down or bottom-up strategy, whichever is most appropriate, is applied to the individual subdesigns

In the following sections, the top-down and bottom-up compile strategies are demonstrated, using the simple design shown in Figure 9-1.

Figure 9-1 Design to Illustrate Compile Strategies



The top-level, or global, specifications for this design, given in Table 9-1, are defined by the script of Example 9-1. These specifications apply to TOP and all its subdesigns.

Table 9-1 Design Specifications for Design TOP

Specification type	Value
Operating condition	WCCOM
Wire load model	"20x20"
Clock frequency	40 MHz
Input delay time	3 ns
Output delay time	2 ns
Input drive strength	drive_of (IV)
Output load	1.5 pF

Example 9-1 Constraints File for Design TOP (defaults.con)

```
set_operating_conditions WCCOM
set_wire_load_model "20x20"
create_clock -period 25 clk
set_input_delay 3 -clock clk \
   [remove_from_collection [all_inputs] [get_ports clk]]
set_output_delay 2 -clock clk [all_outputs]
set_load 1.5 [all_outputs]
set_driving_cell -lib_cell IV [all_inputs]
set_drive 0 clk
```

Note:

To prevent buffering of the clock network, the script sets the input drive resistance of the clock port (clk) to 0 (infinite drive strength).

Top-Down Compile

You can use the top-down compile strategy for designs that are not memory or CPU limited. Furthermore, top-level designs that are memory limited can often be compiled using the top-down strategy if you first replace some of the subdesigns with interface logic model representations. Replacing a subdesign with an interface logic model can greatly reduce the memory requirements for the subdesign instantiation in the top-level design. For information about how to generate and use interface logic models, see the *Interface Logic Model User Guide*.

The top-down compile strategy has these advantages:

- Provides a push-button approach
- Takes care of interblock dependencies automatically

On the other hand, the top-down compile strategy requires more memory and might result in longer runtimes for designs with over 100K gates.

To implement a top-down compile, carry out the following steps:

Note:

If your top-level design contains one or more interface logic models, use the compile flow described in the *Interface Logic Model User Guide*.

1. Read in the entire design.
2. Apply attributes and constraints to the top level.

   Attributes and constraints implement the design specification. For information about attributes, see “Working With Attributes” on page 5-55. For information about constraints, see Chapter 6, “Defining the Design Environment” and Chapter 7, “Defining Design Constraints.”

   Note:

   You can assign local attributes and constraints to subdesigns, provided that those attributes and constraints are defined with respect to the top-level design.
3. Compile the design.

A top-down compile script for the TOP design is shown in Example 9-2.The script contains comments that identify each of the steps. The constraints are applied by including the constraint file (defaults.con) shown in Example 9-1 on page 9-8.

Example 9-2 Top-Down Compile Script

```
/* read in the entire design */
read_verilog E.v
read_verilog D.v
read_verilog C.v
read_verilog B.v
read_verilog A.v
read_verilog TOP.v
current_design TOP
link

/* apply constraints and attributes */
source defaults.con

/* compile the design */
compile
```

Bottom-Up Compile

Use the bottom-up compile strategy for medium-size and large designs.

Note:

The bottom-up compile strategy is also known as the compile-characterize-write_script-recompile method.

The bottom-up compile strategy provides these advantages:

- Compiles large designs by using the divide-and-conquer approach
- Requires less memory than top-down compile
- Allows time budgeting

The bottom-up compile strategy requires

- Iterating until the interfaces are stable

- Manual revision control

The bottom-up compile strategy compiles the subdesigns separately and then incorporates them in the top-level design. The top-level constraints are applied, and the design is checked for violations. Although it is possible that no violations are present, this outcome is unlikely because the interface settings between subdesigns usually are not sufficiently accurate at the start.

To improve the accuracy of the interblock constraints, you read in the top-level design and all compiled subdesigns and apply the `characterize` command to the individual cell instances of the subdesigns. Based on the more realistic environment provided by the compiled subdesigns, `characterize` captures environment and timing information for each cell instance and then replaces the existing attributes and constraints of each cell's referenced subdesign with the new values.

Using the improved interblock constraint, you recompile the characterized subdesigns and again check the top-level design for constraint violations. You should see improved results, but you might need to iterate the entire process several times to remove all significant violations.

The bottom-up compile strategy requires these steps:

1. Develop both a default constraint file and subdesign-specific constraint files.

   The default constraint file includes global constraints, such as the clock information and the drive and load estimates. The subdesign-specific constraint files reflect the time budget allocated to the subblocks.

2. Compile the subdesigns independently.

3. Read in the top-level design and any compiled subdesigns not already in memory.
4. Set the current design to the top-level design, link the design, and apply the top-level constraints.

   If the design meets its constraints, you are finished. Otherwise, continue with the following steps.

5. Apply the `characterize` command to the cell instance with the worst violations.
6. Use `write_script` to save the characterized information for the cell.

   You use this script to re-create the new attribute values when you are recompiling the cell's referenced subdesign.

7. Use `remove_design -all` to remove all designs from memory.
8. Read in the RTL design of the previously characterized cell.

   Recompiling the RTL design instead of the cell's mapped design usually leads to better optimization.

9. Set `current_design` to the characterized cell's subdesign and recompile, using the saved script of characterization data.
10. Read in all other compiled subdesigns.
11. Link the current subdesign.
12. Choose another subdesign, and repeat steps 3 through 9 until you have recompiled all subdesigns, using their actual environments.

When applying the bottom-up compile strategy, consider the following:

- The `read_file` command runs most quickly with the .ddc format. If you will not be modifying your RTL code after the first time you read (or elaborate) it, save the unmapped design to a .ddc file. This will save time when you reread the design.
- The `compile` command affects all subdesigns of the current design. If you want to optimize only the current design, you can remove or not include its subdesigns in your database, or you can place the `dont_touch` attribute on the subdesigns (by using the `set_dont_touch` command).
- The subdesign constraints are not preserved after you perform a top-level compile. To ensure that you are using the correct constraints, always reapply the subdesign constraints before compiling or analyzing a subdesign.
- By default, compile modifies the original copy in the current design. This could be a problem when the same design is referenced from multiple modules, which are compiled separately in sequence. For example, the first compile could change the interface or the functionality of the design by boundary optimizaiton. When this design is referenced from another module in the subsequent compile, the modified design is uniquified and used.

  You can set the `compile_keep_original_for_external_references` variable to true, which enables compile to keep the original design when there is an external reference to the design. When the variable is set to true, the original design and its sub-designs are copied and preserved (before doing any modifications during compile) if there is an external reference to this design.

Typically, you require this variable only when you are doing a bottom-up compile without setting a `dont_touch` attribute on all the sub-designs, especially those with boundary optimizations turned on. If there is a `dont_touch` attribute on any of the instances of the design or in the design, this variable has no effect.

A bottom-up compile script for the TOP design is shown in Example 9-3 on page 9-15. The script contains comments that identify each of the steps in the bottom-up compile strategy. In the script it is assumed that block constraint files exist for each of the subblocks (subdesigns) in design TOP. The compile script also uses the default constraint file (defaults.con) shown in Example 9-1 on page 9-8.

Note:

This script shows only one pass through the bottom-up compile procedure. If the design requires further compilations, you repeat the procedure from the point where the top-level design, TOP.v, is read in.

Example 9-3 Bottom-Up Compile Script

```
set all_blocks {E D C B A}

# compile each subblock independently
foreach block $all_blocks {
   # read in block
   set block_source "$block.v"
   read_file -format verilog $block_source
   current_design $block
   link
   # apply global attributes and constraints
   source defaults.con
   # apply block attributes and constraints
   set block_script "$block.con"
   source $block_script
   # compile the block
   compile
}

# read in entire compiled design
read_file -format verilog TOP.v
current_design TOP
link
write -hierarchy -format ddc -output first_pass.ddc

# apply top-level constraints
source defaults.con
source top_level.con

# check for violations
report_constraint

# characterize all instances in the design
set all_instances {U1 U2 U2/U3 U2/U4 U2/U5}
characterize -constraint $all_instances

# save characterize information
foreach block $all_blocks {
   current_design $block
   set char_block_script "$block.wscr"
   write_script > $char_block_script
}

# recompile each block
```

```
foreach block $all_blocks {

   # clear memory
   remove_design -all

   # read in previously characterized subblock
   set block_source "$block.v"
   read_file -format verilog $block_source

   # recompile subblock
   current_design $block
   link
   # apply global attributes and constraints
   source defaults.con
   # apply characterization constraints
   set char_block_script "$block.wscr"
   source $char_block_script
   # apply block attributes and constraints
   set block_script "$block.con"
   source $block_script
   # recompile the block
   compile
}
```

Note:

When performing a bottom-up compile, if the top-level design contains glue logic as well as the subblocks (subdesigns), you must also compile the top-level design. In this case, to prevent Design Compiler from recompiling the subblocks, you first apply the `set_dont_touch` command to each subdesign.

Mixed Compile Strategy

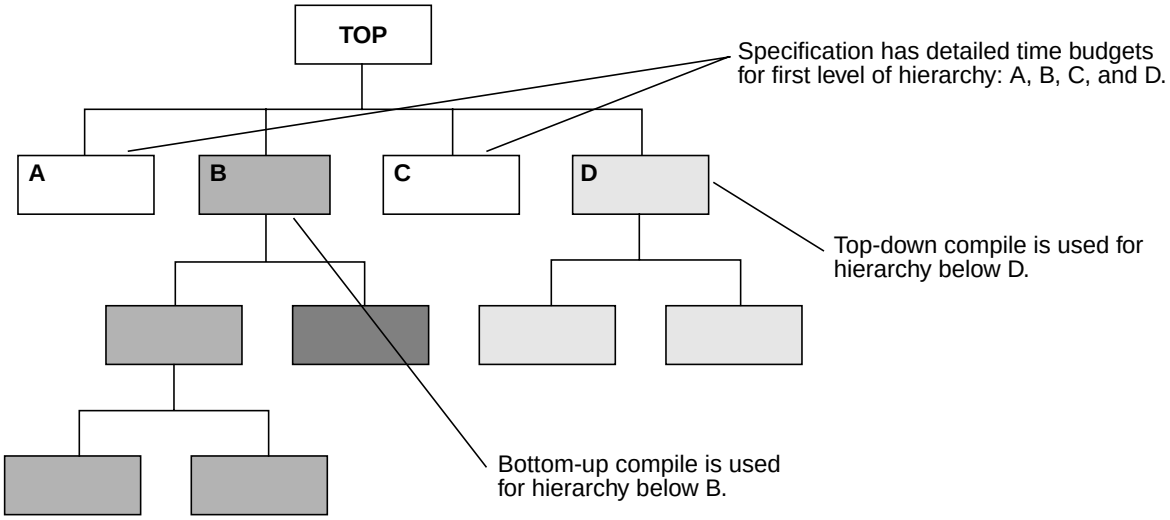
You can take advantage of the benefits of both the top-down and the bottom-up compile strategies by using both.

- Use the top-down compile strategy for small hierarchies of blocks.

- Use the bottom-up compile strategy to tie small hierarchies together into larger blocks.

Figure 9-2 shows an example of the mixed compilation strategy.

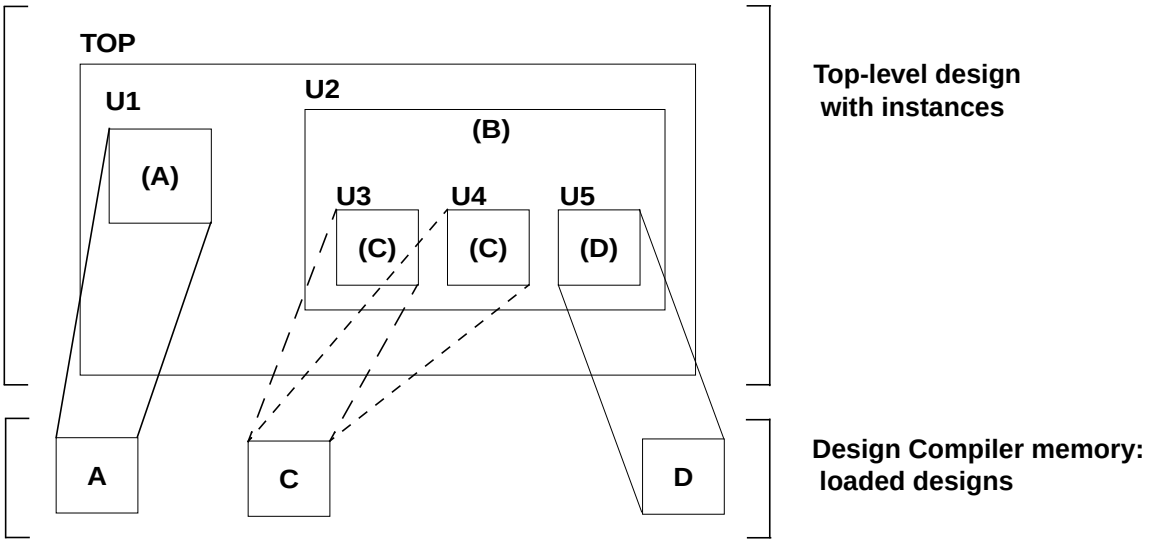
Figure 9-2 Mixing Compilation Strategies



Resolving Multiple Instances of a Design Reference

In a hierarchical design, subdesigns are often referenced by more than one cell instance, that is, multiple references of the design can occur. For example, Figure 9-3 shows the design TOP, in which design C is referenced twice (U2/U3 and U2/U4).

Figure 9-3 Multiple Instances of a Design Reference



Use the `check_design -multiple_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:

- The uniquify method

  In earlier releases, you had manually to run the `uniquify` command to create a uniquely named copy of the design for each instance. However, beginning with version V-2004.06, the tool automatically uniquifies designs as part of the compile process.

  Note that you can still manually force the tool to uniquify designs before compile by running the `uniquify` command, but this step contributes to longer runtimes because the tool automatically “re-uniquifies” the designs when compile the design. You cannot turn off the uniquify process.

- The compile-once-don’t-touch method

  This method uses the `set_dont_touch` command to preserve the compiled subdesign while the remaining designs are compiled.

- The ungroup method

  This method uses the `ungroup` command to remove the hierarchy.

These methods are described in the following sections.

Uniquify Method

The uniquify process copies and renames any multiply referenced design so that each instance references a unique design. The process removes the original design from memory after it creates the new, unique designs. The original design and any collections that contain it or its objects are no longer accessible.

In earlier releases, you had manually to run the `uniquify` command to create a uniquely named copy of the design for each instance. However, beginning with version V-2004.06, the tool automatically uniquifies designs as part of the compile process. The uniquification process can resolve multiple references throughout the hierarchy the current design (except those having a `dont_touch` attribute). After this process finishes, the tool can optimize each design copy based on the unique environment of its cell instance.

You can also create unique copies for specific references by using the `-reference` option of the `uniquify` command, or you can specify specific cells by using the `-cell` option. Design Compiler makes unique copies for cells specified with the `-reference` or the `-cells` option, even if they have a `dont_touch` attribute.

The `uniquify` command accepts instance objects—that is, cells at a lower level of hierarchy. When you use the `-cell` option with an instance object, the complete path to the instance is uniquified. For example, the following command uniquifies both instances mid1 and mid1/bot1 (assuming that mid1 is not unique):

```
dc_shell-xg-t> uniquify -cell mid1/bot1
```

Design Compiler uses the naming convention specified in the `uniquify_naming_style` variable to generate the name for each copy of the subdesign. The default naming convention is

```
%s_%d
```

`%s`

The original name of the subdesign (or the name specified in the `-base_name` option).

`%d`

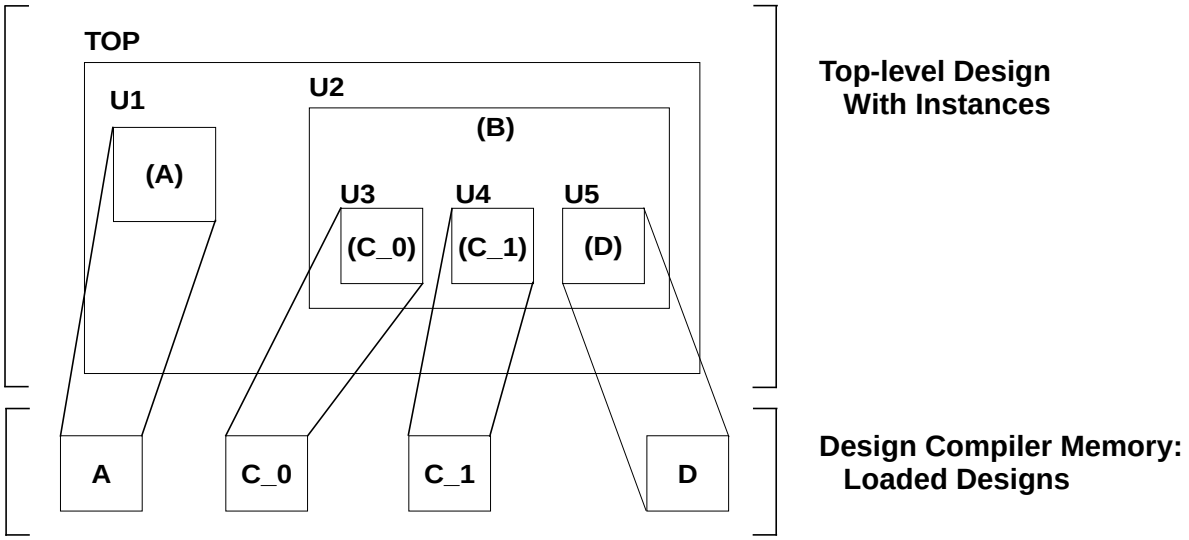
The smallest integer value that forms a unique subdesign name.

The following command sequence resolves the multiple instances of design C in design TOP shown in Figure 9-3 on page 9-18; it uses the automatic uniquify method to create new designs C_0 and C_1 by copying design C and then replaces design C with the two copies in memory.

```
dc_shell-xg-t> current_design top
dc_shell-xg-t> compile
```

Figure 9-4 shows the result of running this command sequence.

Figure 9-4 Uniquify Results



Compared with the compile-once-don't-touch method, the uniquify method has the following characteristics:

- Requires more memory
- Takes longer to compile

Compile-Once-Don't-Touch Method

If the environments around the instances of a multiply referenced design are sufficiently similar, use the compile-once-don't-touch method. In this method, you compile the design, using the environment of one of its instances, and then you use the `set_dont_touch` command to preserve the subdesign during the remaining optimization. For details about the `set_dont_touch` command, see "Preserving Subdesigns" on page 9-25.

To use the compile-once-don't-touch method to resolve multiple instances, follow these steps:

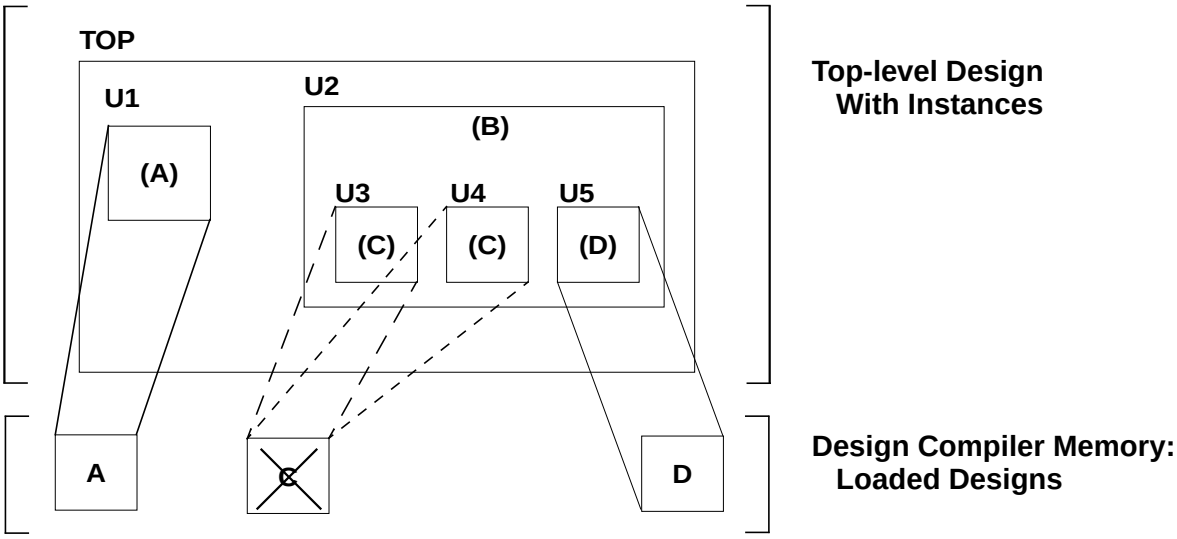
1. Characterize the subdesign's instance that has the worst-case environment.
2. Compile the referenced subdesign.
3. Use the `set_dont_touch` command to set the `dont_touch` attribute on all instances that reference the compiled subdesign.
4. Compile the entire design.

For example, the following command sequence resolves the multiple instances of design C in design TOP by using the compile-once-don't-touch method (assuming U2/U3 has the worst-case environment). In this case, no copies of the original subdesign are loaded into memory.

```
dc_shell-xg-t> current_design top
dc_shell-xg-t> characterize U2/U3
dc_shell-xg-t> current_design C
dc_shell-xg-t> compile
dc_shell-xg-t> current_design top
dc_shell-xg-t> set_dont_touch {U2/U3 U2/U4}
dc_shell-xg-t> compile
```

Figure 9-5 shows the result of running this command sequence. The X drawn over the C design, which has already been compiled, indicates that the `dont_touch` attribute has been set. This design is not modified when the top-level design is compiled.

Figure 9-5 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.

Ungroup Method

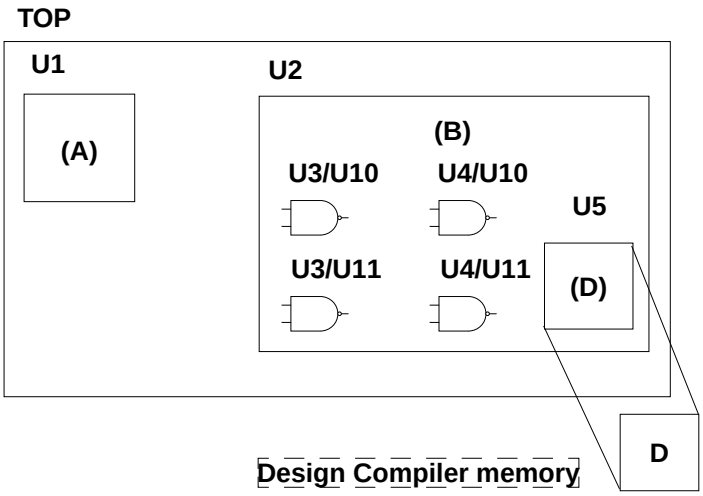
The ungroup method has the same effect as the uniquify method (it makes unique copies of the design), but in addition, it removes levels of hierarchy. This method uses the `ungroup` command to produce a flattened netlist. For details about the `ungroup` command, See "Removing Levels of Hierarchy" on page 5-34.

After ungrouping the instances of a subdesign, you can recompile the top-level design. For example, the following command sequence uses the ungroup method to resolve the multiple instances of design C in design TOP:

```
dc_shell-xg-t> current_design B
dc_shell-xg-t> ungroup {U3 U4}
dc_shell-xg-t> current_design top
dc_shell-xg-t> compile
```

Figure 9-6 shows the result of running this command sequence.

Figure 9-6 Ungroup Results



The ungroup method has the following characteristics:

- Requires more memory and takes longer to compile than the compile-once-don’t-touch method
- Provides the best synthesis results

The obvious drawback in using the ungroup method is that it removes the user-defined design hierarchy.

Preserving Subdesigns

The `set_dont_touch` command preserves a subdesign during optimization. It places the `dont_touch` attribute on cells, nets, references, and designs in the current design to prevent these objects from being modified or replaced during optimization.

Note:

Any interface logic model present in your design is automatically marked as `dont_touch`. Also, the cells of an interface logic model are marked as `dont_touch`. For information about interface logic models, see the *Interface Logic Model User Guide*.

Use the `set_dont_touch` command on subdesigns you do not want optimized with the rest of the design hierarchy. The `dont_touch` attribute does not prevent or disable timing through the design.

When you use `set_dont_touch`, remember the following points:

- Setting `dont_touch` on a hierarchical cell sets an implicit `dont_touch` on all cells below that cell.
- Setting `dont_touch` on a library cell sets an implicit `dont_touch` on all instances of that cell.
- Setting `dont_touch` on a net sets an implicit `dont_touch` only on mapped combinational cells connected to that net. If the net is connected only to generic logic, optimization might remove the net.
- Setting `dont_touch` on a reference sets an implicit `dont_touch` on all cells using that reference during subsequent optimizations of the design.

- Setting `dont_touch` on a design has an effect only when the design is instantiated within another design as a level of hierarchy. In this case, the `dont_touch` attribute on the design implies that all cells under that level of hierarchy are subject to the `dont_touch` attribute. Setting `dont_touch` on the top-level design has no effect because the top-level design is not instantiated within any other design.
- You cannot manually or automatically ungroup objects marked as `dont_touch`. That is, the `ungroup` command and the compile `-ungroup_all` and `-auto_ungroup` options have no effect on `dont_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 `dont_touch` nets connected to unmapped cells (generic logic).

Use the `report_design` command to determine whether a design has the `dont_touch` attribute set.

```
dc_shell-xg-t> set_dont_touch SUB_A
1
dc_shell-xg-t> report_design

****************************************
Report : design
Design : SUB_A
Version: Y-2006.06
Date   : Mon May 1 10:56:49 2006
****************************************

Design is dont_touched.
```

To remove the `dont_touch` attribute, use the `remove_attribute` command or the `set_dont_touch` command set to false.

Understanding the Compile Cost Function

The compile cost function consists of design rule costs and optimization costs. By default, Design Compiler prioritizes costs in the following order:

1. Design rule costs
    a. Connection class
    b. Multiple port nets
    c. Maximum transition time
    d. Maximum fanout
    e. Maximum capacitance
    f. Cell degradation
2. Optimization costs
    a. Maximum delay
    b. Minimum delay
    c. Maximum power
    d. Maximum area

The compile cost function considers only those components that are active in your design. Design Compiler evaluates each cost function component independently, in order of importance.

When evaluating cost function components, Design Compiler considers only violators (positive difference between actual value and constraint) and works to reduce the cost function to zero.

The goal of Design Compiler is to meet all constraints. However, by default, it gives precedence to design rule constraints because design rule constraints are functional requirements for designs.

Using the default priority, Design Compiler fixes design rule violations even at the expense of violating your delay or area constraints.

You can change the priority of the maximum design rule costs and the delay costs by using the `set_cost_priority` command to specify the ordering. You must run the `set_cost_priority` command before running the `compile` command.

You can disable evaluation of the design rule cost function by using the `-no_design_rule` option when running the `compile_ultra` command or `compile` command.

You can disable evaluation of the optimization cost function by using the `-only_design_rule` option when running the `compile_ultra` command or `compile` command.

For more information on constraints, see the *Design Compiler Reference Manual: Constraints and Timing.* For more information on the compile cost function, see the *Design Compiler Reference Manual: Optimization and Timing Analysis*.

Performing Design Exploration

In design exploration, you use the default synthesis algorithm to gauge the design performance against your goals. To invoke the default synthesis algorithm, use the `compile` command with no options:

```
dc_shell-xg-t> compile
```

The default compile uses the `-map_effort medium` option of the `compile` command. The default area effort of the area recovery phase of the compile is the specified value of the `map_effort` option. You can change the area effort by using the `-area_effort` option.

If the performance violates the timing goals by more than 15 percent, you should consider whether to refine the design budget or modify the HDL code.

Performing Design Implementation

The default compile generates good results for most designs. If your design meets the optimization goals after design exploration, you are finished. If not, try the techniques described in the following sections:

- Optimizing High-Performance Designs
- Optimizing for Maximum Performance
- Optimizing for Minimum Area
- Optimizing Data Paths

Optimizing High-Performance Designs

For high-performance designs that have significantly tight timing constraints, you can invoke a single DC Ultra command, `compile_ultra`, for better quality of results (QoR). This command allows you to apply the best possible set of timing-centric variables or commands during compile for critical delay optimization as well as

improvement in area QoR. Because `compile_ultra` includes all compile options and starts the entire compile process, no separate `compile` command is necessary.

The command syntax is

```
int compile_ultra [-scan] [-exact_map][-no_uniquify]
[-no_boundary_optimization]
[-no_autoungroup][-no_seq_output_inversion]
[-area_high_effort_script][-timing_high_effort_script]
```

Note:

Additional options are available in DC Topographical mode. Compile options, such as `-map_effort`, `-incremental_mapping`, and `-area_effort`, are not compatible with the `compile_ultra` command. See “Using Design Compiler Topographical Technology” on page 8-1.

By default, if the dw_foundation.sldb library is not in the synthetic library list but the DesignWare license has been successfully checked out, the dw_foundation.sldb library is automatically added to the synthetic library list. This behavior applies to the current command only. The user-specified synthetic library and link library lists are not affected.

In addition, all DesignWare hierarchies are, by default, unconditionally ungrouped in the second pass of the compile. You can prevent this ungrouping by setting the `compile_ultra_ungroup_dw` variable to false (the default is true).

To use the `compile_ultra` command, you will need a DC Ultra license and a DesignWare Foundation license.

For more information on this command, see the man page and the *Design Compiler Reference Manual:Optimization and Timing Analysis*.

Optimizing for Maximum Performance

If your design does not meet the timing constraints, you can try the following methods to improve performance:

- Create path groups
- Fix heavily loaded nets
- Auto-ungroup hierarchies on the critical path
- Perform a high-effort incremental compile
- Perform a high-effort compile

Creating Path Groups

By default, Design Compiler groups paths based on the clock controlling the endpoint (all paths not associated with a clock are in the default path group). If your design has complex clocking, complex timing requirements, or complex constraints, you can create path groups to focus Design Compiler on specific critical paths in your design.

Use the `group_path` command to create path groups. The `group_path` command allows you to

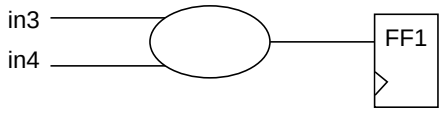
- Control the optimization of your design
- Optimize near-critical paths
- Optimize all paths

Controlling the Optimization of Your Design. You can control the optimization of your design by creating and prioritizing path groups, which affect only the maximum delay cost function. By default, Design Compiler works only on the worst violator in each group.

Set the path group priorities by assigning weights to each group (the default weight is 1.0). The weight can be from 0.0 to 100.0.

For example, Figure 9-7 shows a design that has multiple paths to flip-flop FF1.

Figure 9-7 Path Group Example



To indicate that the path from input in3 to FF1 is the highest-priority path, use the following command to create a high-priority path group:

```
dc_shell-xg-t> group_path -name group3 \
          -from in3 -to FF1/D -weight 2.5
```

Optimizing Near-Critical Paths. When you add a critical range to a path group, you change the maximum delay cost function from worst negative slack to critical negative slack. Design Compiler optimizes all paths within the critical range.

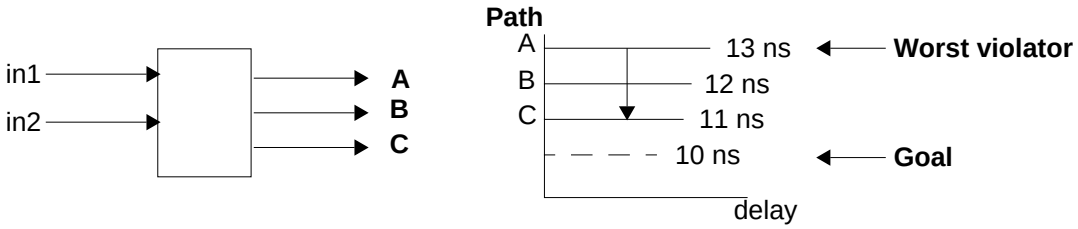
Specifying a critical range can increase runtime. To limit the runtime increase, use critical range only during the final implementation phase of the design, and use a reasonable critical range value. A guideline for the maximum critical range value is 10 percent of the clock period.

Use one of the following methods to specify the critical range:

- Use the -`critical_range` option of the `group_path` command.
- Use the `set_critical_range` command.

For example, Figure 9-8 shows a design with three outputs, A, B, and C.

Figure 9-8 Critical Range Example



Assume that the clock period is 20 ns, the maximum delay on each of these outputs is 10 ns, and the path delays are as shown. By default, Design Compiler optimizes only the worst violator (the path to output A). To optimize all paths, set the critical delay to 3.0 ns. For example,

```
create_clock -period 20 clk
set_critical_range 3.0 $current_design
set_max_delay 10 {A B C}
group_path -name group1 -to {A B C}
```

Optimizing All Paths. You can optimize all paths by creating a path group for each endpoint in the design. Creating a path group for each endpoint enables total negative slack optimization but results in long compile runtimes.

Use the following script to create a path group for each endpoint.

```
set endpoints \
     [add_to_collection [all_outputs] \
     [all_registers -data_pins]]
foreach_in_collection endpt $endpoints {
   set pin [get_object_name $endpt]
   group_path -name $pin -to $pin
}
```

Fixing Heavily Loaded Nets

Heavily loaded nets often become critical paths. To reduce the load on a net, you can use either of two approaches:

- If the large load resides in a single module and the module contains no hierarchy, fix the heavily loaded net by using the `balance_buffer` command. For example, enter

  ```
  source constraints.con
  compile_ultra
  balance_buffer -from [get_pins buf1/Z]
  ```

  Note:

  The `balance_buffer` command provides the best results when your library uses linear delay models. If your library uses nonlinear delay models, the second approach provides better results.

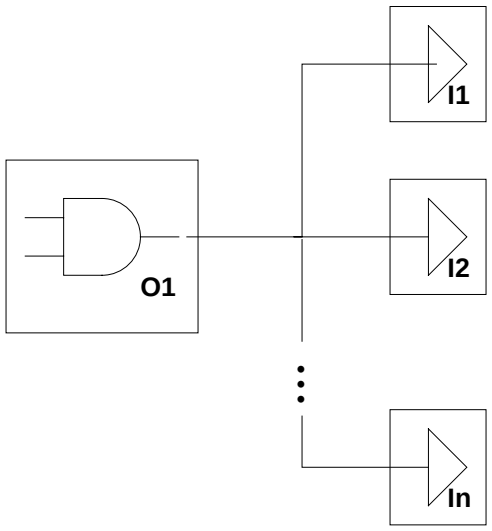
- If the large loads reside across the hierarchy from several modules, apply design rules to fix the problem. For example,

  ```
  source constraints.con
  compile_ultra
  set_max_capacitance 3.0
  compile_ultra -only_design_rule
  ```

In rare cases, hierarchical structure might disable Design Compiler from fixing design rules.

In the sample design shown in Figure 9-9, net O1 is overloaded. To reduce the load, group as many of the loads (I1 through In) as possible in one level of hierarchy by using the `group` command or by changing the HDL. Then you can apply one of the approaches.

Figure 9-9 Heavily Loaded Net



Automatically Ungrouping Hierarchies on the Critical Path

Automatically ungrouping hierarchies during compile can often improve performance. Ungrouping removes hierarchy boundaries and allows Design Compiler to optimize over a larger number of gates, generally improving timing. You use delay-based auto-ungrouping to ungroup hierarchies along the critical path.

To use the auto-ungroup capability, use the `compile_ultra` command or the `-auto_ungroup delay` option of the `compile` command.

For more information on auto-ungrouping, see the *Design Compiler Reference Manual:Optimization and Timing Analysis*.

Performing a High-Effort Compile

The optimization result depends on the starting point. Occasionally, the starting point generated by the default compile results in a local minimum solution, and Design Compiler quits before generating an optimal design. A high-effort compile might solve this problem.

The high-effort compile uses the `-map_effort high` option of the `compile` command on the initial compile (on the HDL description of the design).

```
dc_shell-xg-t> elaborate my_design
dc_shell-xg-t> compile -map_effort high
```

A high-effort compile pushes Design Compiler to the extreme to achieve the design goal. A high-effort compile invokes the critical path resynthesis strategy to restructure and remap the logic on and around the critical path.

This compile strategy is CPU intensive, especially when you do not use the incremental compile option, with the result that the entire design is compiled using a high map effort.

Performing a High-Effort Incremental Compile

You can often improve compile performance of a high-effort compile by using the incremental compile option. Also, if none of the previous strategies results in a design that meets your optimization goals, a high-effort incremental compile might produce the desired result.

An incremental compile (`-incremental_mapping` compile option) allows you to incrementally improve your design by experimenting with different approaches. An incremental compile performs only

gate-level optimization and does not perform logic-level optimization. The resulting design's performance is the same or better than the original design's.

This technique can still require large amounts of CPU time, but it is the most successful method for reducing the worst negative slack to zero. To reduce runtime, you can place a `dont_touch` attribute on all blocks that already meet timing constraints.

```
dc_shell-xg-t> dont_touch noncritical_blocks
dc_shell-xg-t> compile -map_effort high \
                  -incremental_mapping
```

This incremental approach works best for a technology library that has many variations of each logic cell.

Optimizing for Minimum Area

If your design has timing constraints, these constraints always take precedence over area requirements. For area-critical designs, do not apply timing constraints before you compile. If you want to view timing reports, you can apply timing constraints to the design after you compile.

If your design does not meet the area constraints, you can try the following methods to reduce the area:

- Disable total negative slack optimization
- Optimize across hierarchical boundaries

Disabling Total Negative Slack Optimization

By default, Design Compiler prioritizes total negative slack over meeting area constraints. This means Design Compiler performs area optimization only on those paths that have positive slack.

To change the default priorities (prioritize area over total negative slack), use the `-ignore_tns` option when setting the area constraints.

```
dc_shell-xg-t> set_max_area -ignore_tns max_area
```

Optimizing Across Hierarchical Boundaries

Design Compiler respects levels of hierarchy and port functionality (except when automatic ungrouping of small hierarchies is enabled). Boundary optimizations, such as constant propagation through a subdesign, do not occur automatically.

To fine-tune the area, you can leave the hierarchy intact and enable boundary optimization. For greater area reduction, you might have to remove hierarchical boundaries.

Boundary Optimization. Direct Design Compiler to perform optimization across hierarchical boundaries (boundary optimization) by using one of the following commands:

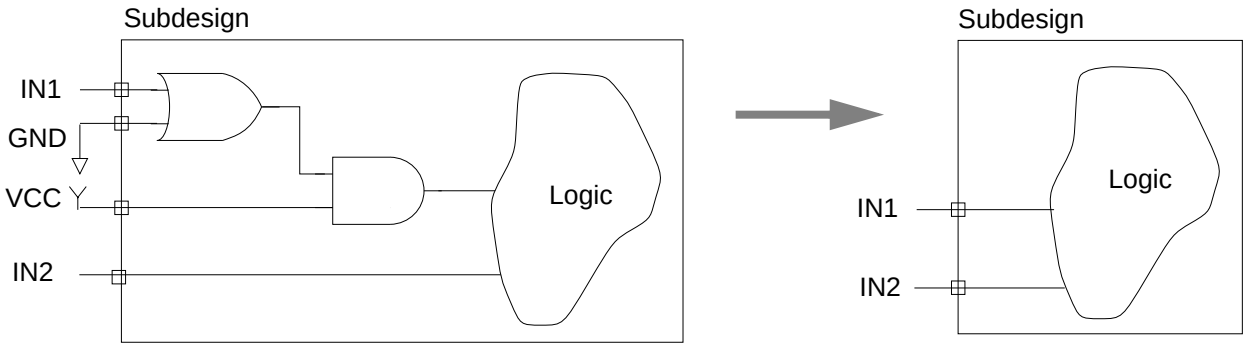
```
dc_shell-xg-t> compile -boundary_optimization
```

or

```
dc_shell-xg-t> set_boundary_optimization subdesign
```

If you enable boundary optimization, Design Compiler 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 9-10 shows this relationship.

Figure 9-10 Benefits of Boundary Optimization



Hierarchy Removal. Removing levels of hierarchy by ungrouping gives Design Compiler more freedom to share common terms across the entire design. You can ungroup specific hierarchies before optimization by using the `set_ungroup` command or the `compile` command with the `-ungroup_all` option to designate which cells you want ungrouped. Also, you can use the auto-ungroup capability of Design Compiler to ungroup small hierarchies during optimization. In this case, you do not specify the hierarchies to be ungrouped.

For details about ungrouping hierarchies, see “Removing Levels of Hierarchy” on page 5-34.

Optimizing Data Paths

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 (for example, 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.

Beginning with version W-2004.12, Design Compiler provides an improved datapath generator and better arithmetic components for both DC Expert and DC Ultra. To take advantage of these enhancements, make sure that the dw_foundation.sldb library is listed in the synthetic library. If necessary, use the `set synthetic_library dw_foundation.sldb` command. These enhancements require a DesignWare license.

DC Ultra enables datapath extraction and explores various datapath and resource-sharing options during compile. DC Ultra datapath optimization provides the following benefits:

- Shares datapath operators
- Extracts the datapath
- Explores better solutions that might involve a different resource-sharing configuration
- Allows the tool to make better tradeoffs between resource sharing and datapath optimization

DC Ultra datapath optimization is enabled by default when you use any of the following:

- `set_ultra_optimization true` followed by `compile`
- `compile_ultra`

To disable DC Ultra datapath optimization, set `hlo_disable_datapath_optimization` to true. (The default is false.)

For more information on datapath synthesis, see the *Design Compiler Reference Manual: Optimization and Timing Analysis*.

10

Using a Milkyway Database

You can write a Milkyway database within Design Compiler to use with other Synopsys Galaxy platform tools, such as Physical Compiler, JupiterXT, and Astro. You do this by way of the `write_milkyway` commands. You can use a single Milkyway library across the entire Galaxy flow.

Note:

Design Compiler does not support the `read_milkyway` command.

When you use a Milkyway database, you do not need to use an intermediate netlist file exchange format such as Verilog or VHDL to communicate with other Synopsys Galaxy platform tools.

Before you can use a Milkyway database within Design Compiler, you must prepare a design library and a reference library and know about the concepts and tasks described in these sections:

Licensing and Required Files

The Milkyway-Interface license is already provided to Design Compiler users. This license is needed for the `write_milkyway` commands.

Table 10-1 and Table 10-2 list the files required for using a Milkyway database.

Table 10-1 Source Files

Source file	Description
.lib	Source for logic libraries

Table 10-2 Compiled Databases

Compiled databases	Description
.db	Logic library containing standard cell timing, power, function, test, and so forth
Milkyway library	Physical technology data, FRAM

Invoking the Milkyway Tool

You invoke the Milkyway tool by using the `Milkyway -galaxy` command, which checks out the Milkyway-Interface license.

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 on 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.

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 `write_milkyway`.
- The Design Compiler unit is based on the logic library, and the Astro unit is from the Milkyway technology file. 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 `write_sdc` 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 `write_milkyway` is run:

```
set_load 1425.15 [get_nets {gpdhi_word_d_21_}]
```

After `write_milkyway` is run, the SDC file shows

```
set_load 8.36909 [get_nets {gpdhi_word_d_21_}]
```

- Design Compiler is case-sensitive. However, you can use the tool in case-insensitive mode by doing the following before you run `write_milkyway`:
  - Prepare uppercase versions of the libraries used in the link library.
  - Use the `change_names` command to make sure the netlist is uppercased.

Converting Design Data From .db Format to Milkyway Format

To convert your design data from .db format to Milkyway format, read the .db file in to dc_shell, then save the design in Milkyway format (`write_milkyway`). For example,

```
dc_shell-xg-t> set mw_cel_without_fram_tech true
dc_shell-xg-t> set mw_design_library design_dir

dc_shell-xg-t> read_db my_design.db
dc_shell-xg-t> write_milkyway -output my_design
```

Preparing to Use the Milkyway Database

Before you can use the `write_milkyway` command, you must do the following:

1. Specify the Milkyway reference library directories by setting the `mw_reference_library` variable.

   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.

2. Specify the design database location by setting the `mw_design_library` variable.

   The `write_milkyway` command uses the design database location specified in this variable.

3. Define the power nets by setting the `mw_logic0_net` and `mw_logic1_net` variables.

   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.

4. Run the `create_mw_design` command to create the Milkyway design library.

   Specify the design name, technology file, and Milkyway reference library files. Design Compiler uses the FRAM view of the reference libraries as the default physical model for your design. For example, enter

```
dc_shell-xg-t> create_mw_design design_name -tech_file \
               ./path/tech_file_name.tf
```

Writing the Milkyway Database

To save the design data in a Milkyway design library, use the `write_milkyway` command. The `write_milkyway` command writes netlist and physical data from memory to Milkyway design library format. You can read in this data to Astro and use it to route the design correctly.

To use the `write_milkyway` command, enter

```
dc_shell-xg-t> write_milkyway [options] file_name
```

To do this	Use this
Specify the name of the design file to write (required). If you omit this switch, the tool displays an error message Error: Required argument '-output' was not found (CMD-007)	`-output`
Overwrite the current version of the design file. If you omit this option, a new, additional file version is written. Use this switch to save disk space.	`-overwrite`

You must use the `-output` option to specify the file name.

```
dc_shell-xg-t> write_milkyway -output file_name -overwrite
```

For more information, see the man page.

Example

To write design information from memory to a Milkyway library named testmw and name the design file TOP, enter

```
dc_shell-xg-t> set mw_design_library testmw
dc_shell-xg-t> write_milkyway -output TOP
```

If you plan to read the Milkyway design file created by `write_milkyway` into JupiterXT or Astro, see the JupiterXT or Astro documentation for the requirements.

Important Points About the write_milkyway Command

When you use the `write_milkyway` command, keep the following points in mind:

- You must run `create_mw_design` before you run `write_milkyway`.

  Optionally, if the Milkyway design already exists, you can use the `set_mw_design` command. See “Setting the Milkyway Design Library for Writing an Existing Milkyway Database” on page 10-12.

- If a design file already exists (that is, you ran `write_milkyway` more than once on the design with the same output directory), `write_milkyway` creates a new, additional design file and increments the version number. You must make sure you open the correct version in Milkyway; by default Milkyway opens the latest version. To avoid creating an additional version, use the `-overwrite` switch to overwrite the current version of the design file and save disk space.
- The command does not modify in-memory data.
- Attributes present in the design in memory that have equivalent attributes in Milkyway are translated (not all attributes present in the design database are translated).
- A hierarchical netlist translated using `write_milkyway` retains its hierarchy in the Milkyway database.

Results of Running the write_milkyway Command

The `write_milkyway` command does the following:

- Creates a design file based on the netlist in memory and saves the design data for the current design in the file. The path for the design file is *design_dir*/CEL/*file_name*:*version*, where *design_dir* is the location you specified in `mw_design_library`.
- Re-creates the hierarchy preservation information.

Limitations When Writing Milkyway Format

The following limitations apply when you write your design in Milkyway format:

- The design must be mapped.

  Because the Milkyway format describes physical information, it supports mapped designs only. You cannot use the Milkyway format to store design data for unmapped designs.

- The design must not contain multiple instances.

  You must uniquify your design before saving it in Milkyway format. Use the `check_design -multiple_designs` command to report information related to multiply-instantiated designs.

- The `write_milkyway` command saves the entire hierarchical design in a single Milkyway design file. You cannot generate separate design files for each subdesign.

- When you save a design in Milkyway format, the `write_milkyway` command does not save the interface logic model (ILM) instances in the Milkyway design library. You must explicitly save each ILM (see the *Interface Logic Model User Guide*).
- The constraints saved in the Milkyway design library cannot be read by versions of JupiterXT or Astro prior to W-2004.12.

  If you are using one of these tools, you must input the constraints by reading the golden SDC file for the design after reading the Milkyway design library.

Script to Set Up and Write a Milkyway Database

Example 10-1 is a script to set up and write a Milkyway database.

Example 10-1 Script to Set Up and Write a Milkyway Database

```
dc_shell-xg-t> set search_path [concat ../51_library $search_path]
dc_shell-xg-t> set synthetic_library {dw01.sldb dw02.sldb dw0-3.sldb dw04.sldb \
                   dw05.sldb dw06.sldb dw07.sldb dw08.sldb dw_foundation.sldb}
dc_shell-xg-t> set target_library "cells.db rams.db"
dc_shell-xg-t> set link_library [concat * $target_library $synthetic_library]
dc_shell-xg-t> set mw_reference_library [list ../milky_51m_library]
dc_shell-xg-t> set mw_design_library myDesignLib
dc_shell-xg-t> create_mw_design -tech_file tech.tf
dc_shell-xg-t> read_file -format ddc design.ddc
dc_shell-xg-t> current_design TopDesign
dc_shell-xg-t> link
dc_shell-xg-t> write_milkyway -output myTop
```

Maintaining the Milkyway Design Library

To maintain your Milkyway design library (for example, to delete unneeded versions of your design), you must use the Milkyway tool. This tool is a graphical user interface (GUI) that enables manipulation of the Milkyway libraries.

To invoke the Milkyway tool, enter

```
% Milkyway -galaxy
```

For information about using the Milkyway tool, see the Milkyway documentation.

Setting the Milkyway Design Library for Writing an Existing Milkyway Database

If the Milkyway design library already exists, run the `set_mw_design` command before you run the `write_milkyway` command to set the Milkyway design library directory.

When you use `set_mw_design`, keep the following points in mind:

- The Milkyway design library must already exist (created previously by `create_mw_design`).
- Specify the `mw_reference_library` and `mw_design_library` variables. The `mw_reference_library` variable is used to set the Milkyway reference libraries for the design and to enhance the `search_path` variable.

11

Analyzing and Resolving Design Problems

Use the reports generated by Design Compiler to analyze and debug your design. You can generate reports both before and after you compile your design. Generate reports before compiling to check that you have set attributes, constraints, and design rules properly. Generate reports after compiling to analyze the results and debug your design.

This chapter contains the following sections:

- Analyzing Timing
- Resolving Specific Problems

Checking for Design Consistency

A design is consistent when it does not contain errors such as unconnected ports, constant-valued ports, cells with no input or output pins, mismatches between a cell and its reference, multiple driver nets, connection class violations, or recursive hierarchy definitions.

Design Compiler runs the `-check_design -summary` command on all designs that are compiled; however, you can also use the command explicitly to verify design consistency. The command reports a list of warning and error messages.

- It reports an error if it finds a problem that Design Compiler cannot resolve. For example, recursive hierarchy (when a design references itself) is an error. You cannot compile a design that has `check_design` errors.
- It reports a warning if it finds a problem that indicates a corrupted design or a design mistake not severe enough to cause the `compile` command to fail.

  For a complete list of error and warning messages issued by the `check_design` command, see the manpage. Use options to the `check_design` command as follows:

To do this	Use this
To perform checks at only the current level of hierarchy (by default, the `check_design` command validates the entire design hierarchy)	`-one_level`
Disable warning messages	`-no_warnings`
Display a summary of warning messages instead of one message per warning	`-summary`

To do this	Use this
Report information messages related to multiply instantiated designs. When you use this option, a list of multiply instantiated designs along with instance names and associated attributes (`dont_touch`, `black_box`, and `ungroup`) are displayed. By default, messages related to multiply instantiated designs are suppressed.	`-multiple_designs`
Suppress warning messages related to connection class violations before compile. Use this option when you are working on a GTECH design or netlist in which connection class violations are expected. Doing so improves runtime (especially if the GTECH design is large and has several connection class violations).	`-no_connection_class`

By default, during compile or execution of the `check_design` command, Design Compiler issues a warning message if a tri-state bus is driven by a non tri-state driver. You can have Design Compiler display an error message instead by setting the `check_design_allow_non_tri_drivers_on_tri_bus` variable to false. The default is true. When the variable is set to false, the `compile` command stops after reporting the error; however, the `check_design` command continues to run after the error is reported.

Analyzing Your Design During Optimization

Design Compiler provides the following capabilities for analyzing your design during optimization:

- It lets you customize the compile log.

- It lets you save intermediate design databases.

The following sections describe these capabilities.

Customizing the Compile Log

The compile log records the status of the compile run. Each optimization task has an introductory heading, followed by the actions taken while that task is performed. There are three tasks in which Design Compiler works to reduce the compile cost function:

- Delay optimization
- Design rule fixing
- Area optimization

While completing these tasks, Design Compiler performs many trials to determine how to reduce the cost function. For this reason, these tasks are collectively known as the trials phase of optimization.

By default, Design Compiler logs each action in the trials phase by providing the following information:

- Elapsed time
- Design area
- Worst negative slack
- Total negative slack
- Design rule cost
- Endpoint being worked on

You can customize the trials phase output by setting the `compile_log_format` variable. Table 11-1 lists the available data items and the keywords used to select them. For more information about customizing the compile log, see the man page for the `compile_log_format` variable.

Table 11-1 Compile Log Format Keywords

Column	Column header	Keyword	Column description
Area	AREA	area	Shows the area of the design.
CPU seconds	CPU SEC	cpu	Shows the process CPU time used (in seconds).
Design rule cost	DESIGN RULE COST	drc	Measures the difference between the actual results and user-specified design rule constraints.
Elapsed time	ELAPSED TIME	elap_time	Tracks the elapsed time since the beginning of the current compile or reoptimization of the design.
Endpoint	ENDPOINT	endpoint	Shows the endpoint being worked on. When delay violations are being fixed, the endpoint is a cell or a port. When design rule violations are being fixed, the endpoint is a net. When area violations are being fixed, no endpoint is printed.
Maximum delay cost	MAX DELAY COST	max_delay	Shows the maximum delay cost of the design.
Megabytes of memory	MBYTES	mem	Shows the process memory used (in MB).
Minimum delay cost	MIN DELAY COST	min_delay	Shows the minimum delay cost of the design.

Table 11-1 Compile Log Format Keywords (Continued)

Column	Column header	Keyword	Column description
Path group	PATH GROUP	group_path	Shows the path group of an endpoint.
Time of day	TIME OF DAY	time	Shows the current time.
Total negative slack	TOTAL NEG SLACK	tns	Shows the total negative slack of the design.
Trials	TRIALS	trials	Tracks the number of transformations that the optimizer tried before making the current selection.
Worst negative slack	WORST NEG SLACK	wns	Shows the worst negative slack of the current path group.

Saving Intermediate Design Databases

Design Compiler provides the capability to output an intermediate design database during the trials phase of the optimization process. This capability is called checkpointing. Checkpointing saves the entire hierarchy of the intermediate design. You can use this intermediate design to debug design problems, as described in “Analyzing Design Problems” on page 11-8. To checkpoint automatically between each phase of compile, set the `compile_checkpoint_phases` variable to true.

Analyzing Design Problems

Table 11-2 shows the design analysis commands provided by Design Compiler. For additional information about these commands, see the man pages.

Table 11-2 Commands to Analyze Design Objects

Object	Command	Description
Design	`report_design`	Reports design characteristics.
	`report_area`	Reports design size and object counts.
	`report_hierarchy`	Reports design hierarchy.
	`report_resources`	Reports resource implementations.
Instances	`report_cell`	Displays information about instances.
References	`report_reference`	Displays information about references.
Pins	`report_transitive_fanin`	Reports fanin logic.
	`report_transitive_fanout`	Reports fanout logic.
Ports	`report_port`	Displays information about ports.
	`report_bus`	Displays information about bused ports.
	`report_transitive_fanin`	Reports fanin logic.
	`report_transitive_fanout`	Reports fanout logic.
Nets	`report_net`	Reports net characteristics.
	`report_bus`	Reports bused net characteristics.
	`report_transitive_fanin`	Reports fanin logic.
	`report_transitive_fanout`	Reports fanout logic.
Clocks	`report_clock`	Displays information about clocks.

Analyzing Area

Use the `report_area` command to display area information and statistics for the current design or instance. The command reports combinational, non-combinational, and total area. If you have set the

current instance, the report is generated for the design of that instance; otherwise, the report is generated for the current design. Use the `-hierarchy` option to report area used by cells across the hierarchy.

Analyzing Timing

Use the `report_timing` command to generate timing reports for the current design or the current instance. By default, the command lists the full path of the longest maximum delay timing path for each path group. (Design Compiler groups paths based on the clock controlling the endpoint. All paths not associated with a clock are in the default path group. You can also create path groups by using the `group_path` command).

Before you begin debugging timing problems, verify that your design meets the following requirements:

- You have defined the operating conditions.
- You have specified realistic constraints.
- You have appropriately budgeted the timing constraints.
- You have properly constrained the paths.
- You have described the clock skew.

If your design does not meet these requirements, make sure it does before you proceed.

After producing the initial mapped netlist, use the `report_constraint` command to check your design's performance.

Table 11-3 lists the timing analysis commands.

Table 11-3 Timing Analysis Commands

Command	Analysis task description
`report_design`	Shows operating conditions, wire load model and mode, timing ranges, internal input and output, and disabled timing arcs.
`check_timing`	Checks for unconstrained timing paths and clock-gating logic.
`report_port`	Shows unconstrained input and output ports and port loading.
`report_timing_requirements`	Shows all timing exceptions set on the design.
`report_clock`	Checks the clock definition and clock skew information.
`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.

Resolving Specific Problems

This section provides examples of design problems you might encounter and describes the workarounds for them.

Analyzing Cell Delays

Some cell delays shown in the full path timing report might seem too large. Use the `report_delay_calculation` command to determine how Design Compiler calculated a particular delay value.

Example 11-1 shows a full path timing report with a large cell delay value.

Example 11-1 Full Path Timing Report

```
****************************************
Report : timing
         -path full
         -delay max
         -max_paths 1
Design : Adder8
Version: Y-2006.06
Date   : Mon May 1 10:56:49 2006
****************************************

Operating Conditions:
Wire Loading Model Mode: top

  Startpoint: cin (input port)
  Endpoint: cout (output port)
  Path Group: (none)
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  input external delay                     0.00       0.00 f
  cin (in)                                 0.00       0.00 f
  U19/Z (AN2)                              0.87       0.87 f
  U18/Z (EO)                               1.13       2.00 f
  add_8/U1_1/CO (FA1A)                     2.27       4.27 f
  add_8/U1_2/CO (FA1A)                     1.17       5.45 f
  add_8/U1_3/CO (FA1A)                     1.17       6.62 f
  add_8/U1_4/CO (FA1A)                     1.17       7.80 f
  add_8/U1_5/CO (FA1A)                     1.17       8.97 f
  add_8/U1_6/CO (FA1A)                     1.17      10.14 f
  add_8/U1_7/CO (FA1A)                     1.17      11.32 f
  U2/Z (EO)                                1.06      12.38 f
  cout (out)                               0.00      12.38 f
  data arrival time                                        12.38 f
  -----------------------------------------------------------
  (Path is unconstrained)
```

The delay from port cin through cell FA1A seems large (2.27 ns). Enter the following command to determine how Design Compiler calculated this delay:

```
dc_shell-xg-t> report_delay_calculation \
            -from add_8/U1_1/A -to add_8/U1_1/CO
```

Example 11-2 shows the results of this command.

Example 11-2 Delay Calculation Report

```
****************************************
Report : delay_calculation
Design : Adder8
Version: Y-2006.06
Date   : Mon May 1 10:56:49 2006
****************************************

From pin:                          add_8/U1_1/A
To pin:                            add_8/U1_1/CO

arc sense:                         unate
arc type:                          cell
Input net transition times:        Dt_rise = 0.1458, Dt_fall = 0.0653

Rise Delay computation:
rise_intrinsic                     1.89 +
rise_slope * Dt_rise               0 * 0.1458 +
rise_resistance * (pin_cap + wire_cap) / driver_count
0.1458 * (2 + 0) / 1
-----------------------------------------
Total                              2.1816

Fall Delay computation:
fall_intrinsic                     2.14 +
fall_slope * Dt_fall               0 * 0.0653 +
fall_resistance * (pin_cap + wire_cap) / driver_count
0.0669 * (2 + 0) / 1
-----------------------------------------
Total                              2.2738
```

Finding Unmapped Cells

All unmapped cells have the `is_unmapped` attribute. You can use the dctcl `get_cells` command to locate all unmapped components:

```
dc_shell-xg-t> get_cells -hier -filter "@is_unmapped==true"
```

Finding Black Box Cells

All black box cells have the `is_black_box` attribute. You can use the `get_cells` command to locate all black box cells:

```
dc_shell-xg-t> get_cells -hier -filter\
                  "is_black_box==true"
```

Finding Hierarchical Cells

All hierarchical cells have the `is_hierarchical` attribute. You can use the `get_designs` command to locate all hierarchical cells:

```
dc_shell-xg-t> get_designs -filter "is_hierarchical==true"
```

Disabling Reporting of Scan Chain Violations

If your design contains scan chains, it is likely that these chains are not designed to run at system speed. This can cause false violation messages when you perform timing analysis. To mask these messages, use the `set_disable_timing` command to break the scan-related timing paths (scan input to scan output and scan enable to scan output).

```
dc_shell-xg-t> set_disable_timing my_lib/scanf \
          -from TI -to Q

dc_shell-xg-t> set_disable_timing my_lib/scanf \
          -from CP -to TE
```

This example assumes that

- scanf is the scan cell in your technology library
- TI is the scan input pin on the scanf cell
- TE is the scan enable on the scanf cell
- Q is the scan output pin on the scanf cell

Example 11-3 shows the script that you can use to identify the scan pins in your technology library.

Example 11-3 Script to Identify Scan Pins

```
set seq_cell_list [get_cells class/* -filter "@is_sequential==true"]
foreach_in_collection seq_cell $seq_cell_list {
   set seq_pins "[get_object_name $seq_cell]/*"
   set si [get_pins $seq_pins -filter "@signal_type==test_scan_in"]
if {[sizeof_collection $si] > 0} then {
      echo "Scan pins for cell [get_object_name $seq_cell]"
      echo "   scan input: [get_object_name $si]"
      echo "   scan output: [get_object_name [get_pins $seq_pins \
         -filter "@signal_type==test_scan_out"]]"
   }
}
```

Insulating Interblock Loading

Design Compiler determines load distribution in the driving block. If a single output port drives many blocks, a huge incremental cell delay can result. To insulate the interblock loading, fan the heavily loaded net to multiple output ports in the driving block. Evenly divide the total load among these output ports.

Preserving Dangling Logic

By default, Design Compiler optimizes away dangling logic. Use one of the following methods to preserve dangling logic (for example, spare cells) during optimization:

- Place the `dont_touch` attribute on the dangling logic.
- Connect the dangling logic to a dummy port.

Preventing Wire Delays on Ports

If your design contains unwanted wire delays between ports and I/O cells, you can remove these wire delays by specifying zero resistance (infinite drive strength) on the net. Use the `set_resistance` command to specify the net resistance. For example, enter the following command:

```
dc_shell-xg-t> set_resistance 0 [get_nets wire_io4]
```

Breaking a Feedback Loop

Follow these steps to break a feedback loop in your design:

1. Find the feedback loop in your design by using the `report_timing -loop` option.
2. Break the feedback loop by specifying the path as a false path.

Analyzing Buffer Problems

Note:

This section uses the term *buffer* to indicate either a buffer or an inverter chain.

This section describes the following topics:

- Buffer insertion behavior
- Missing buffer problems
- Extra buffer problems
- Hanging buffer problems
- Modified buffer network problems

Understanding Buffer Insertion

Design Compiler inserts buffers to correct maximum fanout load or maximum transition time violations. If Design Compiler does not insert buffers during optimization, the tool probably does not identify a violation. For more information about the maximum fanout load and maximum transition time design rules, see “Setting Design Rule Constraints” on page 7-3.

Use the `report_constraint` command to get details on constraint violations.

Figure 11-1 shows a design containing the IV1 cell.

Figure 11-1 Buffering Example

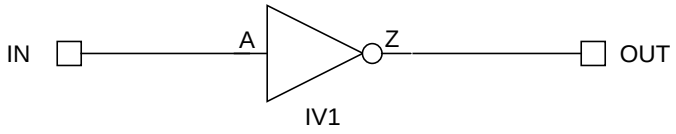


Table 11-4 gives the attributes defined in the technology library for the IV1 cell.

Table 11-4 IV1 Library Attributes

Pin	Attribute	Value
	direction	input
A	capacitance	1.5
	fanout_load	1
	direction	output
Z	rise_resistance	0.75
	fall_resistance	0.75
	max_fanout	3
	max_transition	2.5

Example 11-4 shows the constraint report generated by the following command sequence:

```
set_drive 0 [get_ports IN]
set_load 0 [get_ports OUT]
report_constraint
```

Example 11-4 Constraint Report

```
****************************************
Report : constraint
Design : buffer_example
Version: Y-2006.06
Date   : Mon May 1 10:56:49 2006
****************************************

    Constraint                                         Cost
    -------------------------------------------------------
   max_transition                                0.00 (MET)
   max_fanout                                    0.00 (MET)
```

To see the constraint cost functions used by Design Compiler, specify the -verbose option of the report_constraint command (shown in Example 11-5).

Example 11-5 Constraint Report (-verbose)

```
****************************************
Report : constraint
     -verbose
Design : buffer_example
Version: Y-2006.06
Date   : Mon May 1 10:56:49 2006
****************************************


    Net: OUT

    max_transition          2.50
  - Transition Time         0.00
  -------------------------------
    Slack                   2.50  (MET)

    Net: OUT

    max_fanout              3.00
  - Fanout                  0.00
  -------------------------------
    Slack                   3.00  (MET)
```

The verbose constraint report shows that two constraints are measured:

- Maximum transition time (2.50)
- Maximum fanout load (3.00)

Design Compiler derives the constraint values from the attribute values on the output pin of the IV1 cell.

When you compile this design, Design Compiler does not modify the design because the design meets the specified constraints.

To list all constraint violations, use the `-all_violators` option of the `report_constraint` command (shown in Example 11-6).

Example 11-6 Constraint Report (-all_violators)

```
****************************************
Report : constraint
    -all_violators
Design : buffer_example
Version: Y-2006.06
Date   : Mon May 1 10:56:49 2006
****************************************

This design has no violated constraints.
```

This design does not have any constraint violations. Changing the port attributes, however, can cause constraint violations to occur. Example 11-7 shows the result of the following command sequence:

```
dc_shell-xg-t> set_drive 2.5 IN
dc_shell-xg-t> set_max_fanout 0.75 IN
dc_shell-xg-t> set_load 4 OUT
dc_shell-xg-t> set_fanout_load 3.5 OUT
dc_shell-xg-t> report_constraint -all_violators -verbose
```

Example 11-7 Constraint Report (After Port Attributes Are Modified)

```
****************************************
Report : constraint
     -all_violators
     -verbose
Design : buffer_example
Version: Y-2006.06
Date   : Mon May 1 10:56:49 2006
****************************************

    Net: OUT

    max_transition          2.50
  - Transition Time         3.00
  -----------------------------
    Slack                  -0.50  (VIOLATED)

    Net: OUT

    max_fanout              3.00
  - Fanout                  3.50
  -----------------------------
    Slack                  -0.50  (VIOLATED)

    Net: IN

    max_fanout              0.75
  - Fanout                  1.00
  -----------------------------
    Slack                  -0.25  (VIOLATED)
```

This design now contains three violations:

- Maximum transition time violation at OUT

  Actual transition time is 4.00 * 0.75 = 3.00, which is greater than the maximum transition time of 2.50.

- Maximum fanout load violation at OUT

  Actual fanout load is 3.5, which is greater than the maximum fanout load of 3.00.

- Maximum fanout load violation at IN

  Actual fanout load is 1.00, which is greater than the maximum fanout load of 0.75.

There is no `max_transition` violation at IN, even though the transition time on this net is 2.5 * 1.5 = 3.75, which is well above the `max_transition` requirement of 2.50. Design Compiler does not recognize this as a violation because the requirement of 2.50 is a design rule from the output pin of cell IV1. This requirement applies only to a net driven by this pin. The IV1 output pin does not drive the net connected to port IN, so the `max_transition` constraint does not apply to this net.

If you want to constrain the net attached to port IN to a maximum transition time of 2.50, enter the following command:

```
dc_shell-xg-t> set_max_transition 2.5 [get_ports IN]
```

This command causes `report_constraint -verbose -all_violators` to add the following lines to the report shown in Example 11-7:

```
Net: IN
max_transition          2.50
- Transition Time       3.75
---------------------------
Slack                -1.25  (VIOLATED)
```

When you compile this design, Design Compiler adds buffering to correct the `max_transition` violations.

Remember the following points when you work with buffers in Design Compiler:

- The `max_fanout` and `max_transition` constraints control buffering; be sure you understand how each is used.
- Design Compiler fixes only violations it detects.
- The `report_constraint` command identifies any violations.

Correcting for Missing Buffers

Missing buffers present the most frequent buffering problem. It usually results from one of the following conditions:

- Incorrectly specified constraints
- Improperly constrained designs
- Incorrect assumptions about constraint behavior

To debug the problem, generate a constraint report (`report_constraint`) to determine whether Design Compiler recognized any violations.

If Design Compiler reports no `max_fanout` or `max_transition` violations, check the following:

- Are constraints applied?
- Is the library modeled for the correct attributes?
- Are the constraints tight enough?

If Design Compiler recognizes a violation but `compile` does not insert buffers to remove the violation, check the following:

- Does the violation exist after compile?

- Are there dont_touch or dont_touch_network attributes?
- Are there three-state pins that require buffering?
- Have you considered that max_transition takes precedence over max_fanout?

Incorrectly Specified Constraints. A vendor might omit an attribute you want to use, such as fanout_load. If a vendor has not set this attribute in the library, Design Compiler does not find any violations for the constraint. You can check whether attributes have been assigned to cell pins by using the get_attribute command with the get_pins command. For example, to determine whether a pin has a fanout_load attribute, enter

```
dc_shell-xg-t> get_attribute \
          [get_pins library/cell/pin] fanout_load
```

The vendor might have defined default_fanout_load in the library. If this value is set to zero or to an extremely small number, any pin that does not have an explicit fanout_load attribute inherits this value.

Improperly Constrained Designs. Occasionally, a vendor uses extremely small capacitance values (on the order of 0.001). If your scripts do not take this into account, you might not be constraining your design tightly enough. Try setting an extreme value, such as 0.00001, and run report_constraint to make sure a violation occurs.

You can use the load_of command with the get_pins command to check the capacitance values in the technology library:

```
dc_shell-xg-t> load_of [get_pins library/cell/pin]
```

Incorrect Assumptions About Constraint Behavior. Check to make sure you are not overlooking one of the following aspects of constraint behavior:

- A common mistake is the assumption that the `default_max_transition` or the `default_max_fanout` constraint in the technology library applies to input ports. These constraints apply only to the output pins of cells within the library.
- Maximum transition time takes precedence over maximum fanout load within Design Compiler. Therefore, a maximum fanout violation might not be corrected if the correction affects the maximum transition time of a net.
- Design Compiler might have removed a violation by sizing gates or modifying the structure of the design.

  Generate a constraint report after optimization to verify that the violation still exists.

- Design Compiler cannot correct violations if `dont_touch` attributes exist on the violating path.

  You might have inadvertently placed `dont_touch` attributes on a design or cell reference within the hierarchy. If so, Design Compiler reports violations but cannot correct them during optimization.

  Use the `report_cell` command and the `get_attribute` command to see whether these attributes exist.

- Design Compiler cannot correct violations if `dont_touch_network` attributes exist on the violating path.

If you have set the `dont_touch_network` attribute on a port or pin in the design, all elements in the transitive fanout of that port or pin inherit the attribute. If this attribute is set, Design Compiler reports violations but does not modify the network during optimization.

Use the `remove_attribute` command to remove this attribute from the port or net.

- Design Compiler does not support additional buffering on three-state pins.

  For simple three-state cells, Design Compiler attempts to enlarge a three-state cell to a stronger three-state cell.

  For complex three-state cells, such as sequential elements or RAM cells, Design Compiler cannot build the logic necessary to duplicate the required functionality. In such cases, you must manually add the extra logic or rewrite the source HDL to decrease the fanout load of such nets.

Correcting for Extra Buffers

Extremely conservative numbers for `max_transition`, `max_fanout`, or `max_capacitance` force Design Compiler to buffer nets excessively. If your design has an excessive number of buffers, check the accuracy of the design rule constraints applied to the design.

If you have specified design rule constraints that are more restrictive than those specified in the technology library, evaluate the necessity for these restrictive design rules.

You can debug this type of problem by setting the priority of the maximum delay cost function higher than the maximum design rule cost functions (using the `set_cost_priority -delay` command). Changing the priority prevents Design Compiler from fixing the maximum design rule violations if the fix results in a timing violation.

Correcting for Hanging Buffers

A buffer that does not fan out to any cells is called a hanging buffer. Hanging buffers often occur because the buffer cells have `dont_touch` attributes. These attributes either can be set by you, in the hope of retaining a buffer network, or can be inherited from a library.

The `dont_touch` attribute on a cell signals to Design Compiler that the cell should not be touched during optimization. Design Compiler follows these instructions by leaving the cell in the design. But because the buffer might not be needed to meet the constraints that are set, Design Compiler disconnects the net from the output. The design meets your constraints, but because the cell has the `dont_touch` attribute, the cell cannot be removed. Remove the `dont_touch` attribute to correct this problem.

Correcting Modified Buffer Networks

Sometimes it appears that Design Compiler modifies a buffer network that has `dont_touch` attributes. This problem usually occurs when you place the `dont_touch` attribute on a cell and expect the cells adjacent to that cell to remain in the design.

Design Compiler does not affect the cell itself but modifies the surrounding nets and cells to attain the optimal structure. If you are confident about the structure you want, you can use one of the following strategies to preserve your buffer network:

- Group the cells into a new hierarchy and set `dont_touch` attributes on that hierarchy.
- Set the `dont_touch_network` attribute on the pin that begins the network.
- Set the `dont_touch` attribute on all cells and nets within the network that you want to retain.

A

Design Example

Optimizing a design can involve using different compile strategies for different levels and components in the design. This appendix shows a design example that uses several compile strategies. Earlier chapters provide detailed descriptions of how to implement each compile strategy. Note that the design example used in this appendix does not represent a real-life application.

This appendix includes the following sections:

You can access the files described in these sections at $SYNOPSYS/doc/syn/guidelines.

Design Description

The design example shows how you can constrain designs by using a subset of the commonly used dc_shell commands and how you can use scripts to implement various compile strategies.

The design uses synchronous RTL and combinational logic with clocked D flip-flops.

Figure A-1 shows the block diagram for the design example. The design contains seven modules at the top level: Adder16, CascadeMod, Comparator, Multiply8x8, Multiply16x16, MuxMod, and PathSegment.

Figure A-1 Block Diagram for the Design Example

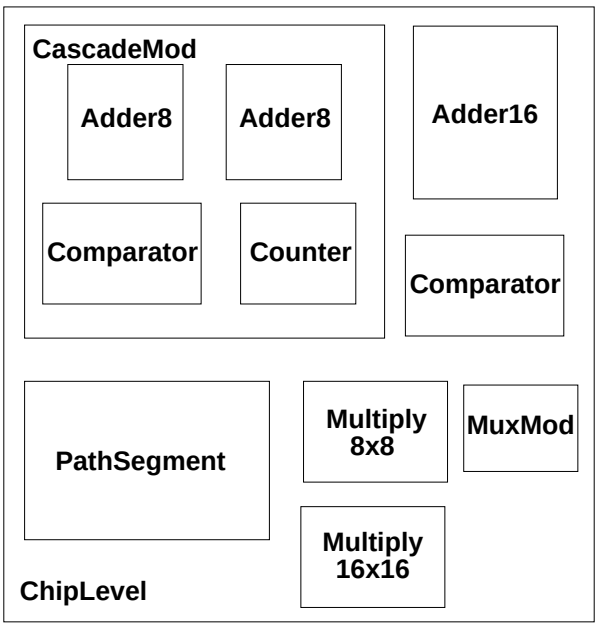
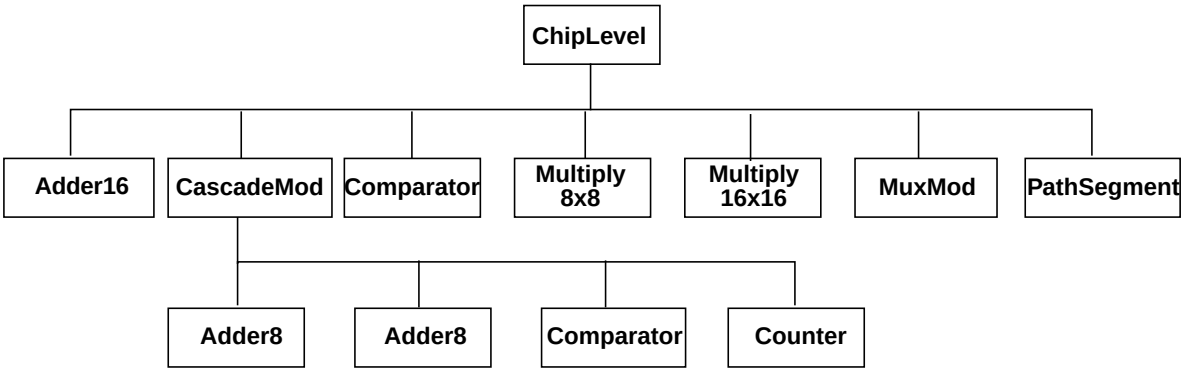


Figure A-2 shows the hierarchy for the design example.

Figure A-2 Hierarchy for the Design Example



The top-level modules and the compilation strategies for optimizing them are

Adder16

Uses registered outputs to make constraining easier. Because the endpoints are the data pins of the registers, you do not need to set output delays on the output ports.

CascadeMod

Uses a hierarchical compile strategy. The compile script for this design sets the constraints at the top level (of CascadeMod) before compilation.

The CascadeMod design instantiates the Adder8 design twice. The script uses the compile-once-don’t-touch method for the Comparator module.

Comparator

Is a combinational block. The compile script for this design uses the virtual clock concept to show the use of virtual clocks in a design.

The ChipLevel design instantiates Comparator twice. The compile script (for CascadeMod) uses the compile-once-don't-touch method to resolve the multiple instances.

The compile script specifies wire load model and mode instead of using automatic wire load selection.

Multiply8x8

Shows the basic timing and area constraints used for optimizing a design.

Multiply16x16

Ungroups DesignWare parts before compilation. Ungrouping your hierarchical module might help achieve better synthesis results. The compile script for this module defines a two-cycle path at the primary ports of the module.

MuxMod

Is a combinational block. The script for this design uses the virtual clock concept.

PathSegment

Uses path segmentation within a module. The script uses the `set_multicycle_path` command for a two-cycle path within the module and the `group` command to create a new level of hierarchy.

Example A-1 through Example A-11 provide the Verilog source code for the ChipLevel design.

Example A-1 ChipLevel.v

```
/* Date: May 11, 1995 */
/* Example Circuit for Baseline Methodology for Synthesis */
/* Design does not show any real-life application but rather
   it is used to illustrate the commands used in the Baseline
   Methodology */

module ChipLevel (data16_a, data16_b, data16_c, data16_d, clk, cin, din_a,
                  din_b, sel, rst, start, mux_out, cout1, cout2, s1, s2, op,
                  comp_out1, comp_out2, m32_out, regout);

   input [15:0] data16_a, data16_b, data16_c, data16_d;
   input [7:0] din_a, din_b;
   input [1:0] sel;
   input clk, cin, rst, start;
   input s1, s2, op;
   output [15:0] mux_out, regout;
   output [31:0] m32_out;
   output cout1, cout2, comp_out1, comp_out2;

   wire [15:0] ad16_sout, ad8_sout, m16_out, cnt;

Adder16 u1 (.ain(data16_a), .bin(data16_b), .cin(cin), .cout(cout1),
            .sout(ad16_sout), .clk(clk));

CascadeMod u2 (.data1(data16_a), .data2(data16_b), .cin(cin), .s(ad8_sout),
               .cout(cout2), .clk(clk), .comp_out(comp_out1), .cnt(cnt),
               .rst(rst), .start(start) );

Comparator u3 (.ain(ad16_sout), .bin(ad8_sout), .cp_out(comp_out2));

Multiply8x8 u4 (.op1(din_a), .op2(din_b), .res(m16_out), .clk(clk));

Multiply16x16 u5 (.op1(data16_a), .op2(data16_b), .res(m32_out), .clk(clk));

MuxMod  u6 (.Y_IN(mux_out), .MUX_CNT(sel), .D(ad16_sout), .R(ad8_sout),
            .F(m16_out), .UPC(cnt));

PathSegment u7 (.R1(data16_a), .R2(data16_b), .R3(data16_c), .R4(data16_d),
                .S2(s2), .S1(s1), .OP(op), .REGOUT(regout), .clk(clk));
endmodule
```

Example A-2 Adder16.v

```
module Adder16 (ain, bin, cin, sout, cout, clk);
/* 16-Bit Adder Module */
output [15:0] sout;
output cout;
input [15:0] ain, bin;
input cin, clk;

wire [15:0] sout_tmp, ain, bin;
wire cout_tmp;
reg [15:0] sout, ain_tmp, bin_tmp;
reg cout, cin_tmp;

always @(posedge clk) begin
   cout = cout_tmp;
   sout = sout_tmp;
   ain_tmp = ain;
   bin_tmp = bin;
   cin_tmp = cin;
end
   assign {cout_tmp,sout_tmp} = ain_tmp + bin_tmp + cin_tmp;
endmodule
```

Example A-3 CascadeMod.v

```
module CascadeMod (data1, data2, s, clk, cin, cout, comp_out, cnt, rst, start);
input [15:0] data1, data2;
output [15:0] s, cnt;
input clk, cin, rst, start;
output cout, comp_out;
wire co;

Adder8 u10 (.ain(data1[7:0]), .bin(data2[7:0]), .cin(cin), .clk(clk),
.sout(s[7:0]), .cout(co));
Adder8 u11 (.ain(data1[15:8]), .bin(data2[15:8]), .cin(co), .clk(clk),
.sout(s[15:8]), .cout(cout));
Comparator u12 (.ain(s), .bin(cnt), .cp_out(comp_out));

Counter u13 (.count(cnt), .start(start), .clk(clk), .rst(rst));
endmodule
```

Example A-4 Adder8.v

```
module Adder8 (ain, bin, cin, sout, cout, clk);
/* 8-Bit Adder Module */
output [7:0] sout;
output cout;
input [7:0] ain, bin;
input cin, clk;

wire [7:0] sout_tmp, ain, bin;
wire cout_tmp;
reg [7:0] sout, ain_tmp, bin_tmp;
reg cout, cin_tmp;

always @(posedge clk) begin
   cout = cout_tmp;
   sout = sout_tmp;
   ain_tmp = ain;
   bin_tmp = bin;
   cin_tmp = cin;
end
  assign {cout_tmp,sout_tmp} = ain_tmp + bin_tmp + cin_tmp;
endmodule
```

Example A-5 Counter.v

```
module Counter (count, start, clk, rst);
/* Counter module */
  input clk;
  input rst;
  input start;
  output [15:0] count;

  wire clk;
  reg [15:0] count_N;
  reg [15:0] count;

  always @ (posedge clk or posedge rst)
    begin : counter_S
      if (rst) begin
        count = 0; // reset logic for the block
        end
      else begin
        count = count_N; // set specified registers of the block
        end
    end

  always @ (count or start)
    begin : counter_C
      count_N = count; // initialize outputs of the block
      if (start) count_N = 1; // user specified logic for the block
      else count_N = count + 1;
    end
endmodule
```

Example A-6 Comparator.v

```
module Comparator (cp_out, ain, bin);
/* Comparator for 2 integer values */
output cp_out;
input [15:0] ain, bin;
   assign cp_out = ain < bin;
endmodule
```

Example A-7 Multiply8x8.v

```
module Multiply8x8 (op1, op2, res, clk);
/* 8-Bit multiplier */
input [7:0] op1, op2;
output [15:0] res;
input clk;

wire [15:0] res_tmp;
reg [15:0] res;

always @(posedge clk) begin
   res = res_tmp;
end
assign res_tmp = op1 * op2;
endmodule
```

Example A-8 Multiply16x16.v

```
module Multiply16x16 (op1, op2, res, clk);
/* 16-Bit multiplier */
input [15:0] op1, op2;
output [31:0] res;
input clk;

wire [31:0] res_tmp;
reg [31:0] res;

always @(posedge clk) begin
   res = res_tmp;
end
assign res_tmp = op1 * op2;
endmodule
```

Example A-9 def_macro.v

```
`define DATA 2'b00
`define REG 2'b01
`define STACKIN 2'b10
`define UPCOUT 2'b11
```

Example A-10 MuxMod.v

```
module  MuxMod (Y_IN, MUX_CNT, D, R, F, UPC);
`include "def_macro.v"
   output [15:0] Y_IN;
   input [ 1:0] MUX_CNT;
   input [15:0] D, F, R, UPC;

   reg [15:0] Y_IN;

always @ ( MUX_CNT or D or R or F or UPC ) begin
   case ( MUX_CNT )
      `DATA :
         Y_IN = D ;
      `REG :
         Y_IN = R ;
      `STACKIN :
         Y_IN = F ;
      `UPCOUT :
         Y_IN = UPC;
   endcase
end

endmodule
```

Example A-11 PathSegment.v

```
module PathSegment (R1, R2, R3, R4, S2, S1, OP, REGOUT, clk);
/* Example for path segmentation */
input [15:0] R1, R2, R3, R4;
input S2, S1, clk;
input OP;
output [15:0] REGOUT;

reg [15:0] ADATA, BDATA;
reg [15:0] REGOUT;
reg MODE;

wire [15:0] product ;

always @(posedge clk)
begin : selector_block
   case(S1)
            1'b0: ADATA <= R1;
            1'b1: ADATA <= R2;
            default: ADATA <= 16'bx;
   endcase
   case(S2)
            1'b0: BDATA <= R3;
            1'b1: BDATA <= R4;
            default: ADATA <= 16'bx;
   endcase
end

/* Only Lower Byte gets multiplied */
// instantiate DW02_mult
DW02_mult #(8,8) U100 (.A(ADATA[7:0]), .B(BDATA[7:0]), .TC(1'b0),
.PRODUCT(product));

always @(posedge clk)
begin : alu_block
   case (OP)
      1'b0 : begin
                REGOUT <= ADATA + BDATA;
             end
      1'b1 : begin
                 REGOUT <= product;
             end
      default : REGOUT <= 16'bx;
   endcase
end

endmodule
```

Setup File

When running the design example, copy the project-specific setup file in Example A-12 to your project working directory. This setup file is written in the Tcl subset and can be used in the dctcl command language. For more information about the Tcl subset, see *Using Tcl With Synopsys Tools* and the *Design Compiler Command-Line Interface Guide*.

For details on the synthesis setup files, see “Setup Files” on page 2-8.

Example A-12 .synopsys_dc.setup File

```
# Define the target technology library, symbol library,
# and link libraries
set target_library lsi_10k.db
set symbol_library lsi_10k.sdb
set link_library [concat $target_library "*"]
set search_path [concat $search_path ./src]
set designer "Your Name"
set company "Synopsys, Inc."
# Define path directories for file locations
set source_path "./src/"
set script_path "./scr/"
set log_path "./log/"
set ddc_path "./ddc/"
set db_path "./db/"
set netlist_path "./netlist/"
```

Default Constraints File

The file shown in Example A-13 defines the default constraints for the design. In the scripts that follow, Design Compiler reads this file first for each module. If the script for a module contains additional

constraints or constraint values different from those defined in the default constraints file, Design Compiler uses the module-specific constraints.

Example A-13 defaults.con

```
# Define system clock period
set clk_period 20

# Create real clock if clock port is found
if {[sizeof_collection [get_ports clk]] > 0} {
   set clk_name clk
   create_clock -period $clk_period clk
}

# Create virtual clock if clock port is not found
if {[sizeof_collection [get_ports clk]] == 0} {
   set clk_name vclk
   create_clock -period $clk_period -name vclk
}

# Apply default drive strengths and typical loads
# for I/O ports
set_load 1.5 [all_outputs]
set_driving_cell -lib_cell IV [all_inputs]

# If real clock, set infinite drive strength
if {[sizeof_collection [get_ports clk]] > 0} {
   set_drive 0 clk
}

# Apply default timing constraints for modules
set_input_delay 1.2 [all_inputs] -clock $clk_name
set_output_delay 1.5 [all_outputs] -clock $clk_name
set_clock_uncertainty -setup 0.45 $clk_name

# Set operating conditions
set_operating_conditions WCCOM

# Turn on auto wire load selection
# (library must support this feature)
set auto_wire_load_selection true
```

Compile Scripts

Example A-14 through Example A-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 A-14 run.tcl

```
# Initial compile with estimated constraints
source "${script_path}initial_compile.tcl"

current_design ChipLevel
if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}ChipLevel_init.db"
} else {
write -f ddc -hier -o "${ddc_path}ChipLevel_init.ddc"}

# Characterize and write_script for all modules
source "${script_path}characterize.tcl"

# Recompile all modules using write_script constraints
remove_design -all
source "${script_path}recompile.tcl"

current_design ChipLevel
if {[shell_is_in_xg_mode]==0}{
write -hier -out "${db_path}ChipLevel_final.db"
} else {
write -f ddc -hier -out "${ddc_path}ChipLevel_final.ddc"}
```

Example A-15 initial_compile.tcl

```
# Initial compile with estimated constraints
source "${script_path}read.tcl"

current_design ChipLevel
source "${script_path}defaults.con"

source "${script_path}adder16.tcl"
source "${script_path}cascademod.tcl"
source "${script_path}comp16.tcl"
source "${script_path}mult8.tcl"
source "${script_path}mult16.tcl"
source "${script_path}muxmod.tcl"
source "${script_path}pathseg.tcl"
```

Example A-16 adder16.tcl

```
# Script file for constraining Adder16
set rpt_file "adder16.rpt"
set design "adder16"

current_design Adder16
source "${script_path}defaults.con"

# Define design environment
set_load 2.2 sout
set_load 1.5 cout
set_driving_cell -lib_cell FD1 [all_inputs]
set_drive 0 $clk_name

# Define design constraints
set_input_delay 1.35 -clock $clk_name {ain bin}
set_input_delay 3.5 -clock $clk_name cin
set_max_area 0

compile

if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}${design}.db"
} else {
write -f ddc -hier -o "${ddc_path}${design}.ddc"}

source "${script_path}report.tcl"
```

Example A-17 cascademod.tcl

```
# Script file for constraining CascadeMod
# Constraints are set at this level and then a
# hierarchical compile approach is used

set rpt_file "cascademod.rpt"
set design "cascademod"

current_design CascadeMod
source "${script_path}defaults.con"

# Define design environment
set_load 2.5 [all_outputs]
set_driving_cell -lib_cell FD1 [all_inputs]
set_drive 0 $clk_name

# Define design constraints
set_input_delay 1.35 -clock $clk_name {data1 data2}
set_input_delay 3.5 -clock $clk_name cin
set_input_delay 4.5 -clock $clk_name {rst start}
set_output_delay 5.5 -clock $clk_name comp_out
set_max_area 0

# Use compile-once, dont_touch approach for Comparator
set_dont_touch u12

compile

if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}${design}.db"
} else {
write -f ddc -hier -o "${ddc_path}${design}.ddc"}

source "${script_path}report.tcl"
```

Example A-18 comp16.tcl

```
# Script file for constraining Comparator
set rpt_file "comp16.rpt"
set design "comp16"

current_design Comparator
source "${script_path}defaults.con"

# Define design environment
set_load 2.5 cp_out
set_driving_cell -lib_cell FD1 [all_inputs]

# Override auto wire load selection
set_wire_load_model -name "05x05"
set_wire_load_mode enclosed

# Define design constraints
set_input_delay 1.35 -clock $clk_name {ain bin}
set_output_delay 5.1 -clock $clk_name {cp_out}
set_max_area 0

compile

if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}${design}.db"
} else {
write -f ddc -hier -o "${ddc_path}${design}.ddc"}

source "${script_path}report.tcl"
```

Example A-19 mult8.tcl

```
# Script file for constraining Multiply8x8
set rpt_file "mult8.rpt"
set design "mult8"

current_design Multiply8x8
source "${script_path}defaults.con"

# Define design environment
set_load 2.2 res
set_driving_cell -lib_cell FD1P [all_inputs]
set_drive 0 $clk_name

# Define design constraints
set_input_delay 1.35 -clock $clk_name {op1 op2}
set_max_area 0

compile

if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}${design}.db"
} else {
write -f ddc -hier -o "${ddc_path}${design}.ddc"}

source "${script_path}report.tcl"
```

Example A-20 mult16.tcl

```
# Script file for constraining Multiply16x16
set rpt_file "mult16.rpt"
set design "mult16"

current_design Multiply16x16
source "${script_path}defaults.con"

# Define design environment
set_load 2.2 res
set_driving_cell -lib_cell FD1 [all_inputs]
set_drive 0 $clk_name

# Define design constraints
set_input_delay 1.35 -clock $clk_name {op1 op2}
set_max_area 0

# Define multicycle path for multiplier
set_multicycle_path 2 -from [all_inputs] \
   -to [all_registers -data_pins -edge_triggered]

# Ungroup DesignWare parts
set designware_cells [get_cells \
   -filter "@is_oper==true"]
if {[sizeof_collection $designware_cells] > 0} {
   set_ungroup $designware_cells true
}

compile

if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}${design}.db"
} else {
write -f ddc -hier -o "${ddc_path}${design}.ddc"}

source "${script_path}report.tcl"
report_timing_requirements -ignore \
   >> "${log_path}${rpt_file}"
```

Example A-21 muxmod.tcl

```
# Script file for constraining MuxMod
set rpt_file "muxmod.rpt"
set design "muxmod"

current_design MuxMod
source "${script_path}defaults.con"

# Define design environment
set_load 2.2 Y_IN
set_driving_cell -lib_cell FD1 [all_inputs]

# Define design constraints
set_input_delay 1.35 -clock $clk_name {D R F UPC}
set_input_delay 2.35 -clock $clk_name MUX_CNT
set_output_delay 5.1 -clock $clk_name {Y_IN}
set_max_area 0

compile

if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}${design}.db"
} else {
write -f ddc -hier -o "${ddc_path}${design}.ddc"}

source "${script_path}report.tcl"
```

Example A-22 pathseg.tcl

```
# Script file for constraining path_segment
set rpt_file "pathseg.rpt"
set design "pathseg"

current_design PathSegment
source "${script_path}defaults.con"

# Define design environment
set_load 2.5 [all_outputs]
set_driving_cell -lib_cell FD1 [all_inputs]
set_drive 0 $clk_name

# Define design rules
set_max_fanout 6 {S1 S2}

# Define design constraints
set_input_delay 2.2 -clock $clk_name {R1 R2}
set_input_delay 2.2 -clock $clk_name {R3 R4}
set_input_delay 5 -clock $clk_name {S2 S1 OP}
set_max_area 0

# Perform path segmentation for multiplier
group -design mult -cell mult U100
set_input_delay 10 -clock $clk_name mult/product*
set_output_delay 5 -clock $clk_name mult/product*
set_multicycle_path 2 -to mult/product*

compile

if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}${design}.db"
} else {
write -f ddc -hier -o "${ddc_path}${design}.ddc"}

source "${script_path}report.tcl"
report_timing_requirements -ignore \
   >> "${log_path}${rpt_file}"
```

Example A-23 characterize.tcl

```
# Characterize and write_script for all modules
current_design ChipLevel
characterize u1
current_design Adder16
write_script > "${script_path}adder16.wtcl"

current_design ChipLevel
characterize u2
current_design CascadeMod
write_script -format dctcl >
     "${script_path}cascademod.wtcl"

current_design ChipLevel
characterize u3
current_design Comparator
write_script -format dctcl > "${script_path}comp16.wtcl"

current_design ChipLevel
characterize u4
current_design Multiply8x8
write_script -format dctcl > "${script_path}mult8.wtcl"

current_design ChipLevel
characterize u5
current_design Multiply16x16
write_script -format dctcl > "${script_path}mult16.wtcl"

current_design ChipLevel
characterize u6
current_design MuxMod
write_script -format dctcl > "${script_path}muxmod.wtcl"

current_design ChipLevel
characterize u7
current_design PathSegment

echo "current_design PathSegment" > \
   "${script_path}pathseg.wtcl"

echo "group -design mult -cell mult U100" >> \
   "${script_path}pathseg.wtcl"
write_script -format dctcl >> "${script_path}pathseg.wtcl"
```

Example A-24 recompile.tcl

```
source "${script_path}read.tcl"

current_design ChipLevel
source "${script_path}defaults.con"

source "${script_path}adder16.wtcl"
compile
if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}adder16_wtcl.db"
} else {
write -f ddc -hier -o "${ddc_path}adder16_wtcl.ddc"}
set rpt_file adder16_wtcl.rpt
source "${script_path}report.tcl"

source "${script_path}cascademod.wtcl"
dont_touch u12
compile
if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}cascademod_wtcl.db"
} else {
write -f ddc -hier -o "${ddc_path}cascademod_wtcl.ddc"}
set rpt_file cascade_wtcl.rpt
source "${script_path}report.tcl"
```

```
source "${script_path}comp16.wtcl"
compile
if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}comp16_wtcl.db"
} else {
write -f ddc -hier -o "${ddc_path}comp16_wtcl.ddc"}
set rpt_file comp16_wtcl.rpt
source "${script_path}report.tcl"

source "${script_path}mult8.wtcl"
compile
if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}mult8_wtcl.db"
} else {
write -f ddc -hier -o "${ddc_path}mult8_wtcl.ddc"}
set rpt_file mult8_wtcl.rpt
source "${script_path}report.tcl"

source "${script_path}mult16.wtcl"
compile -ungroup_all
if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}mult16_wtcl.db"
} else {
write -f ddc -hier -o "${ddc_path}mult16_wtcl.ddc"}
set rpt_file mult16_wtcl.rpt
source "${script_path}report.tcl"
report_timing_requirements -ignore \
         >> "${log_path}${rpt_file}"

source "${script_path}muxmod.wtcl"
compile
if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}muxmod_wtcl.db"
} else {
write -f ddc -hier -o "${ddc_path}muxmod_wtcl.ddc"}
set rpt_file muxmod_wtcl.rpt
source "${script_path}report.tcl"
```

Example A-24 recompile.tcl (Continued)

```
source "${script_path}pathseg.wtcl"
compile
if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}pathseg_wtcl.db"
} else {
write -f ddc -hier -o "${ddc_path}pathseg_wtcl.ddc"}
set rpt_file pathseg_wtcl.rpt
source "${script_path}report.tcl"
report_timing_requirements -ignore \
   >> "${log_path}${rpt_file}"
```

Example A-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_compile_options >> "${log_path}${rpt_file}"
report_constraint -all_violators -verbose \
   >> "${log_path}${rpt_file}"
report_timing -path end >> "${log_path}${rpt_file}"
report_timing -max_path $maxpaths \
   >> "${log_path}${rpt_file}"
report_qor >> "${log_path}${rpt_file}"
```

B

Basic Commands

This appendix lists the basic dc_shell commands for synthesis and provides a brief description for each command. The commands are grouped in the following sections:

- Commands for Defining Design Rules
- Commands for Defining Design Environments
- Commands for Setting Design Constraints
- Commands for Analyzing and Resolving Design Problems

Within each section the commands are listed in alphabetical order.

Commands for Defining Design Rules

The commands that define design rules are

`set_max_capacitance`

Sets a maximum capacitance for the nets attached to the specified ports or to all the nets in a design.

`set_max_fanout`

Sets the expected fanout load value for output ports.

`set_max_transition`

Sets a maximum transition time for the nets attached to the specified ports or to all the nets in a design.

`set_min_capacitance`

Sets a minimum capacitance for the nets attached to the specified ports or to all the nets in a design.

Commands for Defining Design Environments

The commands that define the design environment are

`set_drive`

Sets the drive value of input or inout ports. The `set_drive` command is superseded by the `set_driving_cell` command.

`set_driving_cell`

Sets attributes on input or inout ports, specifying that a library cell or library pin drives the ports. This command associates a library pin with an input port so that delay calculators can accurately model the drive capability of an external driver.

`set_fanout_load`

Defines the external fanout load values on output ports.

`set_load`

Defines the external load values on input and output ports and nets.

`set_operating_conditions`

Defines the operating conditions for the current design.

`set_wire_load_model`

Sets the wire load model for the current design or for the specified ports. With this command, you can specify the wire load model to use for the external net connected to the output port.

Commands for Setting Design Constraints

The basic commands that set design constraints are

`create_clock`

Creates a clock object and defines its waveform in the current design.

`set_clock_latency, set_clock_uncertainty, set_propagated_clock, set_clock_transition`

Sets clock attributes on clock objects or flip-flop clock pins.

`set_input_delay`

Sets input delay on pins or input ports relative to a clock signal.

`set_max_area`

Specifies the maximum area for the current design.

`set_output_delay`

Sets output delay on pins or output ports relative to a clock signal.

The advanced commands that set design constraints are

`group_path`

Groups a set of paths or endpoints for cost function calculation. This command is used to create path groups, to add paths to existing groups, or to change the weight of existing groups.

`set_false_path`

Marks paths between specified points as false. This command eliminates the selected paths from timing analysis.

`set_max_delay`

Specifies a maximum delay target for selected paths in the current design.

`set_min_delay`

Specifies a minimum delay target for selected paths in the current design.

`set_multicycle_path`

Allows you to specify the time of a timing path to exceed the time of one clock signal.

Commands for Analyzing and Resolving Design Problems

The commands for analyzing and resolving design problems are

`all_connected`

Lists all fanouts on a net.

`all_registers`

Lists sequential elements or pins in a design.

`check_design`

Checks the internal representation of the current design for consistency and issues error and warning messages as appropriate.

`check_timing`

Checks the timing attributes placed on the current design.

`get_attribute`

Reports the value of the specified attribute.

`link`

Locates the reference for each cell in the design.

`report_area`

Provides area information and statistics on the current design.

`report_attribute`

Lists the attributes and their values for the selected object. An object can be a cell, net, pin, port, instance, or design.

`report_cell`

Lists the cells in the current design and their cell attributes.

`report_clock`

Displays clock-related information on the current design.

`report_constraint`

Lists the constraints on the current design and their cost, weight, and weighted cost.

`report_delay_calculation`

Reports the details of a delay arc calculation.

`report_design`

Displays the operating conditions, wire load model and mode, timing ranges, internal input and output, and disabled timing arcs defined for the current design.

`report_hierarchy`

Lists the children of the current design.

`report_net`

Displays net information for the design of the current instance, if set; otherwise, displays net information for the current design.

`report_path_group`

Lists all timing path groups in the current design.

`report_port`

Lists information about ports in the current design.

`report_qor`

Displays information about the quality of results and other statistics for the current design.

`report_resources`

Displays information about the resource implementation.

`report_timing`

Lists timing information for the current design.

`report_timing_requirements`

Lists timing path requirements and related information.

`report_transitive_fanin`

Lists the fanin logic for selected pins, nets, or ports of the current instance.

`report_transitive_fanout`

Lists the fanout logic for selected pins, nets, or ports of the current instance.

C

Predefined Attributes

This appendix contains tables that list the Design Compiler predefined attributes for each object type.

Table C-1 Clock Attributes

Attribute name	Value
dont_touch_network	{true, false}
fall_delay	float
fix_hold	{true, false}
max_time_borrow	float
minus_uncertainty	float
period	float
plus_uncertainty	float
propagated_clock	{true, false}

Table C-1 Clock Attributes (Continued)

Attribute name	Value
rise_delay	float

Table C-2 Design Attributes

Attribute name	Value
actual_max_net_capacitance	float
actual_min_net_capacitance	float
boundary_optimization	{true, false}
default_flip_flop_type	internally generated string
default_flip_flop_type_exact	library_cell_name
default_latch_type	library_cell_name
design_type	{equation, fsm, pla, netlist}
dont_touch	{true, false}
dont_touch_network	{true, false}
driven_by_logic_one	{true, false}
driven_by_logic_zero	{true, false}
driving_cell_dont_scale	string
driving_cell_fall	string
driving_cell_from_pin_fall	string
driving_cell_from_pin_rise	string
driving_cell_library_fall	string
driving_cell_library_rise	string
driving_cell_multiplier	float

Table C-2 Design Attributes (Continued)

Attribute name	Value
driving_cell_pin_fall	string
driving_cell_pin_rise	string
driving_cell_rise	string
fall_drive	float
fanout_load	float
flatten	{true, false}
flatten_effort	{true, false}
flatten_minimize	{true, false}
flatten_phase	{true, false}
flip_flop_type	internally generated string
flip_flop_type_exact	library_cell_name
is_black_box	{true, false}
is_combinational	{true, false}
is_hierarchical	{true, false}
is_mapped	{true, false}
is_sequential	{true, false}
is_test_circuitry	{true, false}
is_unmapped	{true, false}
latch_type	internally generated string
latch_type_exact	library_cell_name
load	float

Table C-2 Design Attributes (Continued)

Attribute name	Value
local_link_library	design_or_lib_file_name
max_capacitance	float
max_fanout	float
max_time_borrow	float
max_transition	float
min_capacitance	float
minus_uncertainty	float
output_not_used	{true, false}
pad_location (XNF only)	string
part (XNF only)	string
plus_uncertainty	float
port_direction	{in, inout, out, unknown}
port_is_pad	{true, false}
ref_name	reference_name
rise_drive	float
structure	{true, false}
ungroup	{true, false}
wired_logic_disable	{true, false}
xnf_init	string
xnf_loc	string

Table C-3 Library Attributes

Attribute name	Value
default_values	float
k_process_values	float
k_temp_values	float
k_volt_values	float
nom_process	float
nom_temperature	float
nom_voltage	float

Table C-4 Library Cell Attributes

Attribute name	Value
area	float
dont_touch	{true, false}
dont_use	{true, false}
preferred	{true, false}

Table C-5 Net Attributes

Attribute name	Value
ba_net_resistance	float
dont_touch	{true, false}
load	float
subtract_pin_load	{true, false}
wired_and	{true, false}

Table C-5 Net Attributes (Continued)

Attribute name	Value
wired_or	{true, false}

Table C-6 Pin Attributes

Attribute name	Value
disable_timing	{true, false}
max_time_borrow	float
pin_direction	{in, inout, out, unknown}

Table C-7 Reference Attributes

Attribute name	Value
dont_touch	{true, false}
is_black_box	{true, false}
is_combinational	{true, false}
is_hierarchical	{true, false}
is_mapped	{true, false}
is_sequential	{true, false}
is_unmapped	{true, false}
ungroup	{true, false}

Glossary

annotation
A piece of information attached to an object in the design, such as a capacitance value attached to a net; the process of attaching such a piece of information to an object in the design.

back-annotate
To update a circuit design by using extraction and other post-processing information that reflects implementation-dependent characteristics of the design, such as pin selection, component location, or parasitic electrical characteristics. Back-annotation allows a more accurate timing analysis of the final circuit. The data is generated by another tool after layout and passed to the synthesis environment. For example, the design database might be updated with actual interconnect delays; these delays are calculated after placement and routing—after exact interconnect lengths are known.

cell
See instance.

clock
A source of timed pulses with a periodic behavior. A clock synchronizes the propagation of data signals by controlling sequential elements, such as flip-flops and registers, in a digital circuit. You define clocks with the `create_clock` command.

Clocks you create by using the `create_clock` command ignore delay effects of the clock network. Therefore, for accurate timing analysis, you describe the clock network in terms of its latency and skew. See also clock latency and clock skew.

clock gating
The control of a clock signal by logic (other than inverters or buffers), either to shut down the clock signal at selected times or to modify the clock pulse characteristics.

clock latency
The amount of time that a clock signal takes to be propagated from the clock source to a specific point in the design. Clock latency is the sum of source latency and network latency.

Source latency is the propagation time from the actual clock origin to the clock definition point in the design. Network latency is the propagation time from the clock definition point in the design to the clock pin of the first register.

You use the `set_clock_latency` command to specify clock latency.

clock skew
The maximum difference between the arrival of clock signals at registers in one clock domain or between clock domains. Clock skew is also known as clock uncertainty. You use the `set_clock_uncertainty` command to specify the skew characteristics of one or more clock networks.

clock source
The pin or port where the clock waveform is applied to the design. The clock signal reaches the registers in the transitive fanout of all its sources. A clock can have multiple sources.

You use the `create_clock` command with the `source_object` option to specify clock sources.

clock tree
The combinational logic between a clock source and registers in the transitive fanout of that source. Clock trees, also known as clock networks, are synthesized by vendors based on the physical placement data at registers in one clock domain or between clock domains.

clock uncertainty
See clock skew.

core
A predesigned block of logic employed as a building block for ASIC designs.

critical path
The path through a circuit with the longest delay. The speed of a circuit depends on the slowest register-to-register delay. The clock period cannot be shorter than this delay or the signal will not reach the next register in time to be clocked.

datapath
A logic circuit in which data signals are manipulated using arithmetic operators such as adders, multipliers, shifters, and comparators.

current design
The active design (the design being worked on). Most commands are specific to the current design, that is, they operate within the context of the current design. You specify the current design with the `current_design` command.

current instance
The instance in a design hierarchy on which instance-specific commands operate by default. You specify the current instance with the `current_instance` command.

design constraints
The designer's specification of design performance goals, that is, the timing and environmental restrictions under which synthesis is to be performed. Design Compiler uses these constraints—for example, low power, small area, high-speed, or minimal cost—to direct the optimization of a design to meet area and timing goals.

There are two categories of design constraints: design rule constraints and design optimization constraints.

- Design rule constraints are supplied in the technology library. For proper functioning of the fabricated circuit, they must not be violated.
- Design optimization constraints define timing and area optimization goals.

Design Compiler optimizes the synthesis of the design in accordance with both sets of constraints; however, design rule constraints have higher priority.

false path
A path that you do not want Design Compiler to consider during timing analysis. An example of such a path is one between two multiplexed blocks that are never enabled at the same time, that is, a path that cannot propagate a signal.

You use the `set_false_path` command to disable timing-based synthesis on a path-by-path basis. The command removes timing constraints on the specified path.

fanin
The pins driving an endpoint pin, port, or net (also called sink). A pin is considered to be in the fanin of a sink if there is a timing path through combinational logic from the pin to the sink. Fanin tracing starts at the clock pins of registers or valid startpoints. Fanin is also known as transitive fanin.

You use the `report_transitive_fanin` command to report the fanin of a specified sink pin, port, or net.

fanout
The pins driven by a source pin, port, or net. A pin is considered to be in the fanout of a source if there is a timing path through combinational logic from the source to that pin or port. Fanout tracing stops at the data pin of a register or at valid endpoints. Fanout is also known as transitive fanout or timing fanout.

You use the `report_transitive_fanout` command to report the fanout of a specified source pin, port, or net.

fanout load
A unitless value that represents a numerical contribution to the total fanout. Fanout load is not the same as load, which is a capacitance value.

Design Compiler models fanout restrictions by associating a `fanout_load` attribute with each input pin and a `max_fanout` attribute with each output (driving) pin on a cell and ensures that the sum of fanout loads is less than the `max_fanout` value.

flatten
To convert combinational logic paths of the design to a two-level, sum-of-products representation. During flattening, Design Compiler removes all intermediate terms, and therefore all associated logic structure, from a design. Flattening is constraint based.

forward-annotate
To transfer data from the synthesis environment to other tools used later in the design flow. For example, delay and constraints data in Standard Delay Format (SDF) might be transferred from the synthesis environment to guide place and route tools.

generated clock
A clock signal that is generated internally by the integrated circuit itself; a clock that does not come directly from an external source. An example of a generated clock is a divide-by-2 clock generated from the system clock. You define a generated clock with the `create_generated_clock` command.

hold time
The time that a signal on the data pin must remain stable after the active edge of the clock. The hold time creates a minimum delay requirement for paths leading to the data pin of the cell.

You calculate the hold time by using the formula

```
hold = max clock delay - min data delay
```

ideal clock
A clock that is considered to have no delay as it propagates through the clock network. The ideal clock type is the default for Design Compiler. You can override the default behavior (using the `set_clock_latency` and `set_propagated_clock` commands) to obtain nonzero clock network delay and specify information about the clock network delays.

ideal net
Nets that are assigned ideal timing conditions—that is, latency, transition time, and capacitance are assigned a value of zero. Such nets are exempt from timing updates, delay optimization, and design rule fixing. Defining certain high fanout nets that you intend to synthesize separately (such as scan-enable and reset nets) as ideal nets can reduce runtime.

You use the `set_ideal_net` command to specify nets as ideal nets.

input delay
A constraint that specifies the minimum or maximum amount of delay from a clock edge to the arrival of a signal at a specified input port.

You use the `set_input_delay` command to set the input delay on a pin or input port relative to a specified clock signal.

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

leaf cell
A fundamental unit of logic design. A leaf cell cannot be broken into smaller logic units. Examples are NAND gates and inverters.

link library
A technology library that Design Compiler uses to resolve cell references. Link libraries can contain technology libraries and design files. Link libraries also contain the descriptions of cells (library cells as well as subdesigns) in a mapped netlist.

Link libraries include both local link libraries (`local_link_library` attribute) and system link libraries (`link_library` variable).

multicycle path
A path for which data takes more than one clock cycle to propagate from the startpoint to the endpoint.

You use the `set_multicycle_path` command to specify the number of clock cycles Design Compiler should use to determine when data is required at a particular endpoint.

netlist
A file in ASCII or binary format that describes a circuit schematic—the netlist contains a list of circuit elements and interconnections in a design. Netlist transfer is the most common way of moving design information from one design system or tool to another.

operating conditions
The process, voltage, and temperature ranges a design encounters. Design Compiler optimizes your design according to an operating point on the process, voltage, and temperature curves and scales cell and wire delays according to your operating conditions.

By default, operating conditions are specfied in a technology library in an `operating_conditions` group.

optimization
The step in the logic synthesis process in which Design Compiler attempts to implement a combination of technology library cells that best meets the functional, timing, and area requirements of the design.

output delay
A constraint that specifies the minimum or maximum amount of delay from an output port to the sequential element that captures data from the output port. This constraint establishes the times at which signals must be available at the output port to meet the setup and hold requirements of the sequential element.

You use the `set_output_delay` command to set the output delay on a pin or output port relative to a specified clock signal.

pad cell
A special cell at the chip boundaries that allows connection or communication with integrated circuits outside the chip.

path group
A group of related paths, grouped either implicitly by the `create_clock` command or explicitly by the `group_path` command. By default, paths whose endpoints are clocked by the same clock are assigned to the same path group.

pin
A part of a cell that provides for input and output connections. Pins can be bidirectional. The ports of a subdesign are pins within the parent design.

propagated clock
A clock that incurs delay through the clock network. Propagated clocks are used to determine clock latency at register clock pins. Registers clocked by a propagated clock have edge times skewed by the path delay from the clock source to the register clock pin.

You use the `set_propagated_clock` command to specify that clock latency be propagated through the clock network.

real clock
A clock that has a source, meaning its waveform is applied to pins or ports in the design. You create a real clock by using a `create_clock` command and including a source list of ports or pins. Real clocks can be either ideal or propagated.

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

RTL
RTL, or register transfer level, is a register-level description of a digital electronic circuit. In a digital circuit, registers store intermediate information between clock cycles; thus, RTL describes the intermediate information that is stored, where it is stored within

the design, and how it is transferred through the design. RTL models circuit behavior at the level of data flow between a set of registers. This level of abstraction typically contains little timing information, except for references to a set of clock edges and features.

setup time
The time that a signal on the data pin must remain stable before the active edge of the clock. The setup time creates a maximum delay requirement for paths leading to the data pin of a cell.

You calculate the setup time by using the formula

```
setup = max data delay - min clock delay
```

slack
A value that represents the difference between the actual arrival time and the required arrival time of data at the path endpoint in a mapped design. Slack values can be positive, negative, or zero.

A positive slack value represents the amount by which the delay of a path can be increased without violating any timing constraints. A negative slack value represents the amount by which the delay of a path must be reduced to meet its timing constraints.

structuring
To add intermediate variables and logic structure to a design, which can result in reduced design area. Structuring is constraint based. It is best applied to noncritical timing paths.

By default, Design Compiler structures your design.

synthesis
A software process that generates an optimized gate-level netlist, which is based on a 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. Design Compiler uses symbol libraries to generate the design schematic. You can use Design Vision to view the design schematic.

target library
The technology library to which Design Compiler maps during optimization. Target libraries contain the cells used to generate the netlist and definitions for the design's operating conditions.

technology library
A library of ASIC cells that are available to Design Compiler during the synthesis process. A technology library can contain area, timing, power, and functional information on each ASIC cell. The technology of each library is specific to a particular ASIC vendor.

timing exception
An exception to the default (single-cycle) timing behavior assumed by Design Compiler. For Design Compiler to analyze a circuit correctly, you must specify each timing path in the design that does not conform to the default behavior. Examples of timing exceptions include false paths, multicycle paths, and paths that require a specific minimum or maximum delay time different from the default calculated time.

timing path
A point-to-point sequence that dictates data propagation through a design. Data is launched by a clock edge at a startpoint, propagated through combinational logic elements, and captured at an endpoint by another clock edge. The startpoint of a timing path is an input port or clock pin of a sequential element. The endpoint of a timing path is an output port or a data pin of a sequential element.

transition delay
A timing delay caused by the time it takes the driving pin to change voltage state.

ungroup
To remove hierarchy levels in a design. Ungrouping merges subdesigns of a given level of the hierarchy into the parent cell or design.

You use the `ungroup` command or the `compile` command with the `auto_ungroup` option to ungroup designs.

uniquify
To resolve multiple cell references to the same design in memory.

The uniquify process creates unique design copies with unique design names for each instantiated cell that references the original design.

virtual clock
A clock that exists in the system but is not part of the block. A virtual clock does not clock any sequential devices within the current design and is not associated with a pin or port. You use a virtual clock as a reference for specifying input and output delays relative to a clock outside the block.

You use the `create_clock` command without a list of associated pins or ports to create a virtual clock.

wire load model
An estimate of a net's RC parasitics based on the net's fanout, in the absence of placement and routing information. The estimated capacitance and resistance are used to calculate the delay of nets. After placement and routing, you should back-annotate the design with detailed information on the net delay.

The wire load model is shipped with the technology library; vendors develop the wire load model based on statistical information specific to the vendor's process. You can also custom-generate the model based on back-annotation. The model includes coefficients for area,

capacitance, and resistance per unit length, and a fanout-to-length table for estimating net lengths (the number of fanouts determines a nominal length).

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X