Design Compiler
Reference Manual: Optimization and Timing Analysis

Version 1999.10, October 1999

SYNOPSYS®

Printed in the U.S.A.

Document Order Number: 31735-000 HA
Design Compiler Reference Manual: Optimization and Timing Analysis, v1999.10

Table of Contents

List of Figures

List of Tables

List of Examples

About This Manual

The *Design Compiler Reference Manual: Optimization and Timing Analysis* describes concepts and commands used for optimizing designs and performing timing analysis using Synopsys Design Compiler tools.

This preface includes the following sections:

- Audience
- Related Publications
- SOLV-IT! Online Help
- Customer Support
- Conventions

Audience

This manual is for logic designers and engineers who use the Design Compiler tools to design ASICs, ICs, and FPGAs, and who are familiar with high-level design techniques, hardware description language (such as VHDL or Verilog-HDL), and UNIX.

If you plan to use the Synopsys VHDL System Simulator (VSS) with the Synopsys synthesis software, you need a basic knowledge of VSS.

Related Publications

For additional information about Design Compiler, see Synopsys Online Documentation (SOLD), which is included with the software, or Documentation on the Web, which is available through SolvNET on the Synopsys Web page at

`http://www.synopsys.com`

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

- Floorplan Manager
- FPGA Compiler
- HDL Compiler for Verilog
- VHDL Compiler

SOLV-IT! Online Help

SOLV-IT! is the Synopsys electronic knowledge base. It contains information about Synopsys and its tools and is updated daily. For more information about SOLV-IT!, send e-mail to

`solvitfb@synopsys.com`

or go to the Synopsys Web page at

`http://www.synopsys.com`

and click SolvNET.

Customer Support

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

- Send e-mail to

  `support_center@synopsys.com`

- Call (800) 245-8005 from within the continental United States or call (650) 584-4200 from outside the continental United States, from 7:00 a.m. to 5:30 p.m. Pacific time, Monday through Friday.

- Send a fax to (650) 584-2539.

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
`Courier`	Indicates command syntax. In command syntax and examples, shows system prompts, text from files, error messages, and reports printed by the system.
italic	Indicates a user specification, such as *`object_name`*
bold	In interactive dialogs, indicates user input (text you type).
[ ]	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.

1

Basic Concepts for Optimizing Designs

Optimizing (compiling) is the step in the synthesis process that attempts to implement a combination of library cells that meets the functional, speed, and area requirements of your design. Optimization transforms a design into a technology-specific circuit based on the attributes and constraints you place on the design.

The quality of optimization results depends on how the HDL description is written. In particular, the partitioning of the hierarchy in the HDL, if done well, can enhance optimization.

Before you optimize, you need to know about the topics explained in the following sections:

- Using DC Ultra
- Exploring the Design Space
- Optimization Phases

- Full Compilation
- Test-Ready Compilation
- Incremental Compilation
- Critical Path Resynthesis
- Logic Duplication and Mapping to Wide-Fanin Gates
- Optimizing Once for Best- and Worst-Case Conditions
- Optimizing With Multiple Libraries
- Optimization Flow

Before you optimize, see *Design Compiler User Guide* and *Design Compiler Reference Manual: Constraints and Timing*.

During optimization, the Design Compiler tool from Synopsys attempts to meet the constraints you have set on the design. The constraints cost vector, described in Chapter 1 of the Design Compiler Reference Manual: Constraints and Timing, guides optimization. By default, the design rule constraints (transition, fanout, capacitance, and cell degradation) have a higher priority than the optimization constraints (delay and area).

Using DC Ultra

Unless otherwise noted, this manual describes the optimization capabilities and flow you get with DC Expert. However, the Design Compiler high-end configuration, DC Ultra, provides additional algorithms for performing aggressive logic duplication and mapping to wide-fanin gates. See the *Design Compiler User Guide* for more details about DC Ultra.

The DC Ultra configuration requires a DC Ultra license. Features that require the license are noted throughout the reference manual set.

With the DC Ultra license, you can invoke the logic duplication and wide-fanin gate-mapping algorithms as part of a high-effort compilation, using the `set_ultra_optimization` command.

When you enter the `set_ultra_optimization` command, Design Compiler checks to ensure that a DC Ultra license is available. If a DC Ultra license is not available, the tool issues a warning and uses a regular Design Compiler license instead. If you use the `-force` option with the command and a DC Ultra license is not available, the return status is 0 (it is 1 otherwise) and compilation stops.

The command syntax is

```
set_ultra_optimization [-force]
```

DC Ultra is equipped with two types of optimization engines. The default DC Ultra engine is described previously. The second optimization engine can produce significantly better delay and area QoR for libraries with large number of consistently sized gates. See the DC-Ultra Library Guidelines application note for additional details.

The second optimization engine is enabled by setting the variable `compile_new_optmization` to true. You must also invoke DC Ultra by entering the `set_ultra_optimization` command.

When the new optimization engine is enabled, DC Ultra performs a library analysis step to determine whether these new algorithms will work well with the specified library. If the library is not suitable for the second optimization engine algorithms, Design Compiler reverts back to the default optimization engine. These optimization steps work with both medium- and high-effort compiles.

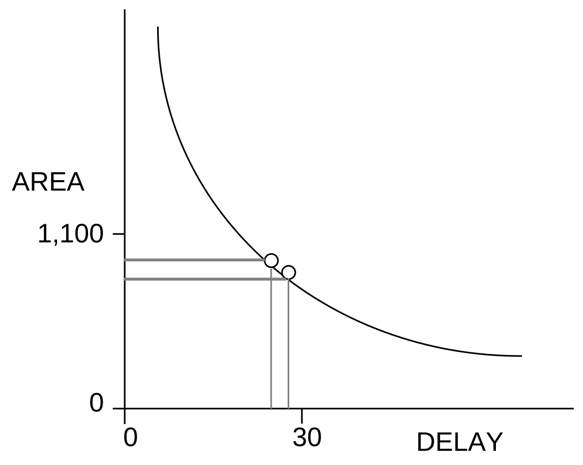
Exploring the Design Space

Experimenting with speed and area to get the smallest or fastest design is called exploring the design space.

Using Design Compiler, you can examine different implementations of the same design in a relatively short time.

Figure 1-1 shows a design space curve. The shape of the curve demonstrates the tradeoff between area-efficient and speed-efficient circuits.

Figure 1-1 Design Space Curve



Optimization Phases

The optimization process modifies the logic in a netlist. Optimization uses cells from the technology library in an attempt to meet specified constraints.

Using various commands and options, you can direct Design Compiler to perform all or some combination of the following major phases as it optimizes your design:

- Combinational optimization, including
  - Technology-independent
  - Technology-specific (usually called “mapping”)
- Sequential optimization: initial and final
- I/O pad optimization
- Local optimizations

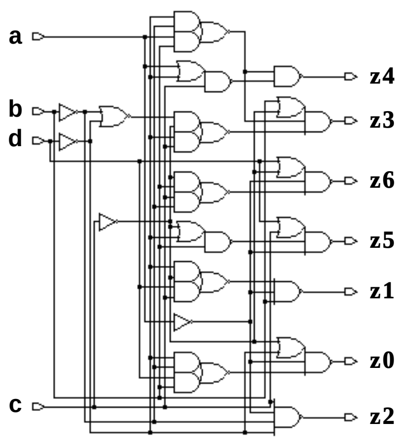
Combinational Optimization

The combinational optimization phase transforms the logic-level description of the combinational logic to a gate-level netlist.

Figure 1-2 shows the logic-level description of the combinational logic and the gate-level optimization for design LED.

Figure 1-2   Logic-Level and Gate-Level Optimization for Design LED

```
design_name LED
 .inputnames a b c d
 .outputnames z0 z1 z2 z3 z4 z5 z6
 n1 = c' ;
 n2 = d' ;
 n3 = a' ;
 n4 = ((n2' + n6') * (d' + b')) ;
 n5 = ((n2' + n1') * (d' + c')) ;
 n6 = b' ;
 n7 = ((n2' + n6') * (a' + b')) ;
 n8 = ((n12' + n1') * (n2' + c')) ;
 n9 = ((a' * n2') + c') ;
 n10 = ((n1' * n2') + b') ;
 n11 = ((n1' + b') * (c' + n6')) ;
 n12 = (n6' * n2') ;
 z0 = ((n1' * n2') + n3' + n4') ;
 z1 = (n5' + n3' + b') ;
 z2 = (c' + n3' + n6' + n2') ;
 z3 = ((b' * n1') + n7' + n8') ;
 z4 = (n7' + n9') ;
 z5 = ((d' * c') + n10' + n3') ;
 z6 = ((d' * n1') + n3' + n11') ;
```



Total Area = 32

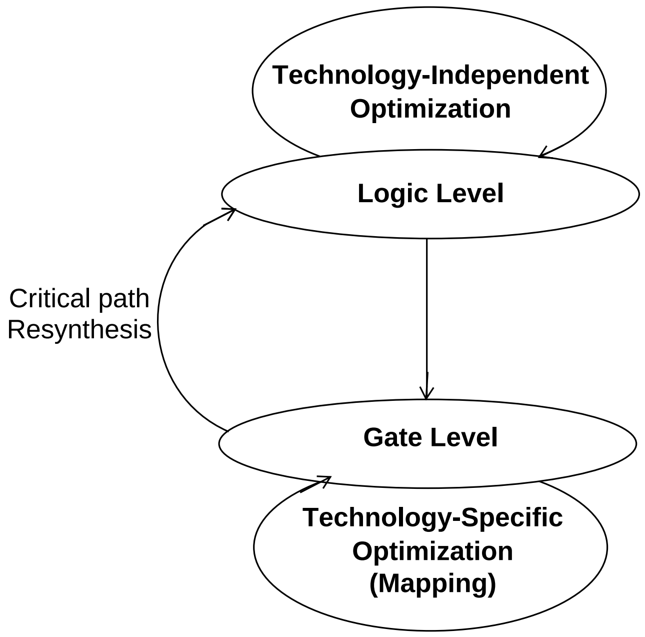
The two major phases in combinational optimization are

- Technology-independent optimization, which operates at the logic level. Design Compiler represents the gates as a set of Boolean logic equations.
- Technology-specific optimization (mapping), which operates at the gate level.

Both the logic and the gate structures are important to the overall quality of the design.

Figure 1-3 shows the two levels of design representation and the related combinational optimization phases.

Figure 1-3   Combinational Optimization Phases


Technology-Independent Optimization
Logic Level
Critical path Resynthesis
Gate Level
Technology-Specific Optimization (Mapping)

Technology-Independent Optimization

Technology-independent optimization applies algebraic and Boolean techniques to a set of logic equations. This optimization reimplements the logic equations to meet your timing and area goals.

The most common technology-independent optimization techniques are flattening and structuring.

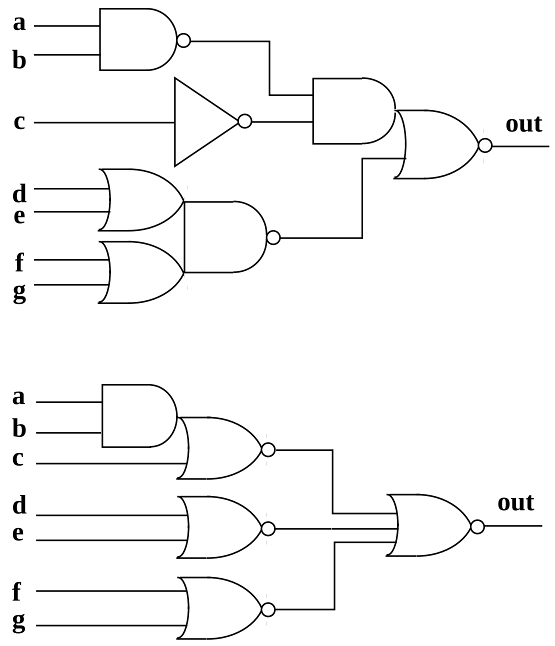
Flattening and structuring are described in detail in Chapter 4, “Controlling Logic-Level and Gate-Level Optimization.”

Technology-Specific Optimization (Mapping)

Technology-specific optimization (usually called mapping) synthesizes a gate-level design that attempts to meet your timing and area constraints.

During mapping, Design Compiler selects components from the technology library to implement the logic structure. Design Compiler tries different logic combinations, using only components that approach the defined speed and area goals. Figure 1-4 shows examples of mapped gates.

Figure 1-4 Mapped Gates



Mapping comprises several phases. These phases are described in “Mapping Optimization” on page 1-26.

Initial Sequential Optimization

Initial sequential optimization maps sequential cells to cells in the library. You can map to either standard sequential cells or scan-equivalent cells. Initial sequential optimization is in the first phase of gate-level optimization.

At this point in the optimization process, information about the delay through the combinational logic is incomplete. Design Compiler does not have enough information to select the optimum sequential cell. The tool can correct this lack of information later, in the final sequential optimization phase.

Design Compiler optimizes the sequential cells, defining the following information:

- Locations of the islands of combinational logic between sequential cells
- Timing constraints on the combinational islands required to meet the setup and hold constraints on the sequential cells

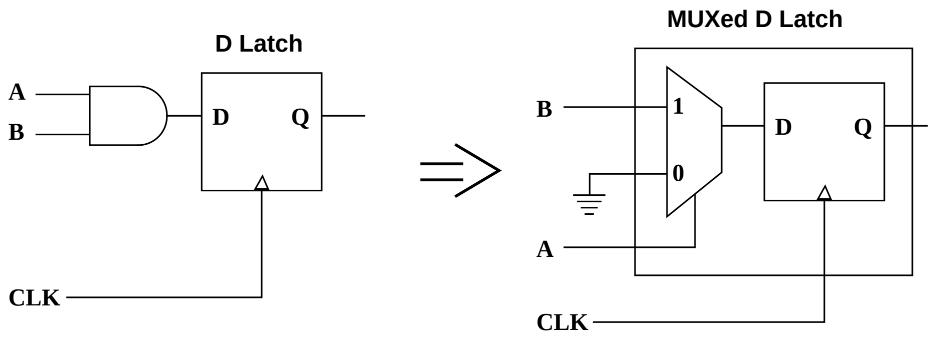
Final Sequential Optimization

Design Compiler has accurate values for all delays through the I/O pads and combinational logic before it enters the final sequential optimization phase. (Final sequential optimization is part of the "Mapping Optimization" phase, described later in this chapter.) In this phase, Design Compiler optimizes timing-critical sequential cells (cells on the critical path). The tool examines each sequential cell and its surrounding combinational logic to determine if they might be replaced by more-complex sequential cells from the target library.

Final sequential optimization can

- Improve design timing by choosing higher-performance sequential cells.
- Possibly reduce the design's area and delay if complex sequential cells exist in the library. The tool incorporates the combinational logic in the sequential cell, as shown in Figure 1-5.
- Improve area by using the variable `compile_sequential_area_recovery`. When this variable is set to true, compile will try to re-map the sequential elements to recover area.

Figure 1-5 Sequential Optimization



I/O Pad Optimization

Design Compiler maps the combinational logic and does an initial sequential logic mapping before it inserts and maps the I/O pads.

Input and output buffers are added to each port in the top-level design. The buffers are sized to meet the port-to-port timing constraints when the delays through the core logic are known.

I/O pad synthesis is described in detail in Chapter 7, "Adding I/O Pad Cells."

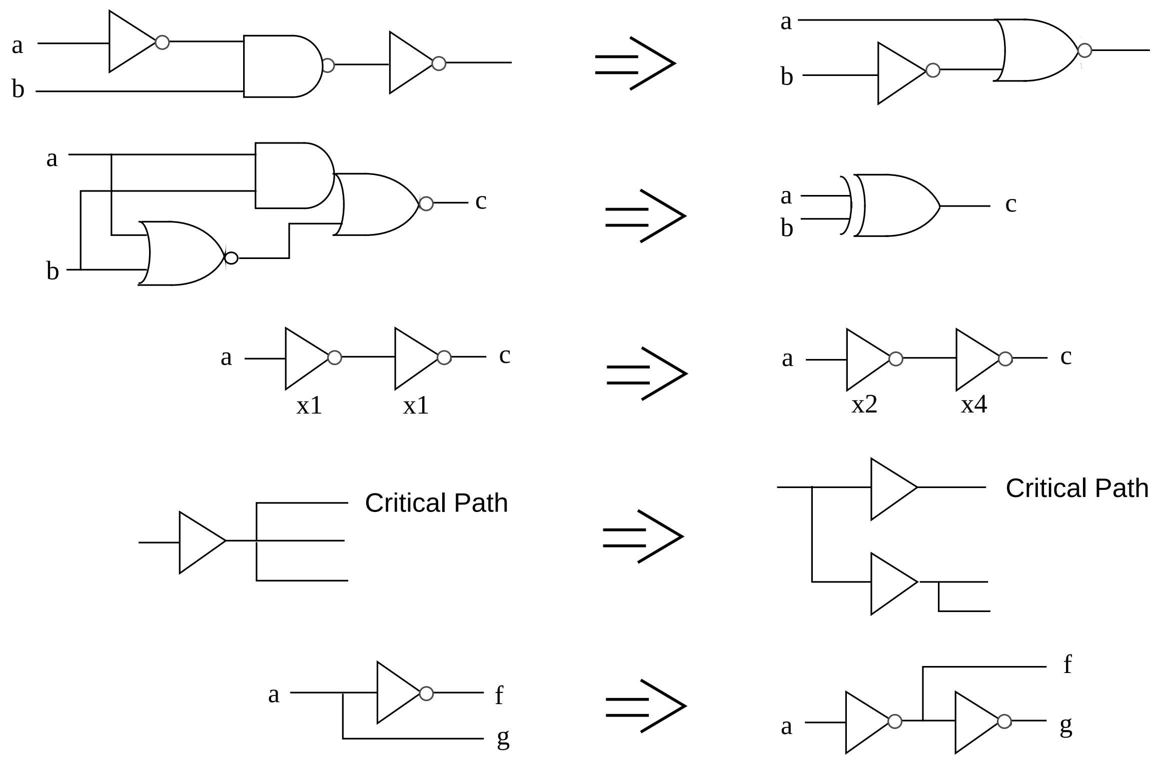
I/O buffers consume significant current; therefore, gate-level optimization selects the smallest I/O buffer that meets the timing specification of the design. You can construct I/O pad logic more complex than simple buffers.

Local Optimizations

The final step in gate-level optimization involves making local changes. Design Compiler makes incremental modifications to the design to adjust for timing or area.

Figure 1-6 shows common local optimization steps.

Figure 1-6 Local Optimization Steps



Full Compilation

The full compilation pass is used to optimize an RTL design. Depending on the compile options and attributes you set, it can include flattening, structuring, and mapping. During the full compilation process, Design Compiler removes the existing gate structure from a design, then rebuilds the design.

A full compilation performs both technology-independent optimization (including flattening, if enabled, and structuring) as well as technology-specific optimization (mapping). This enables more design improvements than an incremental compilation, which focuses only on the portions of the design that do not meet constraints.

The `compile` command invokes the full compilation process.

Test-Ready Compilation

Test-ready compilation integrates logic optimization and scan insertion by mapping all sequential cells directly to scan cells. The optimization cost functions consider the impact of the scan cells themselves and the additional loading due to the scan-chain routing. By accounting for the timing impact of scan design from the start of the synthesis process, test-ready compilation eliminates the need to recompile your design after scan insertion.

The `compile` command, with its `-scan` option, enables test-ready compilation. Test-ready compilation requires a Test-Compiler license (included as part of DC Expert *Plus*).

Incremental Compilation

An incremental compilation improves the existing design cost by focusing on the areas of the design that do not meet constraints. The existing structure is preserved if all constraints are already met.

Incremental mapping uses the existing gates from an earlier compilation as a starting point for the mapping process. Mapping optimizations are accepted only if they improve the circuit speed or area. Incremental mapping guarantees that a circuit can only be improved.

Unlike a full compilation, the design does not go through an initial flattening or structuring phase in an incremental compilation. In addition, portions of the design are restructured if that improves the design costs with high effort compile.

The `compile` command, with its `-incremental_mapping`, option invokes the incremental compilation process.

Top Level Compilation

Design Compiler's top level optimization capability fixes constraint violations occurring at the top level after the subblocks in a design are assembled. These violations might occur due to changes in the environment around the subblocks as a result of the optimizations that have been performed in the subblocks.

Top level optimization fixes only violations of top level nets because it is assumed that the subblocks have been compiled separately and are meeting timing. However, any design rule violations present in the design will be fixed regardless of where the violation occurs.

The `compile` command with its `-top` option invokes the top level optimization process. This option works with mapped designs only and runs significantly faster than an incremental compilation because of its emphasis on top level nets. Additionally, it does not perform any incremental implementation selection of synthetic components or any area recovery on the design.

Critical Path Resynthesis

Using critical path resynthesis on your design improves timing. It identifies the critical path and attempts to do a full compilation on only the logic along that path. This process then repeats on the new critical path.

The `compile` command, with its `-map_effort high`, option enables critical path resynthesis.

Note:

> Critical path resynthesis requires a DC Expert license. The DC Ultra version of the `compile` command with its `-map_effort high` option enables mapping to wide-fanin gates. See the next section, “Logic Duplication and Mapping to Wide-Fanin Gates.”

Logic Duplication and Mapping to Wide-Fanin Gates

The DC Ultra version of Design Compiler includes algorithms that enable mapping to wide-fanin gates, plus extensive logic duplication steps on the critical path. Use the `-map_effort high` option of the `compile` command to invoke these algorithms.

During compilation, Design Compiler evaluates the cells along critical paths. The tool determines whether it can improve the timing or area of the paths by replacing groups of cells with complex, wide-fanin cells from the technology library. Additionally, Design Compiler tries to improve the timing of high-fanout or heavily loaded nets, by duplicating and restructuring large sections of the logic driving the nets. This duplication and restructuring to gain timing improvements often results in significant increases in the design area.

Optimizing Once for Best- and Worst-Case Conditions

Using Design Compiler, you can constrain your design *once* for both minimum (best-case) and maximum (worst-case) optimization and timing analysis. The tool then optimizes and analyzes the timing in a single compile run.

You can constrain the design by using either a single technology library or multiple libraries. Whether you use one or multiple libraries, the general methodology is as follows:

1. Set up a technology library file or files. Make sure the files contain
    - Best- and worst-case operating conditions
    - Optimistic and pessimistic wire load models

  - Minimum and maximum timing delays

Note:

> If you use multiple libraries, see the next section, “Optimizing With Multiple Libraries.”

2. Specify minimum and maximum constraints.
   - Environmental—including operating conditions and wire loads
   - Clock information—including clock skew and clock transition
   - Optimization—including input and output delays, drive, load, and resistance
   - Design rule—including transition time, fanout, and capacitance
3. Optimize the design (run `compile`) for simultaneous minimum and maximum timing. To ensure that minimum delay constraints are optimized with respect to a particular clock, specify the `fix_hold` attribute for that clock, using the `set_fix_hold` command.
4. Back-annotate the Standard Delay Format (SDF) and RC values.

Note:

> At this point, run `reoptimize_design` (and options) to get simultaneous -min/-max optimization, if you have a DC Ultra license or a Floorplan Manager license.

5. Report and analyze the paths showing constraint violations.

The following constraint-related, reporting, and back-annotation commands support both minimum and maximum optimization and timing analysis:

Constraint-related commands

```
set_min_library
set_operating_conditions
set_wire_load_model
set_wire_load_mode
set_wire_load_min_block_size
set_wire_load_selection_group
set_clock_skew
set_clock_transition
set_drive
set_load
set_port_fanout_number
set_resistance
```

Reporting commands

```
report_annotated_delay
report_area
report_attribute
report_bus
report_cache
report_cell
report_clock
report_clusters
report_compile_options
report_constraint
report_delay_calculation
report_design
report_design_lib
report_fpga
report_fsm
report_hierarchy
report_internal_loads
```

```
report_lib
report_name_rules
report_net
report_path_group
report_port
report_power
report_reference
report_resources
report_routability
report_synlib
report_test
report_timing
report_timing_requirements
report_transitive_fanin
report_transitive_fanout
report_wire_load
report_xref
```

Back-annotation commands

```
read_sdf
set_annotated_delay
set_annotated_check
```

Other commands, including `report_design`, `report_port`, and `report_wire_load`, generate reports on the minimum and maximum constraints on your design. You do not need to specify the `-min` option.

The `set_min_library` command is described in the next section, “Optimizing With Multiple Libraries.” The other commands are described elsewhere in this manual or in the *Design Compiler Reference Manual: Constraints and Timing*. When you are not sure which manual describes a particular command related to Design Compiler, see the “commands” listing in the *Design Compiler Reference Manual: Master Index*.

Optimizing With Multiple Libraries

The `set_min_library` command directs Design Compiler to use multiple technology libraries for minimum- and maximum-delay analyses in one optimization run. Thus, you can choose libraries that contain all of the following:

- Best- and worst-case operating conditions
- Optimistic and pessimistic wire load models
- Minimum and maximum timing delays

You can direct Design Compiler to analyze them simultaneously. To accomplish this analysis, use the `set_min_library` commands to create a link between the data in the two libraries.

The `set_min_library` command creates a minimum/maximum relationship between two library files. You specify a *max_library* to be used for maximum delay analysis and a *min_library* to be used for minimum delay analysis. Only *max_library* should be used for linking and as target library. When Design Compiler needs to compute a minimum delay value, it first analyzes the library cell in the *max_library*, then looks to the *min_library* to determine if a match exists. If a library cell with the same name, the same pins, and the same timing arcs exists in the *min_library*, Design Compiler uses the timing information from the *min_library*. If the tool cannot find a matching cell in the *min_library*, it uses the cell in the *max_library*.

The syntax is

```
set_min_library max_library\
     -min_version min_library | -none
```

max_library

Names the library you want Design Compiler to use as a link _library or target_library for maximum delay analysis.

`-min_version` *min_library*

Names the library you want Design Compiler to use for minimum delay analysis. Do *not* specify the *min_library* as the link_library or target_library.

`-none`

Undoes a *min_library* setting.

Example

This example shows how you might use `set_min_library` with `set_operating_conditions` to control and report delay analysis. If you do not specify a minimum, Design Compiler uses the maximum condition for both minimum and maximum delay analysis. You cannot use the `-min` option without also using the `-max` option.

```
dc_shell> link_library = target_library = "LIB_WC_COM.db"            (1)
dc_shell> set_min_library LIB_WC_COM.db -min_version LIB_BC_COM.db    (2)
dc_shell> set_operating_conditions -max WC_COM -min BC_COM            (3)
dc_shell> include minmax.cons
dc_shell> set_fix_hold clk
dc_shell> compile
dc_shell> report_timing -delay max
dc_shell> report_timing -delay min
```

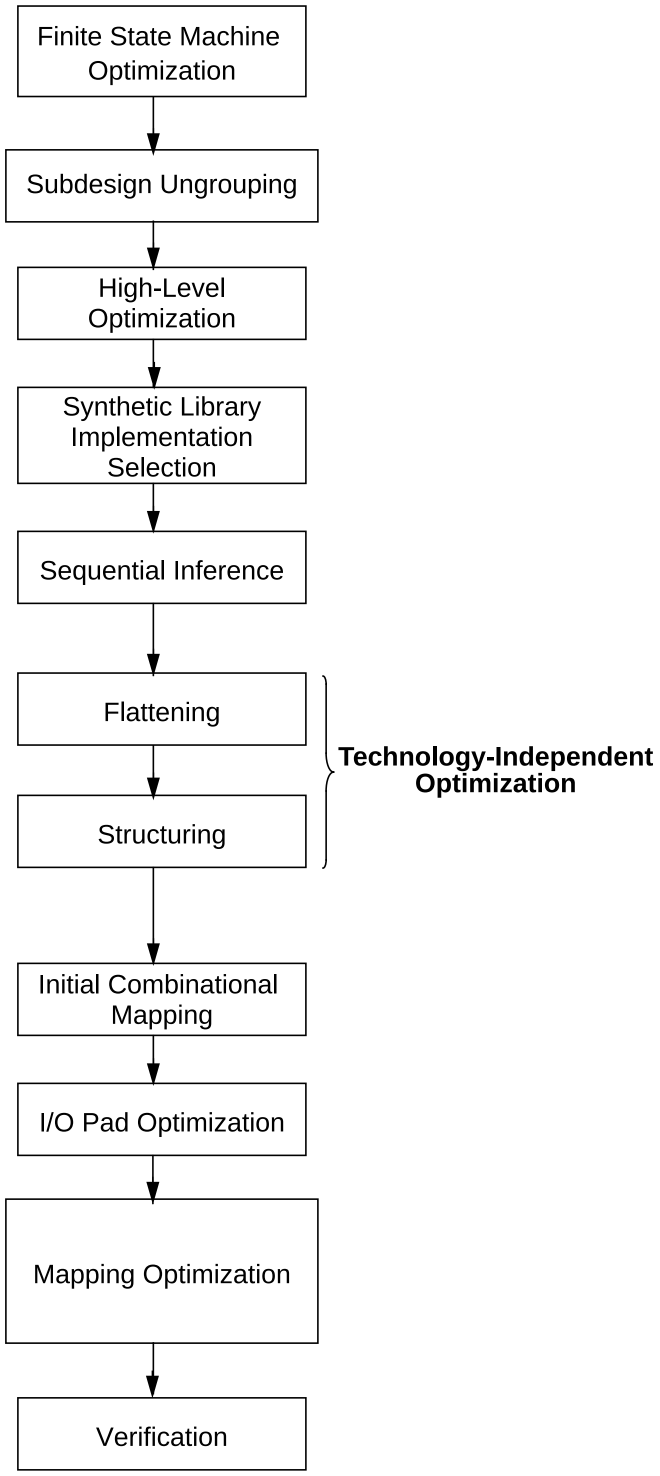
Note:

1. Use the *max_library* (only) as the link and target library.
2. Use the library file name (not the library name) with the `set_min_library` command.
3. Use both the maximum and minimum options with the `set_operating_conditions` command.

Optimization Flow

Figure 1-7 shows the (full compile) optimization flow for one level of hierarchy. The sections following the figure describe the steps in the flow.

Figure 1-7   Full Compile Optimization Flow


Finite State Machine Optimization
Subdesign Ungrouping
High-Level Optimization
Synthetic Library Implementation Selection
Sequential Inference
Flattening
Structuring
Technology-Independent Optimization
Initial Combinational Mapping
I/O Pad Optimization
Mapping Optimization
Verification

Finite State Machine Optimization

Design Compiler processes each finite state machine (FSM) in the design. A subdesign is represented as an FSM if it is read directly from a state table format or if it is extracted with the `extract` command. Design Compiler optimizes the state encoding of each FSM, then converts it to Boolean equations and technology-independent flip-flops.

A group of commands controls FSM optimization. See Chapter 10, “Optimizing Finite State Machines.”

Subdesign Ungrouping

By default, subdesigns are compiled hierarchically, preserving their design boundaries. If you use the `set_ungroup` command with any subdesigns, those subdesigns are ungrouped into their parent designs before being compiled. See the *Design Compiler User Guide*.

High-Level Optimization

During high-level optimization, resources are allocated and shared, depending on timing and area considerations. Additional optimizations, such as arithmetic optimization and the sharing of common subexpressions, are also performed. See the *HDL Compiler for Verilog Reference Manual* and the *VHDL Compiler Reference Manual* for more information.

Synthetic Library Implementation Selection

In this step, synthetic library modules are mapped to architectural representations (implementations). The implementation used depends on your constraints. During an incremental compile, Design Compiler reevaluates the existing implementation to determine if it meets the constraints. If the constraints are not met, the tool selects a different implementation.

Modules simplify the instantiation process. You can define designs on a general level, in terms of functions, and then let Design Compiler automatically select the appropriate implementation to meet your timing and area constraints. See the DesignWare databooks for more information.

Sequential Inference

You can write an HDL description to pass hints to the optimizer, indicating which sequential elements are preferred, on an instance-by-instance basis. The hints are taken into consideration before technology-independent optimization, and the corresponding sequential elements (or their scan equivalents) are instantiated then. This methodology makes it easier to control the kind of sequential elements inferred during compilation.

Flattening

Flattening is an optional optimization step that removes intermediate variables and, therefore, logic structure from a design. A flattened design can be fast, because it consists of just two levels of combinational logic. Flattening is not always practical, in that it is expensive in terms of CPU and memory. See Chapter 4, “Controlling Logic-Level and Gate-Level Optimization.”

Structuring

Structuring is an optional optimization step that adds intermediate variables and logic structure to a design. During structuring, Design Compiler searches for subfunctions that can be factored out, then evaluates these factors based on the size of the factor and the number of times the factor appears in the design. The subfunctions that most reduce the logic are turned into intermediate variables and factored out of the design equations. See Chapter 4, “Controlling Logic-Level and Gate-Level Optimization.”

Initial Combinational Mapping

Mapping is the optimization process that attempts to create a gate-level implementation that meets all constraints. Design Compiler compares each of its circuit implementations against your constraints to gauge the overall quality of that implementation. Your constraints determine which transformations are accepted.

I/O Pad Insertion and Optimization

I/O pad insertion adds I/O buffers to primary inputs and outputs. Instantiated I/O pads have the characteristics you define, such as voltage levels, current levels, pull-up and pull-down resistors, slew-rate control, and so forth. I/O pad optimization potentially modifies previously inserted I/O pads to meet constraints. I/O pads can be optimized to include sizing and incorporation of core logic functionality in the pads. See Chapter 7, “Adding I/O Pad Cells.”

Mapping Optimization

In this step, Design Compiler completes the gate-level implementation it started during initial mapping. The mapping optimization process comprises four phases:

- Delay optimization
- Design rule fixing (two parts)
- Area optimization

The following sections describe each of these phases.

Phase 1: Delay Optimization

In this phase, Design Compiler optimizes the delay performance of critical paths in your design.

Design Compiler takes design rules into account during this phase. When two circuit solutions offer the same delay performance, Design Compiler implements the solution that has the lower design rule cost.

Phase 2: Design Rule Fixing, Part 1

In this phase, Design Compiler fixes whatever design rule violations it can—without increasing delay cost. (Sometimes Design Compiler can find multistep ways to fix design rule violations without increasing delay cost. These multistep solutions are not possible if delay cost is not considered in this phase.)

Whenever possible, Design Compiler fixes design rule violations by resizing gates across multiple logic levels—as opposed to adding buffers to the circuitry.

Phase 3: Design Rule Fixing, Part 2

This phase is on by default. If any design rule violations remain after part 1 of design rule fixing, Design Compiler fixes them here, even if doing so *increases* the delay cost. (By default, design rules carry a higher priority than delay. See the constraints default cost vector information in the *Design Compiler Reference Manual: Constraints and Timing*.)

If you want to skip this phase, use the `set_cost_priority` command to prioritize delay ahead of design rules. The command is described in the *Design Compiler Reference Manual: Constraints and Timing*.

Phase 4: Area Optimization

Assuming that you have placed area constraints on your design, Design Compiler now attempts to minimize the number of gates in the design. Using the `-map_effort` or `-area_effort` option of the `compile` command, you can direct Design Compiler to put a low, medium, or high effort into area optimization. (If you do not place area constraints on your design, Design Compiler performs a limited series of downsizing and area cleanup steps.)

- Low effort
  Design Compiler does gate sizing and buffer and inverter cleanup. Design Compiler allocates limited CPU time to this effort level.

- Medium effort
  Design Compiler adds phase assignment to gate sizing and buffer and inverter cleanup. Design Compiler allocates more CPU time to this effort than to a low-effort optimization.
- High effort
  Design Compiler tries still more gate minimization strategies.The tool adds gate composition to the process and allocates even more CPU time.

Note:

> Whichever area optimization effort level you choose, the overall constraints cost vector (described in the *Design Compiler Reference Manual: Constraints and Timing*) prevails. Even during area optimization, if Design Compiler finds a new opportunity to improve delay cost, it makes the change—even if it increases area cost. Area always has a lower priority than delay.

Verifying Design Functionality

Logic verification determines whether two designs are functionally equivalent. Verification ensures that the synthesis process or manual design changes do not introduce errors. Timing considerations are ignored. You can use logic verification to determine whether a process, such as compilation, preserves the functionality of the original design. Logic verification is also useful for checking manual implementations or changes against the HDL specification. See Chapter 8, “Verifying Designs for Functionality.”

2

Understanding Compilation Strategies

Different pieces of your design require different compilation strategies. You need to develop a compilation strategy before you compile. You can use various strategies to compile, depending on your design, and you can mix strategies.

This chapter contains the following sections:

Choosing the Strategy

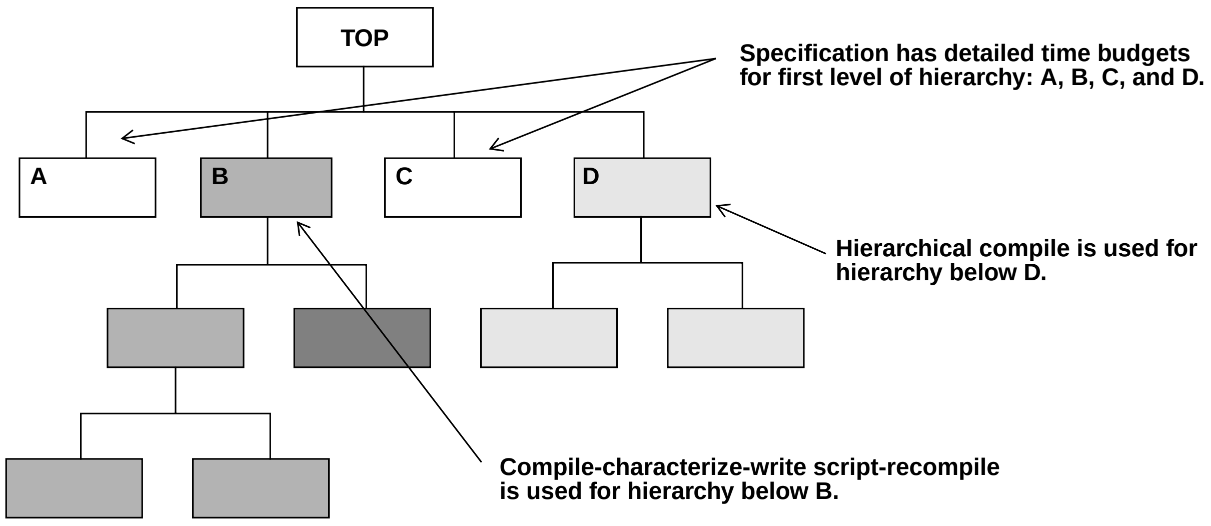
The two strategies for compiling a hierarchical design are

- Top-down hierarchical compile
- compile-characterize-write script-recompile

Mixing Compilation Strategies

You can mix the two compilation strategies, as shown in Figure 2-1.

Figure 2-1 Mixing Compilation Strategies


TOP
Specification has detailed time budgets for first level of hierarchy: A, B, C, and D.
A
B
C
D
Hierarchical compile is used for hierarchy below D.
Compile-characterize-write script-recompile is used for hierarchy below B.

Using the Top-Down Hierarchical Compile Strategy

Design Compiler automatically compiles hierarchical circuits without collapsing the hierarchy. After each module in the design is compiled, Design Compiler continues to optimize the circuit until the constraints are met. This process sometimes requires recompiling subdesigns on a critical path. When the performance goals are achieved or when no further improvement can be made, the compile process stops.

If multiple instances of the same design occur, the `uniquify`, `set_dont_touch`, or `ungroup` command needs to be run, so that each instance can be handled appropriately during optimization.

Hierarchical compilation is automatic when the design being compiled has multiple levels of hierarchy that are not marked `dont_touch`. Design Compiler preserves the hierarchy information and optimizes individual levels automatically, based on the constraints at the top level of hierarchy (`dont_touch` attributes placed on the top level of hierarchy are ignored).

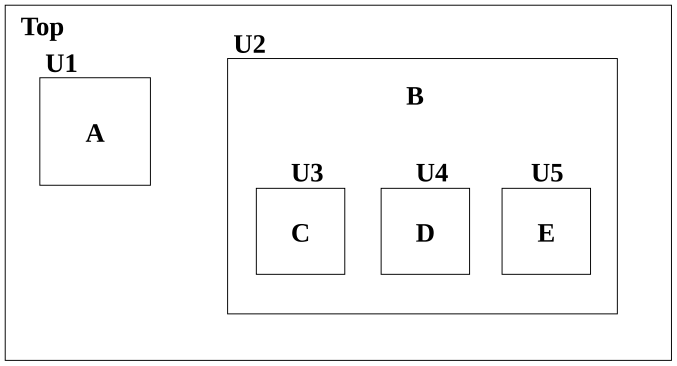
The top-down hierarchical compile strategy is an easy, push-button approach. Intermodule dependencies are taken care of automatically.

The top-down hierarchical compile strategy requires these steps:

1. Read in the entire design.
2. Resolve multiple references, using the `set_dont_touch`, `ungroup`, or `uniquify` command. Apply constraints and attributes to the top level. Constraints and attributes are based on the design specification.
3. Compile.

Example

Use top-down hierarchical compile for design Top.



The design specification for design Top is

Operating conditions:	WCCOM
Wire Load Model:	"20x20"
clock:	22 MHz
Input delay time:	10 ns

Use this compile script to run top-down hierarchical compile for design Top. This script includes a script of defaults, default.scr.

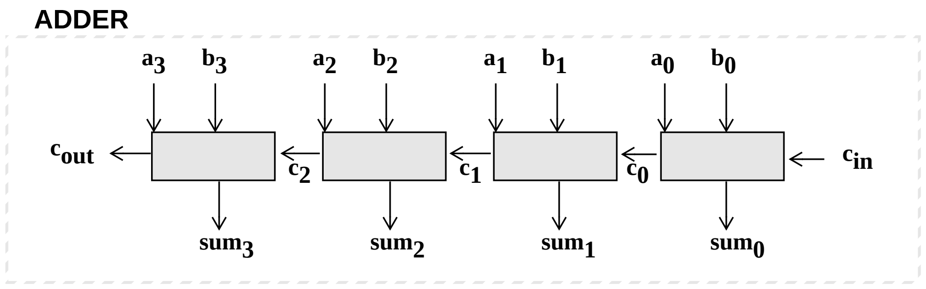
```
read -f vhdl top.vhd
current_design Top
uniquify
include default.scr
set_operating_conditions WCCOM
set_wire_load_model -name “20x20”
create_clock -period 45 clk
set_input_delay 10 -clock clk all_inputs()
set_output_delay
compile
```

Running a Top-Down Hierarchical Compile Strategy

This example of running the hierarchical optimization strategy uses a hierarchical, 4-bit adder design named ADDER. ADDER is composed of four 1-bit full adders with A, B, and CIN (carry in) inputs and SUM and COUT (carry out) outputs.

Figure 2-2 shows a diagram of design ADDER.

Figure 2-2 Hierarchical Design



The suggested optimization strategy for hierarchical designs is

1. Read in as much of the design hierarchy as you need to use in this session. For example, enter

```
dc_shell> read -format edif adder.edif
Loading edif file '/usr/design/adder.edif'
Current design is now '/usr/design/
HALF_ADDER.db:HALF_ADDER'
{HALF_ADDER, FULL_ADDER, ADDER}
```

2. Set the current design to the top of the design hierarchy. For example, enter

```
dc_shell> current_design ADDER
/usr/design/ADDER.db:ADDER
```

3. Check the design to determine whether any subdesigns (components) are referenced more than once (unresolved) or whether the hierarchy is recursive. For example, enter

   ```
   dc_shell> check_design
   Warning: Design ’FULL_ADDER’ is instantiated 4 times.
            Cell ’ADD0’ in design ’ADDER’
            Cell ’ADD1’ in design ’ADDER’
            Cell ’ADD2’ in design ’ADDER’
            Cell ’ADD3’ in design ’ADDER’
   ```

4. Resolve multiple instances.

   For each cell referencing a nonunique subdesign, do one of the following actions (you can use each action with one or more cells):

   - Use the `uniquify` command (with no options, this command resolves all multiple instances).
   - Combine the cell into the surrounding circuitry (ungroup).
   - Compile the cell separately, then use `set_dont_touch`.

   For more information about resolving multiple instances, see *Design Compiler User Guide*.

5. Check the design again to verify that all multiple instances have been uniquified, or ungrouped or have the `dont_touch` attribute set. For example, enter

   ```
   dc_shell> check_design
   ```

6. Compile the design.

   ```
   dc_shell> compile
   ```

Using the Compile-Characterize-Write Script-Recompile Strategy

The compile-characterize-write script-recompile strategy is an alternative to hierarchical compilation. Using this strategy, first optimize nonunique designs, using context information or time budgets. Then, optimize higher-level blocks with the lower blocks marked as `dont_touch`.

Use the compile-characterize-write script-recompile strategy for medium and large designs that do not have good interblock specifications (the typical situation for many designs).

The compile-characterize-write script-recompile strategy assists in compiling large designs, using the divide-and-conquer approach, and is not limited by memory.

The compile-characterize-write script-recompile strategy has these disadvantages:

- It requires iterations until the interfaces are stable.
- It requires manual revision control.

The compile-characterize-write script-recompile strategy requires seven steps:

1. Compile subblocks independently, using estimates for drive and load. Use a default script to estimate drive and load.
2. Read in the entire compiled design.
3. Characterize one subblock.

4. Use `write_script` to save the information from the characterization.
5. Clear memory; read in the previously characterized subblock; and recompile the subblock, using the saved script.

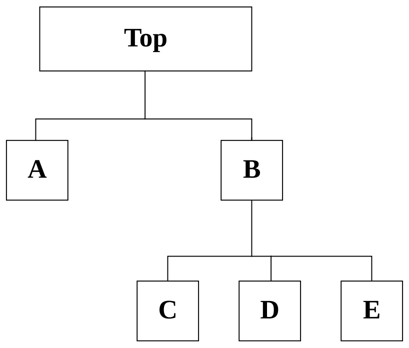
   For characterization information to apply, you must read in the database format that was characterized (in this case, .db).
6. Read in the entire compiled design again without the old subblock; use recompiled subblock.
7. Choose another subblock, and repeat steps 3 through 7 until all subblocks are recompiled, using their actual environments.

When you use the compile-characterize-write script-recompile strategy, consider the following:

- If you do not modify your RTL code after the first time you read it in, you can save it to a .db file (RTL to db). This step saves time later when you reread the design.
- `Compile` automatically goes to the submodule. If you want the compile to affect only the current design, you can either
  - Remove or omit the submodule from your database or
  - Use the `set_dont_touch` command to set the dont_touch attribute on the submodule.
- `Compile` is bottom up.
- `Characterize` is top down.

Example

Use the compile-characterize-write script-recompile compilation strategy for design Top.

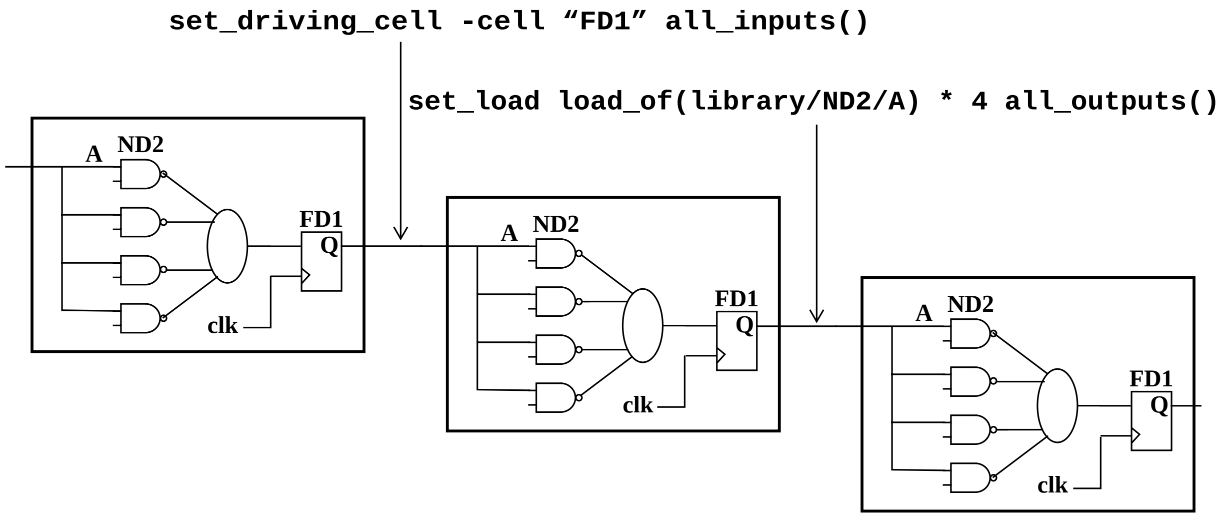


This compile script runs compile-characterize-write script-recompile for design Top. The compile script uses the default script to estimate the drive and the load that follow the compile script.

```
read -f db {top.db, B.db ...}
include defaults.scr
characterize B.blk

current_design B
write_script > b.wscr

remove_design -all
read -f db B.db
include B.wscr
compile

read -f db {top.db, A.db}
include defaults.scr
characterize A_blk

current_design A
write_script > A.wscr
```

This defaults script (defaults.scr) estimates drive and load.

```
set_operating_conditions “WCCOM”
set_wire_load_model -name “10x10”
create_clock -period 20 clk
dont_touch_network clk
set_input_delay -clock clk 4 all_inputs()
set_output_delay -clock clk 5 all_inputs()
set_load load_of(library/ND2/A) * 4 all_outputs()
set_driving_cell -cell “FD1” all_inputs()
set_drive 0 clk
```


set_driving_cell -cell “FD1” all_inputs()
set_load load_of(library/ND2/A) * 4 all_outputs()
A
ND2
FD1
Q
clk
A
ND2
FD1
Q
clk
A
ND2
FD1
Q
clk

3

Optimizing Designs

Optimizing a design produces the smallest design that meets your constraints. The `compile` command enables optimization.

When you compile, Design Compiler attempts to implement a combination of library cells that meets the functional, speed, and area requirements of a design according to the attributes and constraints placed on the design.

Design Compiler provides options that enable you to customize and control optimization.

Before you optimize your design, you need to know about the topics explained in the following sections:

- Before You Start
- Customizing Compilation

- Checking the Compile Log
- Running Compilation and Optimization
- Controlling Verification Effort
- Controlling Design Rule Fixing
- Controlling Sequential Inference
- Controlling Mapping Optimization
- Selecting Gate Sizes
- Directing Combinational Gate Implementation
- Preserving Unconnected Sequential Cells
- Mapping to Multiplexers
- Synthesizing Data-Path and Structured Logic
- Creating Clusters
- Removing Clusters
- Optimizing I/O Pads
- Controlling Final Sequential Optimization
- Defining a Signal for Unattached Slave Clocks
- Building a Balanced Buffer Tree
- Optimizing Across Boundaries
- Enabling Critical Path Resynthesis
- Optimizing for Area
- Optimizing for Delay

- Removing Hierarchy
- Running Incremental Compilation for Post-Placement Optimization
- Propagating Constants

Before You Start

To optimize successfully, do the following before you run the `compile` command:

- Ensure that partitioning is the best possible for synthesis.
- Define constraints as accurately as possible.
- Use a good synthesis library.
- Resolve multiple instances.
- Identify all multicycle and false paths.
- Instantiate clock gating elements.
- Optionally, define scan style.
- Determine the best strategy for optimizing your design.

Customizing Compilation

You can customize the `compile` command by using compile variables. For faster compilation, you can set the simple compile mode, which internally adjusts compile parameters. To optimize the top level and fix the intermodule timing violations and design rule violations in the design that might occur if you compile all subblocks separately and assemble them at the top level, you can specify the `-top` option of the `compile` command. This section discusses these features.

Compile Variables

Compile variables allow you to customize the `compile` command to fit your particular needs. Set variables from the dc_shell prompt, or define them in your .synopsys_dc.setup file.

The syntax is

```
variable_name = value
```

The individual variable descriptions define the value type. When *value* is a string, enclose it in quotation marks (" ").

To list compile variables and their current values, enter

```
dc_shell> list -variables compile
```

The `compile_variables` man page lists the compile variables.

Simple Compile Mode

For faster compilation, you can enable simple compile mode. In this mode, compile time is reduced because compile parameters are adjusted internally so that less compilation time is expended in the timing optimization, structuring, and area recovery phases.

Simple compile mode also configures the `compile` command to run on nonuniquified designs so that multiple instantiated designs are mapped only once. For a design with multiple instances, only one instance is compiled and the single mapped design is used for the other instances, resulting in a faster compile time. Using simple compile mode for a design with multiple instances obviates the need to uniquify all instances of the design and compile each one individually. You can use simple compile mode when working with DesignWare.

To take full advantage of the compile method available in simple compile mode, you should not uniquify designs.

For many designs, simple compile mode produces substantially the same quality of results as other compile modes but with shorter compile times.

Simple compile mode uses area base allocation and resource implementation and sets the following variables to `area_only`:

- `hlo_resource_allocation = area_only`
- `hlo_resource_implementation = area_only`

Note:

> If your design has tight timing constraints, you should consider, before using this mode, that several of the compile parameters adjusted for simple compile mode affect timing optimization.

To enable simple compile mode, use the `set_simple_compile_mode` command before compilation.

The syntax is

```
set_simple_compile_mode [ true | false ] [-verbose]
```

Argument	**Description**
true | false	If true, places Design Compiler in simple compile mode. If false, places Design Compiler into standard mode. (Default is false.)
-verbose	Prints out more-descriptive informational messages.

To enhance the compilation time of the simple compile mode, you can set the following variables:

```
compile_simple_mode_block_effort =
   [ low | medium | high ]
```

This variable specifies the map effort is used for subdesigns when using simple compile mode. If this variable is not set, then simple compile mode uses the same map effort for the subdesigns as specified for the top design.

`compile_dw_simple_mode = [ true | false ]`

This variable when set to true, the DesignWare subdesigns are run in simple compile mode. When working DesignWare and simple compile mode and your design contains multiple instances, only one instance is compiled and the single mapped design is used for the other instances.

Top-Level Timing Resolution Compile Option

If you compile all subblocks in a design separately to meet timing constraints and assemble these subblocks at the top level of the design, violations might occur on the top-level hookup. To direct Design Compiler to fix intramodule timing violations and design rule violations in the design and to optimize the top level, use the `compile -top` option.

To use this compile option, replace the `compile -incremental` command with the `compile -top` command. Use of the `-top` option instead of `-incremental` typically improves runtime performance by a factor of 8 without degrading the quality of results.

Use the `compile -top` option with mapped designs only. This option does not perform any mapping, incremental implementation selection, or area recovery on the design.

The `-no_area_recovery` and `-no_iis` compile options are `compile -top` option defaults.

Use of Compile -top Option With Other Compile Options

The `compile -top` option is incompatible with the following options:

- `-incremental_mapping`
- `-routability`
- `-exact_map`
- `-no_map`
- `-area_effort`

Limiting Optimization to Paths Within a Specific Range

By default, the `compile -top` option aims to fix all design rule violations in the entire design but only the intermodule timing paths with violations. It does not address the intramodule timing paths. To direct Design Compiler to attempt to optimize the intermodule paths with timing delay violations within a specific range, set the critical_range attribute before you run the `compile` command with the `-top` option.

Widening Optimization to Include All Design Rule Violations and Intramodule, Intermodule Paths

By default, when the `compile -top` option is used, Design Compiler addresses intermodule timing paths and ignores intramodule timing paths. Consequently, designs with many intramodule timing violations do not show significant changes in optimization results.

Setting the `compile_top_all_paths` environment variable to true causes the compile –top option to attempt to fix all design rule violations in the entire design and all intermodule and intramodule paths with violations.

Checking the Compile Log

During optimization, each time Design Compiler starts a step it prints a message in the compile log, indicating its progress.

For example

```
dc_shell> compile
```

Beginning Mapping Optimizations (Medium effort)

Trials	**Area**	**Delta Delay**	**Total Neg Slack**	**Design Rule Cost**
3	1296477.2	7.58	3468.1	2.9
1	1296538.9	7.48	3382.7	2.9

For additional information, refer to “Analyzing and Debugging Your Design” chapter in the *Design Compiler User Guide*.

Running Compilation and Optimization

The `compile` command and its options control design optimization. Options apply to the current design and to its subdesigns.

When the current design is hierarchical, the `compile` command checks subdesigns before optimization for multiple design instances. If multiple design instances exist that are not marked `dont_touch` or `ungroup`, `compile` displays an error message and exits.

The compile commands `set_flatten` and `set_structure` are available to customize the optimization of subdesigns.

If the current design or any of its subdesigns is represented as a state table, the compile process automatically performs finite state machine optimizations on these designs.

You can redirect or append the output of the commands to a file you can review. This way, you can archive runtime messages for future reference.

To Do This	**Use This**
Divert command output to a file.	> (redirection operator)
Divert command output and error messages to a file.	>& (redirection operator and ampersand)
Append command output to a file.	>> (append operator)

Note:

The pipe character (`|`) has no meaning in the dc_shell interface.

The syntax is

```
compile [-no_map] [-map_effort low | medium | high]
     [-scan] [-incremental_mapping] [-exact_map]
     [-routability [-add_porosity]] [-verify]
     [-verify_hierarchically]
     [-verify_effort low | medium | high]
     [-ungroup_all][-boundary_optimization]
     [-no_design_rule][-only_design_rule]
     [-background run_name] [-host machine_name]
     [-arch architecture] [-xterm] [-top]
```

Argument	**Description**
`-no_map`	Optimizes your design without mapping it to a target technology.
`-map_effort low | medium | high`	Sets the relative amount of CPU time Design Compiler spends mapping the design.
`-scan`	Synthesizes all sequential cells directly to scan cells. This option is called test-ready compile. Test-ready compile requires both a Design-Compiler license and a Test-Compiler license. For more information, see the *Scan Synthesis User Guide*.
`-incremental_mapping`	Maps unmapped parts of the design, then attempts incremental improvements to the gate structure.
`-exact_map`	Specifies that the sequential elements in the final design must exactly match the HDL descriptions.
`-routability`	Optimizes the design for routability using the porosity constraint specified with the set_min_porosity command.
`-add_porosity`	Favors the porosity cost over the area cost for compile -routability.

Argument	Description
`-verify`	Verifies the logical function of the newly optimized design against the original design. To verify a state machine, use the compare_fsm command.
`-verify_hierarchically`	Performs verification on a copy of each subdesign of the current design.
`-verify_effort low | medium | high`	Sets the relative amount of CPU time for verifying this optimization. The default effort is low. If you use this option, you must use the -verify option. For more information, see “Controlling Verification Effort” on page 3-17.
`-ungroup_all`	Ungroups all subdesigns (collapses all hierarchy levels) not marked with the dont_touch attribute.
`-boundary_optimization`	Optimizes across all subdesign boundaries. For more information, see “Optimizing Across Boundaries” later in this chapter.
`-no_design_rule`	Compiles without performing the design rule fixing phase. For more information, see “Controlling Design Rule Fixing” later in this chapter.
`-only_design_rule`	Performs a limited compilation that does only incremental mapping and design rule fixing. For more information, see “Controlling Design Rule Fixing” later in this chapter.
`-background run_name`	Runs the job in the background, creates the directory you specify (full or relative path name), and stores the optimization results in that directory.
`-host machine_name`	Name of the machine on which the background job runs.
`-arch architecture`	Architecture of the machine where the background job is run. (For example, sparc.)

Argument	Description
`-xterm`	Allows you to monitor the results of a background job in an X-terminal window. The -xterm option is valid only with the -background option in an X11 environment.
`-top`	Fixes design rule and top-level timing violations for a design; does not perform any mapping on the design.

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

Note:

The DC Ultra version of the `compile` command `-map_effort high` option includes algorithms that enable mapping to wide-fanin gates. This reduces critical path length.

Controlling Verification Effort

Because verifying some designs can be long, setting the `-verify_effort` option allows you to place a limit on the amount of CPU time devoted to verification. Verification stops when the limit is reached.

`-verify_effort low`

Is the default. Design Compiler does not pursue parts of the design that are difficult to verify.

`-verify_effort medium`

Uses more CPU time to verify the design.

`-verify_effort high`

Attempts to verify the design completely. This process might never terminate for some designs.

Controlling Design Rule Fixing

You can direct Design Compiler to avoid design rule fixing or to compile with only design rule fixing. The `-no_design_rule` and `-only_design_rule` options determine whether design rule violations are fixed before compilation stops. You cannot use these options together.

`-no_design_rule`

Causes compile to exit before fixing design rule violations. This allows you to check the results in a constraint report before fixing the violations.

`-only_design_rule`

Causes compile to perform only design rule fixing. Mapping optimizations are not performed.

If you omit these options, Design Compiler performs both design rule fixing and mapping optimizations before exiting (the default).

Note:

Another way to limit design rule fixing is by using the `set_cost_priority` command with the `-delay` option. See “Phase 3: Design Rule Fixing, Part 2” in Chapter 1.

Controlling Sequential Inference

You can control sequential inference in the following ways:

- Use specific HDL coding styles.
- Use exact mapping (`compile -exact_map`).
- Use test-ready compile (`compile -scan`).

Using Specific HDL Coding Styles

Write the HDL description to pass hints to the optimizer indicating which sequential elements are preferred on an instance-by-instance basis. The hints are considered before technology-independent optimization, and the corresponding sequential elements are instantiated at that time. This methodology makes it easier to control the kinds of sequential elements inferred during compile.

Using Exact Mapping

Use the `compile` command `-exact_map` option to restrict the inference to sequential cells with simple behavior (synchronous set and reset, synchronous toggle, synchronous enable, asynchronous set and reset, and asynchronous load and data). When you use the `-exact_map` option, sequential inference does not try to encapsulate combinational logic originally outside the generic sequential element (SEQGEN) into the sequential cell.

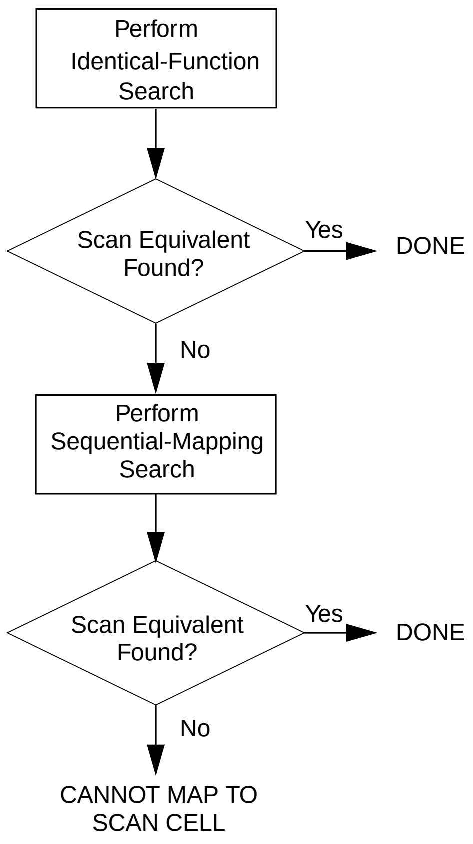
Using Test-Ready Compile

If you are using scan design as your design-for-test (DFT) technique, use the `compile` command `-scan` option to map sequential cells directly to scan cells. Test-ready compile reduces iterations and design time, by accounting for the impact of the scan implementation during the logic optimization process. When you use test-ready compile, DC Expert *Plus*

- Maps all sequential cells directly to the scan equivalent cells
- Ties the test control pins to the appropriate state to enable functional mode
- Connects the Q output of the cell to the scan input pin of the cell to model the scan loading effect

DC Expert *Plus* uses the process shown in Figure 3-1 to map a sequential cell to its scan equivalent.

Figure 3-1   Mapping to Scan Cells



Identical-function mode locates a scan equivalent, using the information in the test_cell group of the library description. A valid scan equivalent must meet the following conditions:

- The scan cell and the nonscan cell must have the same functional input and output pins.
- The functional description in the test_cell group of the library description must exactly match the functional description of the nonscan cell.

Sequential-mapping mode locates a scan equivalent by using the Design Compiler sequential-mapping algorithms. DC Expert *Plus* uses this mode only for edge-triggered nonscan sequential cells. Sequential mapping selects the lowest-cost scan equivalent, which can be a scan cell plus combinational gates.

Identical-function mode is faster than sequential-mapping mode but might not find a scan equivalent in some cases, such as for complex flip-flops. If neither identical-function mode nor sequential-mapping mode can find a scan equivalent, DC Expert *Plus* uses the nonscan cell.

Controlling Mapping Optimization

Select appropriate effort levels for mapping optimization by using the compile command `-map_effort` and `-area_effort` options to select the map effort.

`-map_effort low`

Takes the least time to compile. Use low if you are running test to check the logic. Low is not recommended if the design must meet timing or area goals.

`-map_effort medium`

Is the default. Design Compiler tries to find a good mapping but does not use some CPU-intensive strategies. Medium is appropriate for getting a quick idea of how large a circuit will be. Use medium for most cases.

`-map_effort high`

Takes significantly longer to compile but can produce better designs. The mapping process proceeds until it has tried all strategies. This setting enables critical path resynthesis.

Note:

The DC Ultra version of the `compile` command `-map_effort` high option includes algorithms that enable mapping to wide-fanin gates. This reduces critical path length.

Selecting Gate Sizes

Design Compiler has three primary objectives when it selects gate sizes:

1. Improve optimization results with libraries that contain many (for example, eight) drive strengths
2. Enhance selection of cell sizes and reduce occurrences of unnecessary buffers on critical paths
3. Reduce buffering during design rule fixing

Better cell selection during the initial mapping is a key to obtaining better optimization results. Design Compiler performs a library analysis step at the start of compilation to evaluate the set of cells available for each function. It identifies the cells with the best delay characteristics for the design, based on the pin loads and wire load model specified, then uses those cells for the initial mapping. Instantiated cells are automatically sized up if the library analysis step determines that the starting size is not the optimal size.

You can control the cells Design Compiler uses for the initial mapping, by setting a preferred attribute on them, using the `set_prefer` command. Design Compiler can still change or size those cells during subsequent optimization steps.

Directing Combinational Gate Implementation

The `set_combinational_type` command specifies the library gate to which a combinational GTECH gate maps in your design. Use this command when you want Design Compiler to implement a specific library gate. Once Design Compiler maps a cell, it disables any further optimization of that cell.

The syntax is

```
set_combinational_type
     -replacement_gate replacement_gate_name
     cell_list
```

Argument	Description
`-replacement_gate` *replacement_gate_name*	Specifies the library gate that Design Compiler uses to replace the cells in the cell list.
cell_list	Specifies the cells that the library gate replaces.

For example,

To replace cell U1 with the library cell AN2P, enter

```
dc_shell> set_combinational_type -replacement_gate AN2P U1
```

Preserving Unconnected Sequential Cells

During optimization, Design Compiler deletes sequential cells having outputs that do not drive any loads. The combinational logic cone associated with the input of the sequential cell can also be deleted if the cell is not used elsewhere in the design. Sequential cell outputs can become unconnected due to redundancy in the circuit or as a result of constant propagation. In some designs where the sequential cells have been instantiated, the outputs might already be unconnected.

You can preserve such sequential cells if you need to retain them to maintain consistency between the compiled design and the HDL source or for other design reasons, such as in the case of instantiated cells. Direct Design Compiler to preserve the sequential cells by setting the `compile_delete_unloaded_sequential_cells` variable to false before you compile.

The syntax is

```
compile_delete_unloaded_sequential_cells = false | true
```

The default is true. When this variable is set to false, a warning message appears during compilation, indicating the presence of unloaded sequential cells.

```
Warning: In design ‘design_name’, there are sequential cells
not connected to any load. (OPT-109)
Information: Use the ‘check_design’ command for more
information about warnings. (LINT-99)
```

You can use the `check_design` command after compilation to identify cell instances that have unconnected outputs.

In cases where a feedback loop exists with no path from any section of the loop to a primary output, no warning appears because the output of the sequential cell is theoretically connected. When `compile_delete_unloaded_sequential_cells` is set to true, such cells are optimized away. Setting the value to false retains the cells and the logic cone associated with the inputs to the cells.

Mapping to Multiplexers

Design Compiler can map combinational logic representing multiplexers in the HDL code directly to a single multiplexer (MUX) or a tree of multiplexer cells from the target technology library.

Note:

This feature is supported only with the use of the case statement in VHDL or Verilog code.

You can embed an attribute or directive in the HDL code or use variables to tell the (V)HDL compiler which part of the HDL description to implement as a single multiplexer or a tree of multiplexers. When the HDL is read in, a generic cell called MUX_OP cell represents the multiplexer functionality. During optimization, Design Compiler maps the logic inside the MUX_OP cell to an implementation, using multiplexer cells from the library.

For complete information about multiplexer inference from HDL and the limitations on inference, see the *Guide to HDL Coding Styles for Synthesis*.

The MUX_OP Cell

The MUX_OP cell is a generic representation of an N:1 multiplexer with M outputs. When VHDL or Verilog code is read in, the resulting design contains a MUX_OP cell for every case block inside a process that contains the infer_mux directive. A MUX_OP cell also is inferred for each signal (including bused signals) assigned inside the same case block. The signals in the HDL that compute the selector are connected to the select inputs of the MUX_OP cell.

The naming convention for MUX_OP cells in a design is

`_MUX_OP_`*`N_S.M`*

`N`

    The number of data inputs.

`S`

    The number of select input.

`M`

    The width of the output signal.

During compilation, a new level of hierarchy is created around the implementation of each MUX_OP cell, in the same way that hierarchy is created for each operator such as add, subtract, compare, and multiply. This preserves the multiplexer structure and allows incremental implementation selection for subsequent compilations.

If any of the following is true, the level of hierarchy is not preserved:

- The `compile` command `-ungroup` option is used.
- The variable compile_create_mux_op_hierarchy is set to false. (The default is true.)

- The attribute mux_op_ungroup is true on the MUX_OP cell or its parent designs.
- The number of data inputs to the MUX_OP is fewer than two.

The MUX_OP cell appears in the design as a resource (such as an adder or subtracter). To get information about the MUX_OP cell parameters, use the `report_resources` command.

Multiplexer Optimization

Inference of the MUX_OP cell is necessary for Design Compiler to create the multiplexer tree. During compilation, Design Compiler replaces each MUX_OP cell with the best multiplexer tree implementation of the N-input M-output multiplexer. The implementation depends on the constraints given for the design and the arrival times of the selector inputs. If the hierarchy of the MUX_OP cell is preserved, incremental implementation selection is performed on subsequent compilations when the constraints change.

Design Compiler does not trade off multiplexer implementations defined in the DesignWare library during optimization.

Optimization of the MUX_OP cell involves the following:

- Library cell requirements
- Boundary optimization and constant propagation

Library Cell Requirements

The multiplexer optimization requires the presence of at least a 2:1 multiplexer cell in the technology library. The inputs or outputs of this cell can be inverted.

If a 2:1 multiplexer primitive cell does not exist in the library, you see the warning message

```
Warning: Target library does not contain any 2-1 multiplexer.
(OPT-853)
```

An implementation of the MUX_OP cell from the target library is created, but it might not be the best implementation possible. All multiplexer cells in the target library can be used to construct the implementation of the MUX_OP cell except

- Enabled multiplexer cells
- Bused output multiplexer
- Multiplexers larger than 32 : 1

For Design Compiler to make the best use of the multiplexer cells available in your technology library, recompile the library or obtain a library compiled with version V3.4a or later from your ASIC vendor.

Boundary Optimization and Constant Propagation

Boundary optimization occurs by default on the MUX_OP cells, because MUX_OP cells are treated as synthetic designs and boundary optimization takes place by default on such designs.

Boundary optimization optimizes the multiplexer tree, using the logical relationships of input and output pins. It propagates constants, unconnects, and equal or opposite information across hierarchical boundaries. This optimization can simplify and collapse the multiplexer tree to gain area.

If you want to preserve the multiplexer structure, even at the cost of some area, you can turn off boundary optimization. To do this, set the following variable before compilation:

```
compile_mux_no_boundary_optimization = true
```

You can turn off boundary optimization on an instance-specific basis, using the attribute `mux_no_boundary_opt`. For example, to turn off boundary optimization for cell instance U1/U2, enter

```
dc_shell> set_attribute find(cell, U1/U2)
     mux_no_boundary_opt  true -type boolean
```

Using MUX_OP Cells

The MUX_OP cell should be inferred when you want Design Compiler to build a multiplexer tree structure for the case statement blocks in your HDL. MUXs can be implemented efficiently (in speed and area) in the library.

This type of structure can provide advantages in circuit performance and savings in wiring area, compared to implementation constructed from random logic.

Because the MUX_OP cell is implemented as a level of hierarchy, certain types of logic sharing that might otherwise take place are no longer possible. In some cases, this can lead to a design that is less suitable than one having no MUX_OP cells.

This feature enables you to direct Design Compiler to give a MUX tree structure for a given case statement when you already know that a MUX tree is the best representation for that statement. Design Compiler optimizes the tree based on constraints and yields the best-possible MUX tree.

Observe the following when you use MUX_OP cells:

- Do not set the HDL Compiler variable `hdlin_infer_mux` to all when you compile. Setting this variable to all results in implementing MUX_OP cells for every case statement in your HDL. MUX_OP cells should be inferred for case statements that can benefit most from a multiplexer tree structure.

- Incompletely specified case statements can result in MUX_OP cells with unused inputs or a partially collapsed multiplexer tree structure. In such cases, the difference between using a MUX_OP cell and implementing the multiplexer with random logic might not be significant.
- In general, most gains are realized when you use a MUX_OP cell for fully specified case statements that implement large multiplexer logic. If your design uses mostly small (2:1 or 4:1) multiplexers, you might get better results not by using MUX_OP cells.

Reporting the Presence of MUX_OP Cells

The MUX_OP cell appears in reports as a synthetic operator. The `report_resouces` command displays information about the presence of MUX_OP cells and their parameters.

For example,

This is the report before compilation for a design that contains a _MUX_OP_8_3.2 design with eight data inputs and 2-bit-wide output.

```
***************************************
Report :      resources
Design:       mux8_2
Version:      1997.08
Date:         Tues Aug 23 1997
***************************************
```

```
Resource sharing reports are created during the compile command, so the design
must be compiled before resource sharing can be reported.

No implementations to report

Multiplexor Report
==========================================================
| Cell   | WidthDataReference|
==========================================================
| U1     | 2   | 8   |_MUX_OP_8_3.2|
==========================================================
```

The following example is the report after compilation:

```
***************************************
Report :      resources
Design:       mux8_2
Version: 1997.01
Date:         Wed Jan 14 1997
***************************************
No resource sharing information to report.
No implementations to report
Multiplexor Report
============================================================
| Cell              | Width    | Data      |     Reference|
============================================================
| U1                | 2        | 8         |     _MUX_OP_8_3.2|
============================================================
```

Setting Variables That Affect Multiplexer Mapping

The following variables affect multiplexer mapping.

`compile_create_mux_op_hierarchy = true | false`

> Set to true (the default), implies that MUX_OP implementations (except those with fewer than two data inputs) will have their own level of hierarchy. If you do not want a separate level of hierarchy for the MUX_OP cells, set this to false. After the hierarchy of the MUX_OP is removed, usually the original multiplexer structure cannot be regained.

`compile_mux_no_boundary_optimization = true | false`

> Set to false (the default), enables boundary optimization on the MUX_OP. To disable boundary optimization and preserve the multiplexer tree structure, even at the cost of greater area or delay, set this to true.

Synthesizing Data-Path and Structured Logic

You can use Design Compiler to synthesize certain sequential data-path logic and structured logic, just as you use the tool to synthesize control logic. Specifically, Design Compiler can map the following sequential data-path and structured logic to multi-bit library cells in vendor libraries:

- flip-flops
- latches
- master-slave circuits

- multiplexers
- three-state circuits

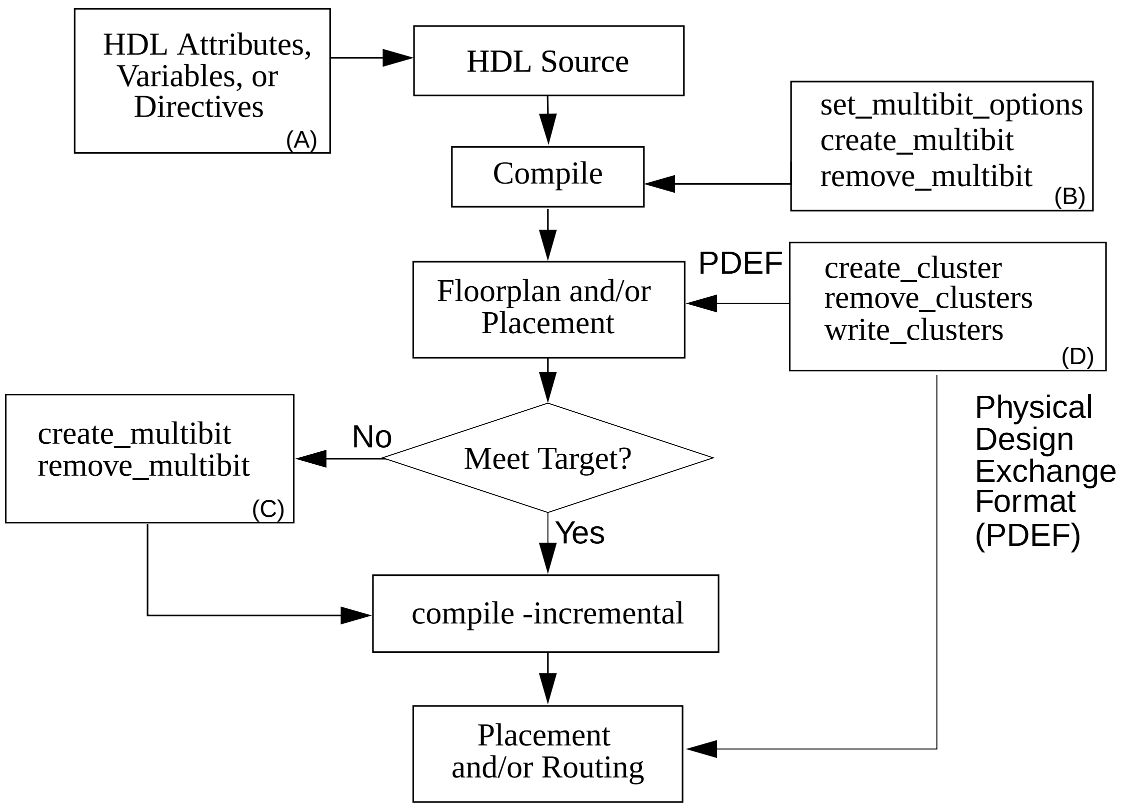
Multi-bit library cells consume less power and area and create a more uniform layout structure than single-bit equivalents can attain.

Design Compiler provides two methodologies for mapping the data-path and structured logic to multi-bit library cells. (You can use either methodology or a combination of the two.) The first is to direct cell inference from the HDL source code. This method is best if you know the design’s layout and can determine where multi-bit cells might have the most impact. This might be the case, for example, if the data-path and control logic are well separated or if you have done early floorplanning. For more information, see the *HDL Compiler for Verilog Reference Manual* and the *VHDL Compiler Reference Manual*.

The second methodology directs multi-bit library cell inference from an already mapped design. This method is most useful after you complete an initial floorplan or placement and determine which areas can benefit from the use of multi-bit cells.

Figure 3-2 is an overview of the methodologies for data-path and structured logic mapping.

Figure 3-2   Data-Path and Structured Logic Mapping



(A) Define multi-bit components, using HDL attributes, variables, or directives.

(B) Define or revise multi-bit components and control multi-bit optimization, using these commands.

(C) Define or revise multi-bit components, using these commands.

(D) Define or revise physical clusters, using these commands.

Design Compiler supports only multi-bit library cells that have identical functionality for each bit. The multi-bit library cell interfaces must be either fully parallel or fully global. For example, if you want to infer a 4-bit banked flip-flop with an asynchronous clear, the clear signal must be either different for each bit or shared among all 4 bits. Design Compiler cannot infer a multi-bit register if the first and second bits share one asynchronous reset but the third and fourth bits share another reset. In that case, Design Compiler does not infer a multi-bit flip-flop but uses 4 single-bit flip-flops instead. You must instantiate the multi-bit flip-flop.

Reporting Multi-Bit Components

A multi-bit component is a group of cells with identical functionality. Two cells can have identical functionality even if they have different bit-widths. Thus, a group of cells including one 3-bit register and one 5-bit register is a multi-bit component (assuming identical functionality). It would still be called a multi-bit component if it were implemented using eight single-bit cells.

You can infer a multi-bit component from the HDL source code by adding directives, or you can create it from Design Compiler, using the `create_multibit` command. The compilation process preserves multi-bit components even if their implementations undergo changes.

Use the `report_multibit` command to report all multi-bit components in your current design. The report lists the multi-bit component name and the cells that implement each bit. (You can use the command on a mapped or an unmapped design.)

The syntax is

```
report_multibit [-nosplit] [object_list]
```

Argument	Description
-nosplit	Prevents line splitting in fixed-width report columns. This facilitates writing software to extract information from the report output.
object_list	Names the cells and multi-bit components to list in the report. If you do not include this option, Design Compiler reports all multi-bit components in the current design.

Here is a sample report produced by the `report_multibit` command.

```
Report: Multibit
Design: your_design
Version: 1998.02
Date: Wed, Feb 12 11:57:12 1998

Multibit Component : U813_multibit

Cell                    Reference    Library        Area   Width  Attributes
------------------------------------------------------------------------------
U813                    mux4x16     your_library    96.00    16
U9101                   mux4x16     your_library    96.00    16
------------------------------------------------------------------------------
Total 2 cells                                      192.00   32
------------------------------------------------------------------------------

Multibit Component : data_reg
Cell                    Reference    Library        Area   Width  Attributes
------------------------------------------------------------------------------
data_reg[0:15]          ff2x16     your_library    48.00    16      n
data_reg[16:31]         ff2x16     your_library    48.00    16      n
------------------------------------------------------------------------------
Total 2 cells                                     96.00    32
------------------------------------------------------------------------------
```

Design Compiler uses a colon to identify multi-bit component registers with consecutive bits (0 through 15 and 16 through 31 in the previous report). If the colon conflicts with your back-end tool's naming requirements, change the colon to another delimiter using the `bus_range_separator_style` variable.

The tool uses a comma to separate nonconsecutive bits. For example, if you use bits 0 through 5 and bit 7 in the multi-bit component, the report lists them as 0 : 5, 7. The `bus_multiple_separator_style` variable controls this delimiter.

Finding Multi-Bit Components

To see a list of all multi-bit components in the current design, use the `find` command, with multibit as the object type.

Example

To find all multi-bit components in your design, enter

```
dc_shell> find multibit *
```

Using the sample report in the previous section, Design Compiler would find U813_multibit and data_reg.

Controlling Multi-Bit-Component Optimization

To control the Design Compiler process of optimizing your design's multi-bit components, use the `set_multibit_options` command. This command sets two attributes on the design: `multibit_mode` and `minimum_multibit_width`.

The `multibit_mode` attribute specifies how multi-bit components are optimized during the compile run. There are four modes: `user_driven`, `structured`, `start_multibit`, and `start_singlebit`. The `minimum_multibit_width` attribute indicates the smallest bit-width that Design Compiler optimizes as a multi-bit component.

You can direct Design Compiler to report the values of the `multibit_mode` and `minimum_multibit_width` attributes, using the `report_compile_options` command.

During compilation, Design Compiler uses only the `multibit_mode` and `minimum_multibit_width` attributes set on the current design. The tool ignores values set on subdesigns. If the library to which you are mapping your design does not contain multi-bit library cells for a certain functionality, Design Compiler implements the function with single-bit library cells.

The syntax of the `set_multibit_options` command is

```
set_multibit_options [-default]
     [-mode [user_driven | structured | start_multibit | \
               start_singlebit]]
     [-minimum_width width]
```

Argument	Description
-default	Sets the multibit_mode and minimum_multibit_width attributes so that the compile command uses default values.
user_driven	This is the default value for the -mode option. It maps multi-bit components to multi-bit cells in the technology library (or to single-bit cells if no multi-bit cells are available). Use this mode if you know from experience where Design Compiler should map multi-bit components to improve post-layout results. This mode uses multi-bit library cells even if single-bit cells can improve timing.
structured	Maps multi-bit components to single-bit cells in the technology library—even if the library contains multi-bit cells. Use this mode to get a uniform single-bit structure for all cells in the multi-bit component.
start_multibit	Infers multi-bit components early in the compilation process. Design Compiler compares an implementation of multi-bit cells with an implementation of single-bit cells and uses the one that produces the best pre-layout delay and area results. The tool might use cells of differing functionality to meet your constraints.
start_singlebit	Infers multi-bit components late in the compilation process. Design Compiler compares an implementation of multi-bit cells with an implementation of single-bit cells and uses the one that produces the best pre-layout delay and area results. The tool might use cells of differing functionality to meet your constraints.
-minimum_width *width*	Specifies the value Design Compiler gives to the minimum_multibit_width attribute. The default value is 2.

Examples

To set the `multibit_mode` attribute to structured, enter

```
dc_shell> set_multibit_options -mode structured
```

To set the `multibit_mode` and `minimum_multibit_width` attributes to default values, enter

```
dc_shell> set_multibit_options -default
```

To direct Design Compiler not to optimize multi-bit components of less than 4 bits, enter

```
dc_shell> set_multibit_options -minimum_width 4
```

Inferring Multi-Bit Library Cells From Already Mapped Designs

You might want Design Compiler to infer multi-bit library cells as multi-bit components on an already mapped design if, for example, the design includes a section of bit-sliced logic that can benefit from a more uniform layout.

To control multi-bit-component inference, use the `create_multibit` and `remove_multibit` commands.

Creating Multi-Bit Components

To create multi-bit components in your design, use the `create_multibit` command.

The syntax is

```
create_multibit object_list [-name multibit_name]
     [-sort] [-no_sort]
```

object_list

Lists cells in the design for which Design Compiler creates a multi-bit component.

`-name` *multibit_name*

Names the new multi-bit component. If you omit this option, Design Compiler uses the name of the first cell in the multi-bit component. If you enter a name that already exists in the design, the tool issues an error message and terminates the command.

`-sort`

Sorts the object_list cells in alphanumeric order by their cell names. If you omit `-sort`, Design Compiler sorts the bits in reverse alphanumeric order by default.

`-no_sort`

Sorts the object_list cells in the order you specify. If you omit `-no_sort`, Design Compiler sorts the bits in reverse alphanumeric order by default.

For example, to create multi-bit components named y_reg[0] through [3] and sort them in 0-2-1-3 order, enter

```
dc_shell> create_multibit -name y{y_reg[0] y_reg[2] y_reg[1]
y_reg[3]} -no_sort
```

Removing Multi-Bit Components

To delete multi-bit components from your design, use the `remove_multibit` command.

The syntax is

```
remove_multibit object_list
```

object_list

Lists the multi-bit component names or cells to remove from the design.

Examples

To remove the cell y_reg[2] from a multi-bit component, enter

```
dc_shell> remove_multibit y_reg[2]
```

To remove multi-bit component y from your design, enter

```
dc_shell> remove_multibit y
```

Recompiling the Design With Multi-Bit Components

After you use the `create_multibit` or `remove_multibit` command, recompile the design, using the `compile` or `compile -incremental` command. Design Compiler builds multi-bit components (or reduces multi-bit components to single-bit components) as it recompiles the design. While doing so, the tool also optimizes other cells in the design. If you want to affect only the multi-bit components, set a `dont_touch` attribute on the other cells in the design.

Controlling the Use of Multi-Bit Library Cells

Large multi-bit cells might cause routing congestion or might be too inflexible for your design. To prevent Design Compiler from using multi-bit library cells larger than 8 bits, for example, use the `set_dont_use` command with the `multibit_width` library attribute, as follows:

```
dc_shell> filter find(cell, library name/*) "multibit_width > 8"
dc_shell> set_dont_use dc_shell_status
```

Creating Clusters

After Design Compiler implements your design, you might want your placement tool to place certain components close together. To do so, use the `create_cluster` command. The command creates clusters and attaches them to either a parent cluster or to the design itself. For example, if newly created cluster B is attached to existing cluster A, cluster A is the parent of cluster B.

Note:

> To use this command requires a DC Ultra license or Floorplan Manager license.

The syntax is

```
create_cluster [-name cluster_name] [-keep] [-multibit]
    [-parent parent_cluster_object][object_list]
```

`-name` *cluster_name*

Names the cluster to create.

`-keep`

Specifies that an object is not placed in a cluster if the object is already in another cluster.

`-multibit`

Specifies that a new cluster is created for every multi-bit component in the design. This option is mutually exclusive to all other options except `-keep`.

`-parent` *parent_cluster_object*

Names the existing cluster to be the parent of the new cluster.

object_list

Names the cells, multi-bit components, or cluster objects you want in the new cluster.

Examples

To create clusters for each multi-bit component group in the design, enter

```
dc_shell> create_cluster -multibit
```

To create a cluster foo containing cells U1 and U2, enter

```
dc_shell> create_cluster -name foo find(cell,"U1,U2")
```

Removing Clusters

The `remove_clusters` command removes the cluster hierarchy associated with a design.

Note:

> To use this command requires a DC Ultra license or Floorplan Manager license.

The syntax is

```
remove_clusters [design_or_cluster]
```

design_or_cluster

> Specifies the cluster within a design you want removed or the entire design whose cluster hierarchy you want removed. If you omit this option, Design Compiler removes the cluster hierarchy on the entire design.

For example, to remove the cluster hierarchy on the entire design, enter

```
dc_shell> remove_clusters
```

Optimizing I/O Pads

Design Compiler fully maps the combinational logic and initially maps the sequential logic before it inserts and maps the I/O pads.

Design Compiler can use I/O pad cells that are more complex than simple buffers.

For more information, see Chapter 7, “Adding I/O Pad Cells.”

Controlling Final Sequential Optimization

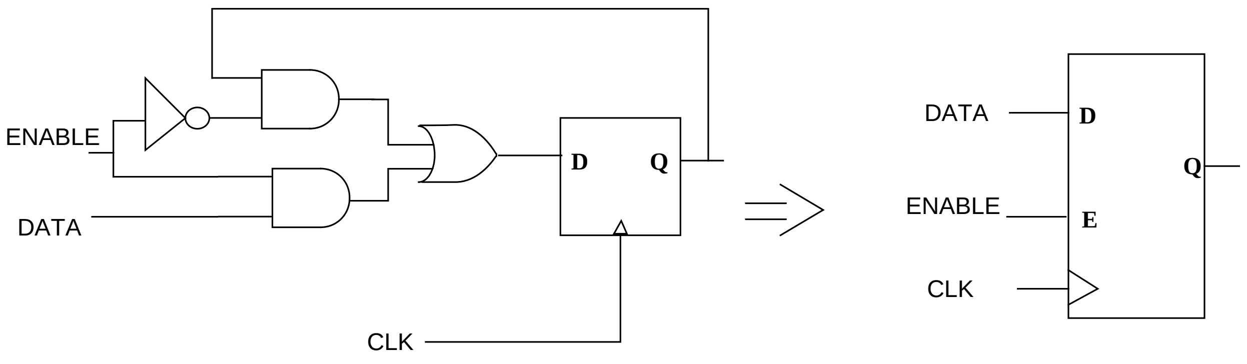
During this phase, Design Compiler optimizes timing-critical sequential cells (those on the critical path). The tool examines each sequential cell and its surrounding combinational logic to determine if it can replace the cell with a more complex sequential cell from the target library to improve delay.

The advantages of this optimization are the following:

- Design timing can be improved by choosing higher-performance sequential cells and by reducing the number of levels of logic.
- Gate-level optimization can possibly reduce the area and delay if complex sequential cells exist in the library. These improvements are accomplished by incorporating the combinational logic in the sequential cell, as shown in Figure 3-3.

To improve area during optimization, you can use the `compile_sequential_area_recovery` variable. When this variable is set to true, compile will try to re-map the sequential elements to recover area.

Figure 3-3 Sequential Optimization



Turning Off Final Sequential Optimization

You can turn off sequential optimization by using these `compile` command options:

`-map_effort low`

> Specifies the relative amount of CPU time spent during the mapping phase of `compile`. The default is medium.

`-exact_map`

> Turns off sequential optimization. Sequential elements inferred are not optimized later. This process improves predictability—but possibly at the expense of delay and area quality.

For more information about these `compile` command options, see the man pages.

Directing Register Implementations

The `set_register_type` command specifies the default flip-flop or latch library cell type for some or all registers in the current design or current instance. A flip-flop type is represented by an example flip-flop; any flip-flop that has the same sequential characteristics as the specified flip-flop is considered to be of that type.

Use `set_register_type` to direct Design Compiler to infer a particular flip-flop or latch.

Note:

> This mechanism for directing register types is not needed if you write the HDL to directly infer the correct register type.

You can specify a latch, a flip-flop, or both.

The syntax is

```
set_register_type [-exact -latch example_latch] [[-exact]
     -flip_flop example_flip_flop ][cell_or_design_list]
```

`-exact -latch` *example_latch*

> Defines the example latch as the exact latch from the target library to use during compilation. If you use `-latch`, you must use `-exact`.

`-exact`

> Defines that an exact mapping to the *example_flip_flop* or the *example_latch* is made during compilation, if possible.

`-flip_flop example_flip_flop`

> Defines the example flip-flop as the exact flip-flop from the target library to use during compilation.

`cell_or_design_list`

Lists cells and designs in which to use the latch or flip-flop. If you omit this list, the latches or flip-flops are added to the current design (the default).

For example,

To set the default flip-flop type to FFX and the default latch type to LTCHZ, enter

```
dc_shell> set_register_type -flip_flop FFX -latch LTCHZ
```

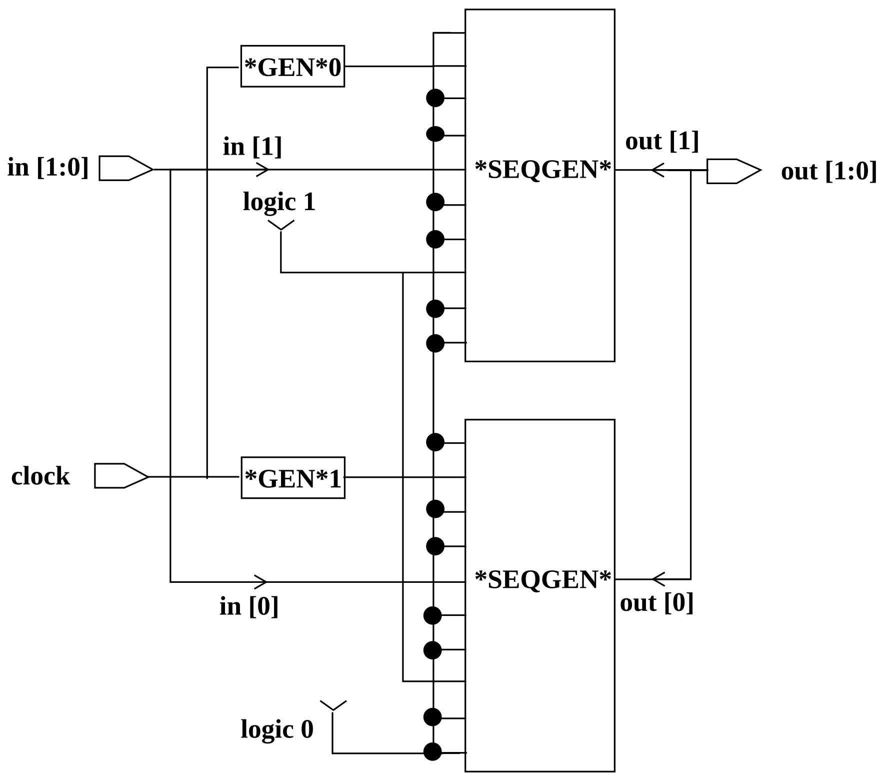
Mapping to Negative Edge Flip-Flops

During compilation, Design Compiler detects where negative-edge flip-flops can be used to optimize a design. You do not have to define any constraints to enable mapping to negative-edge flip-flops.

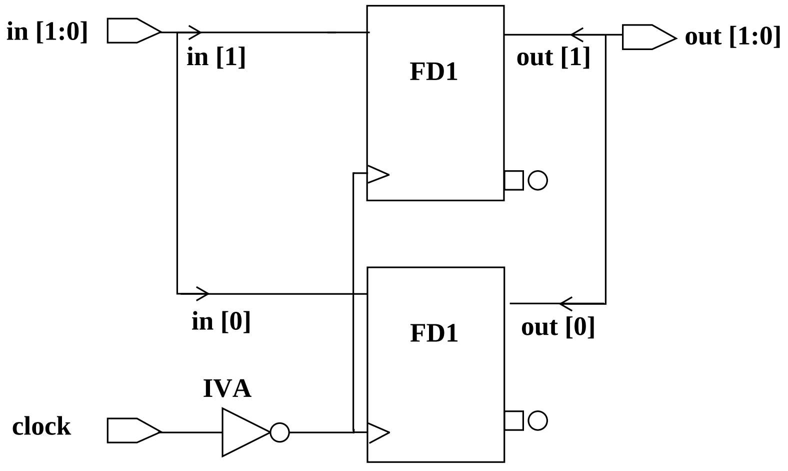
Example

For this Verilog design, HDL Compiler creates the logic shown following the code.

```
module test ( in, out, clock );
input [1:0] in;
input clock;
output [1:0] out;
  reg [1:0] out;
   always @ ( negedge clock ) begin
   out <= in;

  end
endmodule
```



The clock port feeds generic logic that, in turn, feeds the two generic flip-flops (SEQGEN). Given a library with both positive edge and negative edge-triggered flip-flops, Design Compiler creates this optimized negative edge-triggered design.



Reporting Register Types

You can see the current default register type specification for the design and for cells.

Reporting the Register Type Specifications for the Design

The `report_design` command lists the current default register type specifications.

```
dc_shell> report_design

****************************************
Report : design
Design : DESIGN
Version: 1997.01
Date   : Tues Jan 14 1997
****************************************
. . .
Flip-Flop Types:

    Default: FFX, FFXHP, FFXLP
```

Reporting the Register Type Specifications for Cells

The `report_cell all_registers()` command lists the current register type specifications for cells.

```
dc_shell> report_cell all_registers()

****************************************
Report : cell
Design : reg_type
Version: 1997.01
Date   : Tues Jan 14 1997
****************************************

Attributes:
  n - noncombinational
...

Cell      Reference      Library      Area      Attributes
----------------------------------------------------------
ffa       FFY            MY_LIB        9.00      1,n
ffb       FFY            MY_LIB        9.00      1,n
ffc       FFY            MY_LIB        9.00      1,n
ffd       FFY            MY_LIB        9.00      1,n
----------------------------------------------------------
Total 4 cells                         36.00

Flip-Flop Types:
 1 - Exact type FFY
```

Removing Register Type Specifications

To remove all register type specifications, use the `reset_design` command.

```
dc_shell> reset_design
Performing reset_design on design 'DESIGN'.
{DESIGN}
```

Defining a Signal for Unattached Slave Clocks

Design Compiler can automatically connect slave clock pins to a default signal when it translates or optimizes to flip-flops that have master- and slave-clock pins. The tool then defines the master-clock port as having the `clocked_on_also` signal type.

The `set_signal_type` command specifies the signal to attach to all unconnected slave clocks and sets the `signal_type` attribute to `clocked_on_also` on *port_name*. Then `compile` or `translate` connects unattached slave clocks to *port_name*.

The syntax is

```
set_signal_type "clocked_on_also" port_name
```

Example 1

To attach the port CLKBAR to all unattached slave clocks, enter

```
dc_shell> create_clock CLK -period 10
dc_shell> set_signal_type "clocked_on_also" CLKBAR
dc_shell> compile
```

Example 2

To specify a default signal for connecting through ports at various levels of a hierarchical design and to set the signal_type on each design port. Enter

```
dc_shell> read three_level.db
{"top","mid","bottom"}
dc_shell> current_design = bottom
dc_shell> set_signal_type "clocked_on_also" CLKBAR
dc_shell> current_design = mid
dc_shell> set_signal_type "clocked_on_also" CLKBAR
dc_shell> current_design = top
dc_shell> set_signal_type "clocked_on_also" CLKBAR
dc_shell> create_clock CLK -period 10
dc_shell> compile
```

Example 3

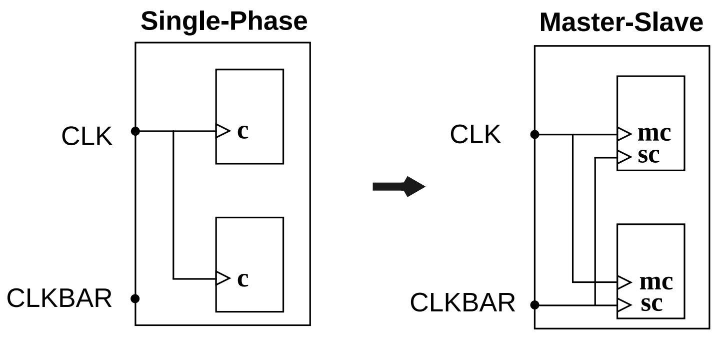
You can use `set_signal_type` when translating a design from single-phase flip-flops to master-slave flip-flops. The pins indicated by the `"clocked_on_also"` attribute are connected to the port with signal_type of `"clocked_on_also"`.

```
dc_shell> read single_phase.db
dc_shell> target_library = master_slave.db
dc_shell> set_signal_type "clocked_on_also" CLKBAR
dc_shell> translate
```

Example 4

You can work with multiple clocks by using the `set_signal_type` command. For example, enter

```
dc_shell> set_signal_type "clocked_on_also"
-associated_clock CK1 CK1B
```



Building a Balanced Buffer Tree

The `balance_buffer` command builds a balanced buffer tree on nets and drivers you specify. Balanced buffering does not automatically occur performed during compilation.

Use `balance_buffer` to build buffer trees within a single level of hierarchy for signals that drive numerous loads. Any logic within the fanout cone of the specified nets and drivers is affected.

The buffer tree is layered, and each stage uses the same buffer (inverter) and each gate of a given stage has the same fanout (one gate per stage can have a lower fanout than the others). You can specify the maximum number of stages.

When all loads have the same value, balanced buffering creates optimal buffer trees. When loads are not equal, balanced buffering creates a suitable buffer tree, but the tree might not be optimal.

An input port driving a net to be buffered by `balance_buffer` must have its drive value set. If you omit the value, no buffer tree is created, because the default value means that the input port has infinite drive.

Balanced buffering is constraints-driven. It fixes design rules first, then optimizes for timing and area.

The syntax is

```
balance_buffer [-verify] [-verify_hierarchically]
    [-verify_effort low | medium | high]
    [-from driver_pin_list] [-to load_pin_list]
    [-net net_list] [-depth depth_value]
```

`-verify`

Verifies functionality of the resulting circuit against the original circuit after the completion of the buffering. If you omit this option, no comparison is made.

`-verify_hierarchically`

Performs verification on a copy of each subdesign of the current_design. The endpoints to be verified are output ports in the top-level design, outputs of subdesigns, and register input pins. An endpoint depends functionally on sourcepoints in the same subdesign. The sourcepoints can be input ports in the top-level design, outputs of subdesign instances, inputs to the subdesign, or register outputs.

`-verify_effort low | medium | high`

Specifies the relative amount of CPU time for the verification phase. For more information, see “Controlling Verification Effort,” earlier in this chapter.

`-from driver_pin_list`

Lists one or more driver pins from which to build a buffer tree.

`-to load_pin_list`

Lists the load to be driven by the buffer tree that is being rebuilt. The driver of each load is inferred automatically.

`-net` `net_list`

Lists nets that need rebuffering. The (single) driver of each is inferred automatically.

`-depth` `depth_value`

Specifies the maximum depth of the new buffer tree. A smaller depth results in shorter execution times, but it might affect the quality of results. The default is 4, which is sufficient for most buffer trees.

Optimizing Across Boundaries

Boundary optimization is an option that can improve a hierarchical design by allowing the compile process to modify the port interface of lower-level designs. Normally, the compile process does not change any port functionality.

During normal compilation, constant and equal (opposite) properties are propagated down the hierarchy, even when boundary optimization is not enabled (`-boundary_optimization`). Two output signals that are functionally equal (opposite) can be detected during compile. The equal (opposite) properties can be propagated to the containing parent design. Consequently, the corresponding signals in the parent design can be optimized. For example, one signal can be unconnected, and all of its loads can be reconnected to the other, equal signal.

Similar optimizations can occur for opposite nets. Output nets in a subdesign that are functionally equal to constant 0 or 1 can be propagated to the parent design. Further optimization in the parent design can result. Verification deals with the propagation of these logical properties. However, hierarchical verification can use significant runtime in determining the properties required to prove that a design is a valid implementation of the original design specification. If some properties cannot be determined, a warning appears. These properties do not need to be computed for nonhierarchical verification, because both the original and implementation designs are flattened prior to verification.

When you specify `-boundary_optimization`, verification is automatically nonhierarchical.

If you enable boundary optimization during optimization, subdesign boundaries (ports) are maintained. Some logic optimization is possible across the boundaries, however, and the phase of the ports can be changed. For example, if the signal to a subdesign input port is always logic 1, you can simplify logic inside the subdesign after optimization.

Boundary optimization is an implicit `characterize -connections` command, where logical port connection information is attached to a subdesign. This connection information is set by the appropriate `set_equal`, `set_opposite`, `set_logic_one`, `set_logic_zero`, and `set_unconnected` commands. For more information, see *Design Compiler Reference Manual: Constraints and Timing*.

The function of subdesign ports can be changed by boundary optimization. For example, if one input of a 2-input adder subdesign is always set to logic 1 in the context of the current design, that subdesign is optimized as an incrementer after boundary optimization. If another design outside the scope of the current design references the original adder subdesign, it now references an incrementer.

Enable boundary optimization by using these commands:

`set_boundary_optimization`

Optimizes across specified hierarchical boundaries for one or more designs or subdesigns.

`compile -boundary_optimization`

Optimizes across all hierarchical boundaries in the current design.

Optimizing Across Specified Boundaries

The `set_boundary_optimization` command enables optimizing across hierarchical boundaries for one or more designs or subdesigns. This command sets the boundary optimization attribute on the specified designs or subdesigns.

The compile process uses this information to create smaller designs. Because this might change the function of the object, do not use the object in any other context.

If a cell with the specified name is found in the current design, the boundary_optimization attribute is set to the specified value in the cell. If no cell with the specified name is found in the current design, a reference is searched for. If a reference is found, the attribute is set for the reference. Otherwise, a design is searched for and the attribute is set for the design.

Occasionally cells use an input signal only in its complemented form. In this case, consider inverting the signal and its ports for optimal results. The `set_boundary_optimization` command considers these optimizations. When this occurs, port names change according to the `port_complement_naming_style` variable setting.

The syntax is

```
set_boundary_optimization subdesign_list true | false
```

subdesign_list

Lists one or more subdesigns (cells) in the current design. If you specify more than one object, enclose the names in quotation marks or braces ({ }).

`true | false`

Enables or disables boundary optimization for the subdesigns in the subdesign list.

- true (the default) enables boundary optimization.
- false disables boundary optimization.

To remove the boundary_optimization attribute, use the `remove_attribute` command.

Optimizing Across All Boundaries

The `compile -boundary_optimization` command optimizes across all hierarchical boundaries in the current design. This command propagates constants, unconnects, and equal or opposite information across hierarchical boundaries.

If there are constraints on the design, `compile -boundary_optimization` inverts hierarchical interface signals (when appropriate) if both of the following criteria are met:

- The complement of the net is already easily available.
- Inverting the net improves the cost function.

If these conditions are satisfied, the entire net is complemented and the affected boundary ports are renamed to indicate that they now represent the complement of the original signal. If a port undergoes, two of these transformations, its name reverts to the original name with no indication that the port was ever inverted.

The affected ports are renamed according to the dc_shell variable `port_complement_naming_style`. The default naming style is %s_BAR, where %s represents the original name of the port. If the new name conflicts with an existing port name in the design, an integer is appended to the new name to produce a unique name.

Enabling Critical Path Resynthesis

The `compile` command `-map_effort high` option implements an optimization strategy called critical path resynthesis. Critical path resynthesis seeks to resolve timing violations on critical paths by resynthesizing the logic on the paths. The goal is to create a small partition of cells on and around part of the current critical path, then restructure and remap the new partition. The new partition is then accepted or rejected, based on the Design Compiler cost vector.

(The critical path is the path in the design that limits the clock speed—the path with the minimum amount of slack. You can list the cells on the critical path by using the `report_timing` command.)

The `-map_effort high` option is CPU-intensive.

Critical path resynthesis requires a DC Expert license.

Optimizing for Area

The area optimization process minimizes the area your design uses. The process tries to improve area, if that can be done without degrading delay cost.

Area optimization requires that you set an area constraint, using the `set_max_area` command.

By default, area optimization does not create a new timing violation or worsen an existing timing violation to gain an improvement in area. If Design Compiler can identify a change to a path with a timing violation, it makes the change only if it can improve area without worsening the timing violation.

You can choose to enable area optimization at the risk of worsening timing violations on some paths and creating some new timing violations, as long as the violation on the most critical path in each path group is not affected. Assuming you are willing to take this risk, you can direct Design Compiler to enable area optimizations on all paths, using the `set_max_area` command with its `-ignore_tns` option. (TNS means total negative slack, the sum of the delay violations of all violating endpoints.) The command line is

```
dc_shell> set_max_area 0 -ignore_tns
```

(The full command syntax is described in the *Design Compiler Reference Manual: Constraints and Timing*.)

To constrain a design for area only, use the following commands:

```
dc_shell> remove_constraint -all
dc_shell> set_max_area 0
```

Achieving the smallest design can require changes in optimization strategy or rewriting the HDL code. Some strategies that can help achieve small designs are

- Use `set_structure` true `-boolean` true `-timing` false.
- Ungroup all or part of the hierarchy. Take care not to ungroup regular structures such as adders.

Optimizing for Delay

Design Compiler optimizes the timing of your design based on the delay constraints you specify. Constraints affecting delay include clocks, input and output delays, external loads, input driving cells, operating conditions, and wire load tables.

The following strategies can help achieve a faster design:

- Ungroup all or part of the hierarchy. This can give optimization more freedom to change logic that previously spanned a hierarchical boundary.
- Turn on boundary optimization for modules you do not want to ungroup.
- Turn on flattening where possible.
- Use multiple passes of `compile`. Always ensure that the map effort is set to medium or high, because low effort can give suboptimal delay results.
- Specify a critical range, so that Design Compiler optimizes not only the critical path but paths within that range of the critical path as well.

- In the design exploration phase, you might want to give delay a higher priority than design rules, using the `set_cost_priority -delay` command.

Removing Hierarchy

Design Compiler provides several commands for flattening design hierarchy:

`set_ungroup`

Ungroups one or more designs, subdesigns (cells), or references during compilation (see the *Design Compiler User Guide*).

`ungroup`

Ungroups a design or reference manually (see the *Design Compiler User Guide*).

`compile -ungroup_all`

Ungroups all subdesigns during optimization (described earlier in this chapter).

Running Incremental Compilation for Post-Placement Optimization

The goal of incremental optimization after layout is to make changes that fix design rule violations or timing violations with minimal impact on the layout. Making widespread changes after layout can introduce other design rule violations. Restrict the optimization to performing only transformations that the layout tool can safely implement.

In all post-placement optimization, it is important to understand the capabilities of the layout tool for making changes to an existing layout. Optimization strategies take this into account and limit the magnitude of the changes to a level that can accommodate the layout tool.

The `compile` command automatically removes back-annotation data before modifying a circuit. Design Compiler removes the information to avoid inconsistencies between estimated and actual value. A warning message appears when back-annotated data is removed.

When Design Compiler removes back-annotation data, it reverts to using wire load model-based estimates. These estimates can be less accurate than the original information, so it is important to preserve as much back-annotation data as possible during an incremental post-placement compilation.

You can preserve back-annotation during `compile -incremental` by setting the `dont_touch` attribute on nets, cells, references, or designs that do not need to be reoptimized. For example, if the design rule violations appeared in only one small block of the design, preserve the back-annotation for the other blocks by using the `dont_touch` attribute. Only the one block will be reoptimized.

Use this technique only for minor modifications. The other techniques available are in-place optimization using the `reoptimize_design` command described in the *Floorplan Manager User Guide*. The `reoptimize_design` command requires a DC Ultra license or Floorplan Manager license.

Propagating Constants

The compilation process entails a series of optimizations including constant propagation. If you want to perform constant propagation only to save area while retaining the basic structure of the remaining portions of the design, you can use the `simplify_constants` command to perform the constant propagation optimization separately.

By default, the `simplify_constants` command uses constants to simplify logic within the current design. However, with use of the command's `-boundary_optimization` option, constant signals are used to allow logic simplification across subdesign boundaries.

The syntax is

```
simplify_constants [ -boundary_optimization ]
     [ -verify ]
     [ -verify_effort low | medium | high ]
     [ -verify_hierarchically ]
```

The `simplify_constants` command optimizes logic `0`, logic `1`, and unconnected signals. By default, the command does not propagate this information into lower-level designs unless the `-boundary_optimization` option is used. The `-verify`, `-verify_effort`, and `-verify_hierarchically` options are equivalent in behavior to the compile options of the same names.

For complete command syntax description and details, see the man page.

You can use the following commands to define which signals are constant:

- set_logic_one
- set_logic_zero
- set_unconnected

The `simplify_constants` command propagates the constant information forward. It propagates information on unconnected signals back through the design.

4

Controlling Logic-Level and Gate-Level Optimization

The goal of logic-level optimization is to improve a design’s logic structure. Design Compiler strives to reduce the number of product terms. As a first-order approximation, decreasing the number of product terms relates to both area and delay reduction.

The optimization of logic equations does not affect a particular part of a function. Rather, it has a global effect on the overall area or speed characteristics of a design. Therefore, the logic optimization strategy you choose affects the optimized design.

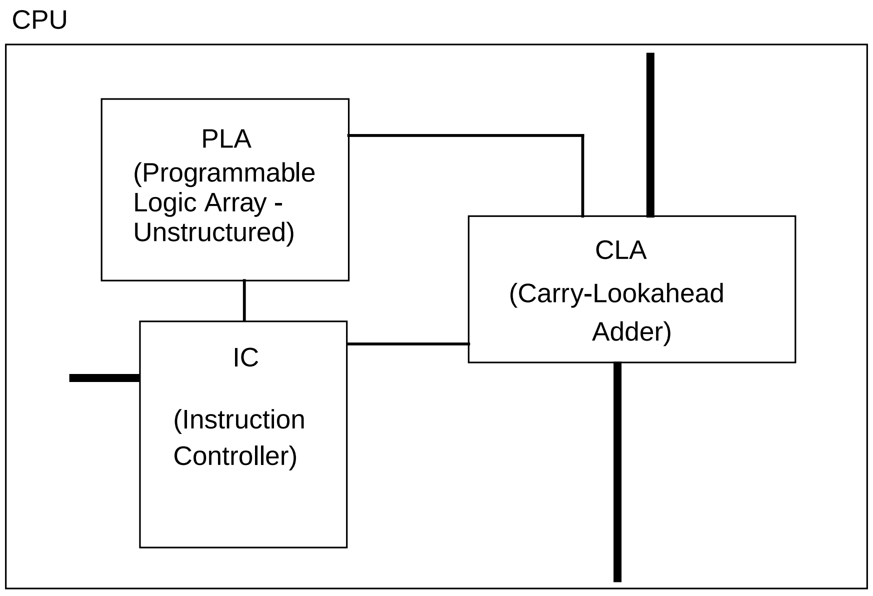
Logic optimization has two components: flattening, which removes intermediate variables and uses Boolean distributive laws to remove all parentheses, and structuring, which adds intermediate variables and logic structure.

This chapter contains the following sections:

Basic Concepts

Many modern digital designs are hierarchical, where the main (top-level) design is composed of interconnected blocks of logic. Some blocks fall along the data path; others are part of the control logic. Some blocks are highly structured (perhaps a carry-lookahead adder), and other pieces of the hierarchy consist primarily of random logic (such as an instruction decoder). Figure 4-1 shows a hierarchical design.

Figure 4-1 A Hierarchical Design



Because the logic in each block in the sample is different, each hierarchical piece requires different treatment by Design Compiler.

- For structured blocks, preserve and build on the existing structure.
- For unstructured blocks (random and control logic), remove redundant gates (logic terms) and have Design Compiler improve the structure.

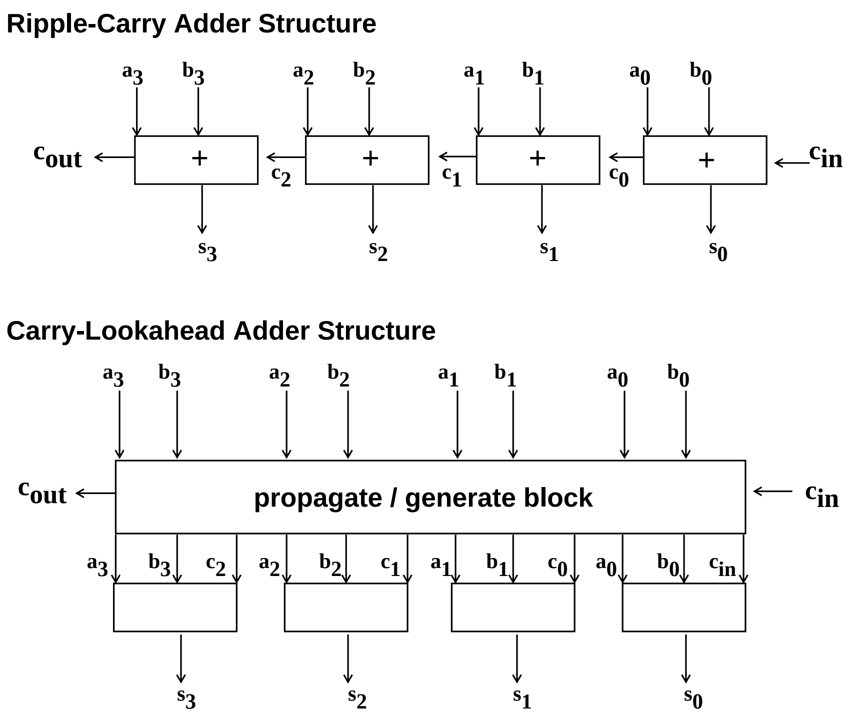
Structure of Designs

Structure exists at both the logic level and the gate level. The global organization of the design defines the logic structure of that design. The logic-level structure is not technology-specific and is represented by a set of Boolean equations.

The local implementation of a design defines its gate structure. For example, a variety of gate structures can be used to implement a 4-bit carry-lookahead adder. A design described at this level is technology-specific and is represented by a gate-level netlist.

Figure 4-2 shows two radically different logic structures that perform the same function. Both logic structures add two 4-bit quantities with carry-in and carry-out (a 4-bit-slice adder).

Figure 4-2 Two Different Logic Structures for a 4-Bit Adder

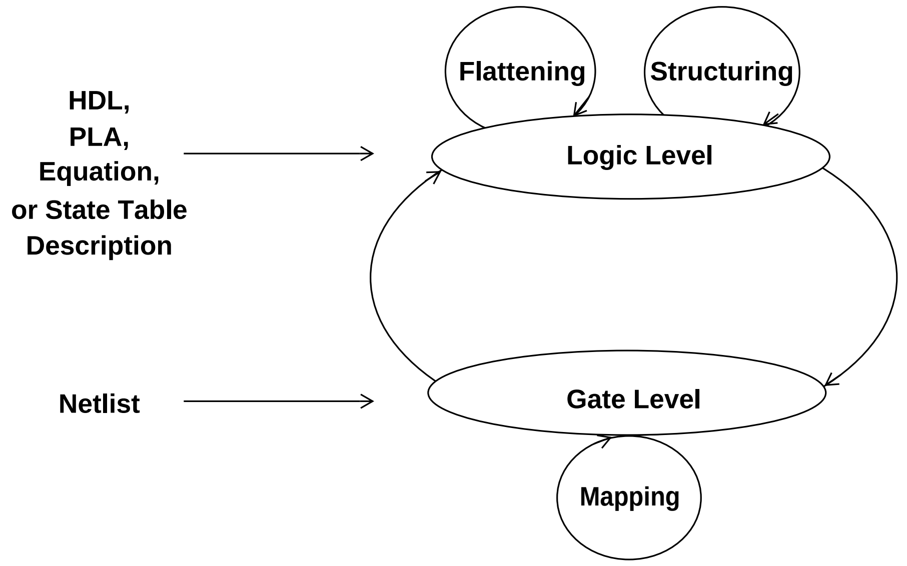

Ripple-Carry Adder Structure
a_3 b_3 a_2 b_2 a_1 b_1 a_0 b_0
c_{out}
+
c_2
c_1
c_0
c_{in}
s_3 s_2 s_1 s_0
Carry-Lookahead Adder Structure
a_3 b_3 a_2 b_2 a_1 b_1 a_0 b_0
c_{out}
propagate / generate block
c_{in}
a_3 b_3 c_2 a_2 b_2 c_1 a_1 b_1 c_0 a_0 b_0 c_{in}
s_3 s_2 s_1 s_0

How Design Compiler Optimizes Your Design

Design Compiler works at two levels: the logic level and the gate level. Optimization occurs at both levels but takes a different form in each.

Figure 4-3 shows the internal operations during optimization.

Figure 4-3 Compilation Operations and Representation Levels



An HDL, programmable logic array (PLA), equation, or state table description is read in and converted to a logic-level description. First, Design Compiler applies logic optimization (flattening and structuring). Flattening is off by default and is applied only if enabled. The resulting structure is mapped to gates.

Alternatively, a netlist description is read in as a gate-level equivalent. Design Compiler extracts the logic equations from it. Logic optimization is then applied, and the resulting equations are mapped back into gates.

In a hierarchical design, different levels of the hierarchy can be read from different sources. The design can be represented as a mix of gate-level and logic-level constructs.

Design Compiler moves the design between the logic-level and gate-level representations to take advantage of the different optimization techniques. The optimization cycle continues until a satisfactory gate-level implementation is achieved or until no further improvements are possible with the current design.

Controlling Logic-Level Optimization

The `set_flatten` and `set_structure` commands control logic-level optimization. These commands set the corresponding design compilation attributes for flattening and structuring. By default, flattening is off and structuring is on.

Controlling Gate-Level Optimization

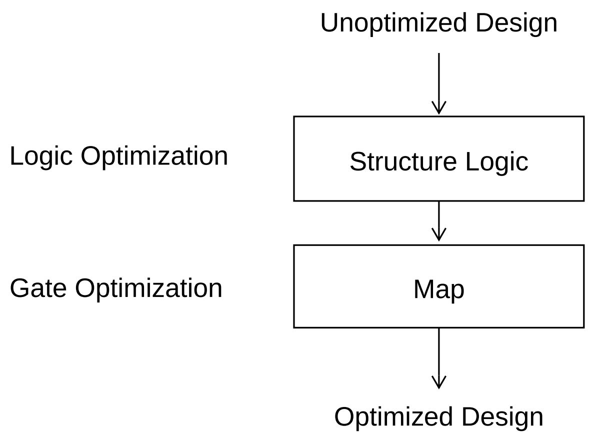
Mapping is enabled by default. The `-no_map` option of the `compile` command controls gate-level optimization (mapping).

Given your optimization objectives (constraints), the mapping process obtains the best (fastest, smallest) mapping from a design’s internal logic representation from the available gates of the target technology library.

The `compile` command and its options are described in Chapter 3, “Optimizing Designs.”

Figure 4-4 shows the default operation of the `compile` command: it first performs logic optimization then performs gate-level optimization.

Figure 4-4   Default Flow of Optimization



Setting Optimization Switches in a Hierarchical Design

In a hierarchical design, set the `flatten` or `structure` attributes on a design-by-design basis; do not set them globally. These commands provide control over the fine-tuning of every design in the hierarchy.

`set_flatten`

Controls whether a design or subdesign is flattened; the effort level used; and the minimization strategy, including phase assignment. By default, a design is not flattened (`set_flatten` false) during compilation. All the flattening options—effort level, minimization strategies, and phase assignment—are disabled unless flattening is enabled for a design (`set_flatten` true).

`set_structure`

Controls whether a design or subdesign is structured and whether timing-driven structuring or Boolean optimization is used. By default, a design is structured (`set_structure` true) during compilation. Timing-driven structuring is on by default (`set_structure` true `-timing` true), but Boolean optimization is off by default (`set_structure` true `-boolean` false). The structuring options—timing-driven structuring and Boolean optimization—are disabled if structuring is disabled (`set_structure` false).

Optimization Switch Default Settings

Table 4-1 lists the default settings for the `structure` and `flatten` attributes.

Table 4-1 Structure and Flatten Attributes Default Settings

Attribute	Value
`flatten`	false
`structure`	true
`structure_boolean`	false
`structure_timing`	true
`-map_effort`	medium

Table 4-2 lists the default settings when the `flatten` attribute is set to true.

Table 4-2 Default Settings When the Flatten Attribute Is Set to true

Attribute	Value
`flatten`	true
`flatten_effort`	low
`flatten_minimize`	single
`flatten_phase`	false

Table 4-2 Default Settings When the Flatten Attribute Is Set to true

Attribute	Value
structure	true
structure_boolean	false
structure_timing	true

Example

Refer to the previous Figure 4-1's hierarchical design when reviewing the following set of examples.

The following command sequence causes Design Compiler to flatten and structure the CPU design. Structuring is on by default, and flattening is enabled for the current design only. Default options apply to the subdesigns, PLA, IC, and CLA.

```
dc_shell> current_design = CPU
dc_shell> set_flatten true
dc_shell> compile
```

The following command sequence flattens and structures the PLA and IC subdesigns. By default, the current design CPU and the CLA subdesign are only structured because structuring is on by default and flattening is enabled only for PLA and IC.

```
dc_shell> current_design = CPU
dc_shell> set_flatten true -design {PLA, IC}
dc_shell> compile
```

The following command sequence uses the find command with the `set_structure` command to disable structuring on all the designs in memory, including CPU, PLA, IC, CLA, and any subdesigns.

```
dc_shell> current_design = CPU
dc_shell> set_structure false find (design "*")
dc_shell> compile
```

Checking the Optimization Strategy Settings

The `report_compile_options` command displays optimization strategy attributes in effect for the current design.

Using Flattening

Flattening is an optional logic optimization step that removes all intermediate variables and uses Boolean distributive laws to remove all parentheses. Thus, flattening removes all logic structure from a design. Removing poor intermediate variables enables Design Compiler to choose more-efficient subfunctions.

Note:

> Flattening is often mistaken for flattening the hierarchy of a design. Flattening the hierarchy is called “ungrouping the design” and is independent of the `flatten` option of `compile` command. A flattened design implies a design optimized with the `flatten` option turned on; a flattened design does not mean a design devoid of hierarchy.

Flattening is off by default because timing-driven structuring (on by default) can consider the critical paths. Some designs might benefit from flattening applied with or without structuring. You can also obtain a two-level (sum of products) equation representation of a design using flattening only, with no structuring or mapping.

The result of flattening is a two-level, sum-of-products form.

Example

Before flattening

```
out = t1 t2
     t1 = a + b(c + f)
     t2 = d + e’
```

After flattening

```
     out = ad + bcd + bdf + ae’ bce’ + be’f
```

The sum-of-products, two-level form can be used in one of two ways:

- Mapped to a PLA-like structure
- Input to the structuring phase to try to find a optimal restructuring of the logic equation

Flattening Considerations

Although the flattened form is often considerably larger than the structured implementation, it can be faster, because the two-level form generally has fewer levels of logic between inputs and outputs. If you want to keep your design in a two-level form, do not structure after flattening.

Flattening is a good way to eliminate bad logic structure, but if used indiscriminately, it can also eliminate good logic structure. If you flatten a carry-lookahead adder, for example, Design Compiler might not be able to restructure that design later.

In general, you can flatten random control logic, because automatic structuring usually improves on manual structuring. Do not flatten regular or highly structured designs, such as adders and ALUs designed with an explicit structure.

Not all designs can be flattened.

- Small designs of 10 inputs or less can always be flattened.
- Large designs that have 20 or more inputs are often impossible to flatten.

Predicting the Design Compiler ability to flatten larger designs is difficult, but you can use the following guidelines:

- Designs whose outputs are consistently true or consistently false for most of the possible input patterns are generally easier to flatten.
- Designs with many XOR and multiplexer gates, such as adders and ALUs, are generally difficult to flatten. For example,
  - An N-bit XOR function produces 2n-1 terms.
  - A 2-input XOR gate can be represented in two-level sum-of-products form as Z = A’B or AB’.
  - A 20-input XOR gate, when flattened, has 219 (524,288) terms.

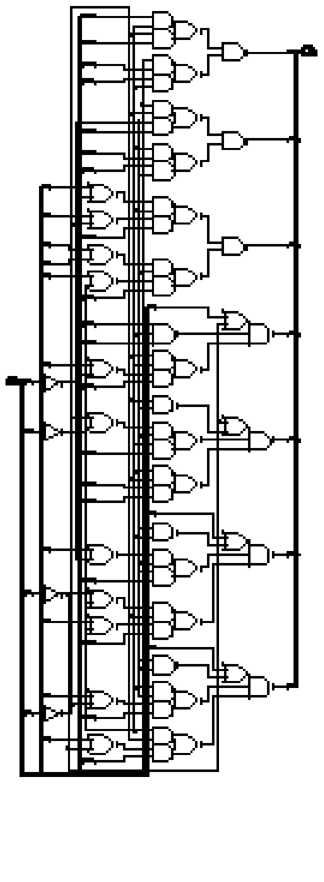
Because of this term explosion, flattening functions with many XORs or multiplexers is difficult and undesirable. Attempting to flatten large ALUs, adders, or multipliers — which contain many XORs — or designs containing such components can have a long runtime and can often exceed the memory available on your workstation. Flattening does not complete because most libraries usually do not have cells with more than five inputs (5-input AND gates, for example), that are required to achieve a two-level, sum-of-products form. If flattening does not complete, Design Compiler issues a message and proceeds to the next step.

Use flattening judiciously; you cannot read in a large netlist and expect the synthesis tool to execute the flatten operation.

Effect of Flattening on Speed

The result of flattening is a two-level, sum-of-products form equivalent to a PLA. If you choose, you can flatten a design and subsequently map the design to gates without structuring—effectively accomplishing PLA minimization, then mapping. The gate-level representation of a flattened design can be fast, because there are few stages between input and output (in the extreme, only an AND plane and an OR plane). Figure 4-5 shows an example of mapping a flattened design without structuring. Note that there are only three levels of logic from input to output and that the result of this technique is a tall, narrow design.

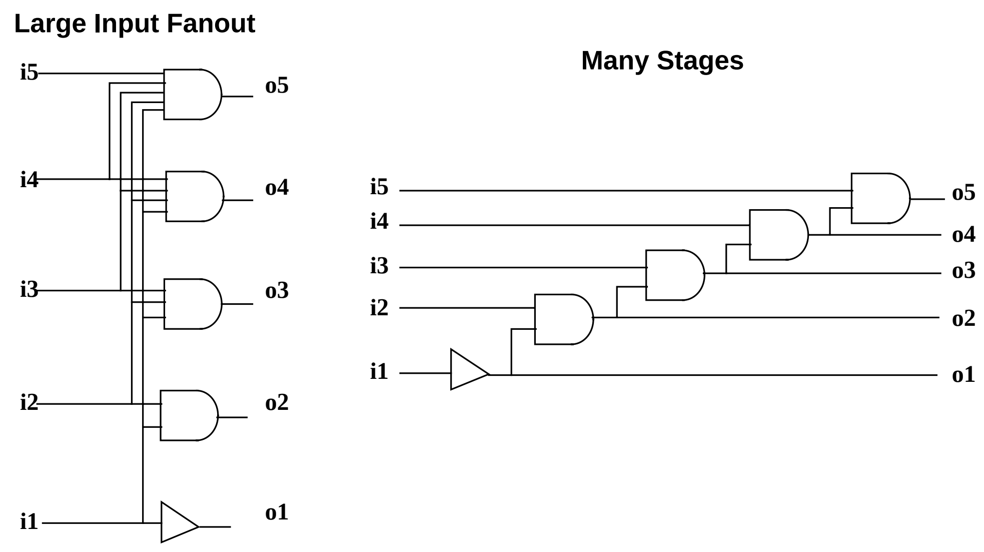
Figure 4-5 Compiling With Flattening and Without Structuring



Flattening does not always yield the fastest design. In the case of a design with few inputs and numerous outputs, flattening can mean that some inputs fan out to many outputs, placing a large load on the inputs.

Figure 4-6 shows the tradeoff between flattening and no structuring for a circuit with a large input fanout versus flattening and structuring for a circuit with many stages. Although Design Compiler has built-in buffering algorithms, heavily loaded inputs might still slow the critical path unacceptably.

Figure 4-6 Tradeoff: Flattening and No Structuring Versus Flattening and Structuring



Enabling and Controlling Flattening

The `set_flatten` command controls whether a design or subdesign is flattened, the effort level used, and the minimization strategy (including phase assignment).

By default, compilation does not flatten a design. To flatten a design or a subdesign during optimization, set the `flatten` attribute to true on that design using the `set_flatten` command.

When set on a hierarchical design, the `flatten` attribute by default is set only on the current design and not set on any of its subdesigns. The exception is when you specify a list of designs, using the `-design` option.

The syntax is

```
set_flatten true | false
    [-design design_list]
    [-effort low | medium | high]
    [-minimize single_output|multiple_output| none]
    [-phase true | false]
```

`true | false`

Enables or disables flattening.

- true causes flattening for the designs named in design_list.
- false is the default.

The default settings when the `flatten` attribute is set to true are

Attribute	**Default setting**
flatten	true
flatten_effort	low
flatten_minimize	single
flatten_phase	false

`-design` *design_list*

Applies the flattening options to the designs listed in design_list. If you omit this option, by default the set_flatten command applies only to the current design, not to its subdesign.

`-effort low | medium | high`

Sets the flattening effort for the designs listed in design_list. If you omit this option and `set_flatten` is set to true, the default effort is low. Use `-effort` only if `set_flatten` is set to true. The values are described in detail in “Setting the Flatten Effort” later in this chapter.

`-minimize single_output | multiple_output | none`

Sets the minimization strategy for the designs listed in design_list. Use `-minimize` only if `set_flatten` is set to true. The values are described in detail in “Enabling and Controlling Minimization” later in this chapter.

`-phase true | false`

Enables or disables phase assignment. Use -phase only if `set_flatten` is set to true.

- true enables phase assignment for the designs listed in design_list.
- false disables phase assignment for the designs listed in design_list (the default).

See “Enabling Phase Inversion” later in this chapter.

Example

```
dc_shell> current_design = TEST
dc_shell> set_flatten true -design TEST
dc_shell> compile
```

Removing the Flatten Attribute

To remove the `flatten` attribute (change the value back to false), use the `set_flatten false` command. For example, enter

```
dc_shell> set_flatten false -design TEST
```

Reporting Flatten Attributes

The `report_compile_options` command lists the flatten attributes in effect for the current design.

Setting the Flatten Effort

Design Compiler has a built-in estimation algorithm that determines the number of product terms necessary to flatten a design. If the number of terms exceeds a certain value, the flattening process stops. When a design cannot be flattened easily, the flattening process flattens the easy parts of the design, then terminates with a status message. A design can be partially flattened and still be functionally correct.

Use the `-effort` option when you want to flatten a design for speed purposes. If Design Compiler fails to flatten with low effort, try a higher setting. Design Compiler probably cannot structure a design effectively after it has been flattened with medium or high effort.

Control the amount of workstation CPU time and memory devoted to flattening by using the `-effort` option with the low, medium, or high value.

`set_flatten -effort low`

> Is the default when flattening is enabled. Typically, it is appropriate for flattening most designs. To increase the degree of flattening, specify a higher effort level.

`set_flatten -effort medium`

> Generally flattens a design beyond the point where Design Compiler can restructure it effectively.

`set_flatten -effort high`

> Causes the flattening process to proceed until the design is completely flattened or until your workstation runs out of memory. Because this effort might never terminate for some designs, use it only if you are sure that the design can be flattened.

Enabling and Controlling Minimization

The minimization step reduces the number and size of the product (AND) and sum (OR) terms of the logic equations. Karnaugh maps and the Quine-McCluskey method are two popular techniques for manually performing this optimization.

For PLA-format designs, don’t care conditions are used during optimization to minimize the resulting logic.

After flattening, Design Compiler can minimize the resulting equations. The `-minimize` option is available only if `set_flatten` is set to true. You can think of the minimization that occurs at this stage as resembling Karnaugh map reduction.

In addition to no minimization, two types of minimization are available:

- Single output (the default when flattening is enabled)
- Multiple output

Control minimization by using the `set_flatten -minimize` option with the single_output, multiple_output, or none value.

`set_flatten -minimize single_output`

Is the default. It causes Design Compiler to minimize the equations for each output individually. This results in the smallest implementation for each output, but the design as a whole might suffer because product (AND) terms are not shared well between outputs. You can abbreviate the single_output value as s.

`set_flatten -minimize multiple_output`

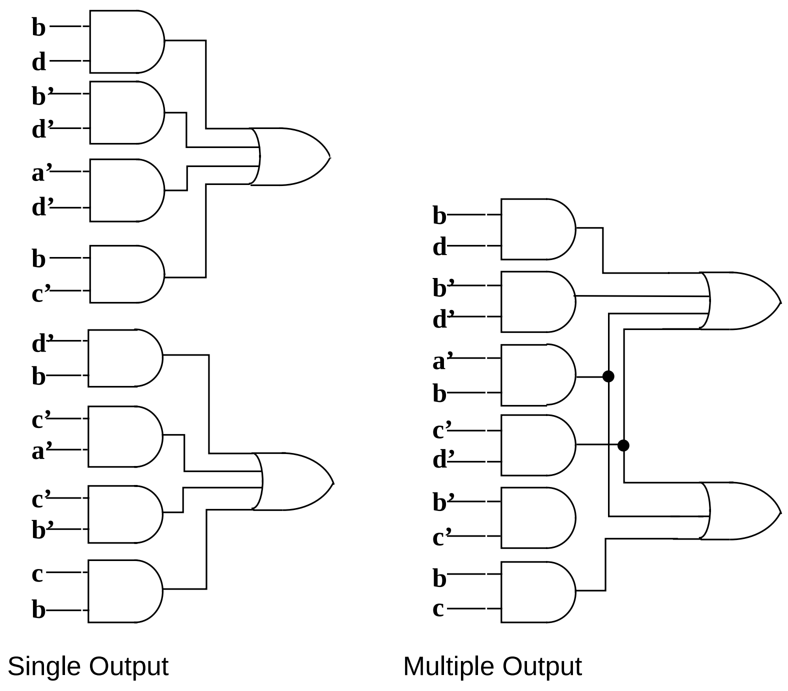
Causes Design Compiler to minimize all logic for a design and to share as many product (AND) terms between outputs as possible. This strategy is beneficial for ROM-like structures in which most outputs are a function of all inputs.The multiple_output value can be abbreviated as m.

`set_flatten -minimize none`

Causes Design Compiler not to perform minimization.The none value can be abbreviated as n.

Figure 4-7 shows the effects of single-output and multiple-output minimization.

Figure 4-7 Single-Output Minimization Versus Multiple-Output Minimization



Enabling Phase Inversion

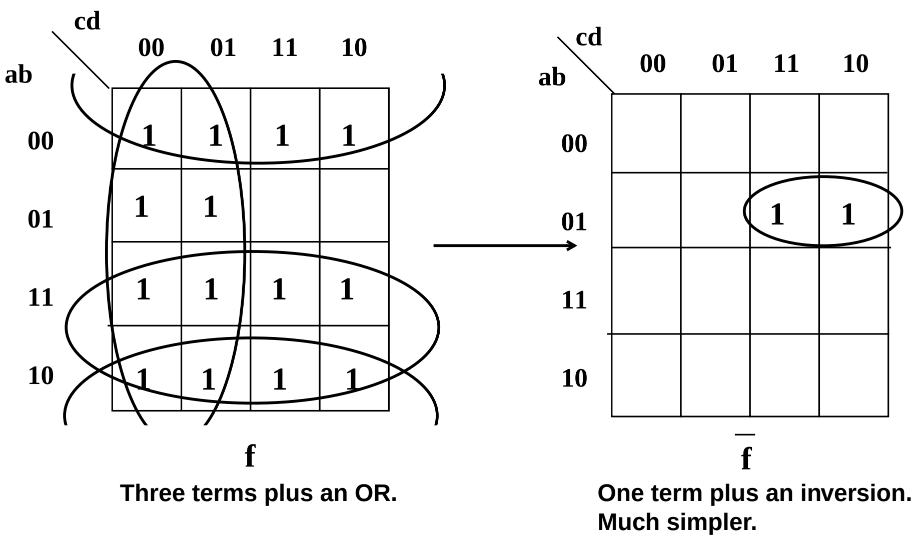
In addition to the primary minimization technique, you can use phase assignment. Phase assignment compares possible implementations of each logic equation for both the original equation and its complemented (negated) form.

You have the option of inverting the phase of the signals in the design. Inverting a Karnaugh map and attaching an inverter on the output (Figure 4-8) occasionally produces remarkable efficiencies in the number of product terms. However, using this technique tends to provide little correlation at the gate level for major designs.

Enable phase inversion by enabling `set_flatten` and using the -phase true value.

```
set_flatten true -phase true
```

Figure 4-8 Phase Inversion



Using Structuring

Structuring is a logic optimization step that adds intermediate variables and logic structure to a design. During structuring, Design Compiler searches for subfunctions that can be factored out, then evaluates these factors based on the size of the factor and the number of times the factor appears in the design. The subfunctions that most reduce the logic are turned into intermediate variables and factored out of the design equations. Design Compiler structures a design during compilation by default.

Structuring provides a powerful way to improve the logic structure of your designs. However, the optimization algorithms used to implement this step are not guaranteed to find the best logic structure for your designs. These algorithms perform well for random control logic but less well for complex designs, such as adders and ALUs.

Structuring is incremental. When you structure a design that already has structure (intermediate variables), Design Compiler builds on the existing logic structure.

Design Compiler also provides these structuring capabilities, which are described later in this chapter:

- Timing-driven structuring

  Timing-driven structuring only is the default. Timing-driven structuring with no is the recommended first strategy to apply to any design, structured or unstructured. Timing-driven structuring can improve critical paths without changing the structures on the noncritical paths.

- Boolean optimization structuring

  Design Compiler provides two sets of algorithms for Boolean structuring. The differences and how to determine the set to use are described later in this chapter.

Figure 4-9 shows the results of structuring a simple set of equations. In this example, the subfunction t0 is isolated as an intermediate variable and then factored out of the remainder of the design.

Figure 4-9 Results of Structuring

```
Before Structuring         After Structuring

f0 = a b + a c             f0 = a t0
f1 = b + c + d             f1 = d + t0
f2 = b' c' e               f2 = t0' e
                           t0 = b + c
```

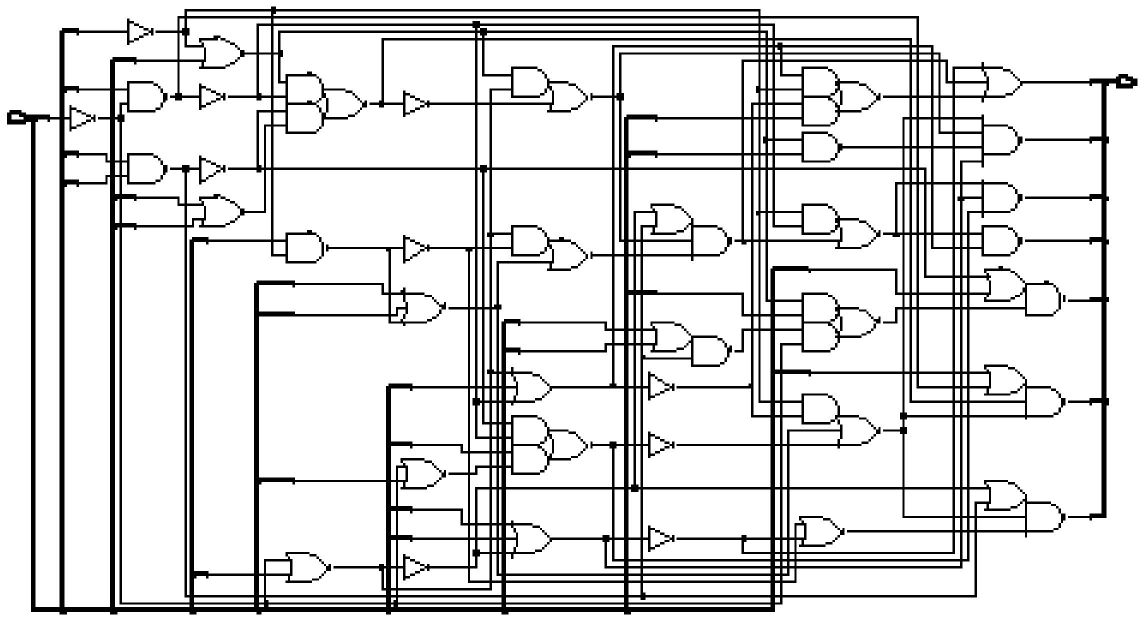
Structuring Considerations

Keep in mind these facts about structuring.

The result of structuring is that terms are shared. This produces an area-efficient design but might have a negative effect on the delay if there are no delay constraints on the design or if timing-driven structuring is turned off (`set_structure -timing` false). In the absence of delay constraints or timing-driven structuring, structuring might produce additional levels of logic on the critical paths.

Figure 4-10 shows the results of structuring with no delay constraints on the same design shown earlier in Figure 4-5. This implementation is substantially more area-efficient, but there are now eight levels of logic along the critical path.

Figure 4-10 Compiling With Structuring



Controlling Structuring

The `set_structure` command controls whether a design or subdesign is structured and whether timing or Boolean optimizations are used.

The `set_structure` command, by default, applies only to the current design and not to its subdesigns, unless you specify a list of designs, using the `-design` option.

The default settings for `set_structure` are

Attribute	**Default settings**
structure	true
structure_boolean	false
structure_timing	true

The syntax is

```
set_structure true | false
     [-design design_list]
     [-boolean true | false]
     [-boolean_effort low | medium | high]
     [-timing true | false]
```

`true | false`

Sets the value for the structure attribute.

- true (the default) causes structuring for the designs listed in *design_list*. When true, `structure_timing` is also true.
- false turns off structuring; both `structure_boolean` and `structure_timing` are ignored.

`-design design_list`

Applies the structuring options to the designs listed in *design_list*. If you omit this option, the `set_structure` command applies only to the current design (the default), not to its subdesigns.

`-boolean true | false`

The value to which the `structure_boolean` attribute is to be set. If `set_structure` is false, this option is ignored.

- true applies Boolean optimization for the designs listed in *design_list* if `set_structure` is also true.
- false (the default) disables Boolean optimization for the designs in *design_list*.

`-boolean_effort low | medium | high`

Sets the Boolean structuring effort for the designs in the design list. If you omit this option and `set_structure` is set to true, the default effort is low. Use `-boolean_effort` only if `set_structure` is set to true. The values are described in detail in "Setting the Boolean Structuring Effort" later in this chapter.

This option has no effect if the `compile_new_boolean_structure` variable is false (the default).

`-timing true | false`

Sets the value for the `structure_timing` attribute. If `set_structure` is false, this option is ignored.

- true enables timing-driven structuring for the designs listed in the design list and is the default if `set_structure` is true.
- false disables timing-driven structuring for the designs listed in the design list.

Turning Off Structuring

To turn structuring off, set the structure attribute on the design to false.

```
dc_shell> set_structure false -design TEST
dc_shell> compile
```

Reporting Structuring Attributes

Use the `report_compile_options` command to see the structuring attributes in effect for the current design.

Using Timing-Driven Structuring

Timing-driven structuring considers a design’s timing (delay) constraints during local and global structuring and improves critical paths as necessary. Timing-driven structuring is on by default when `set_structure` is set to true (the default). Timing-driven structuring occurs as part of the structuring step during compilation.

When you use timing-driven structuring, you must define accurate timing and clock constraints. If you define a maximum delay of zero, the compiled circuit can be unacceptably large, because timing-driven structuring duplicates logic paths, as necessary, to minimize delay.

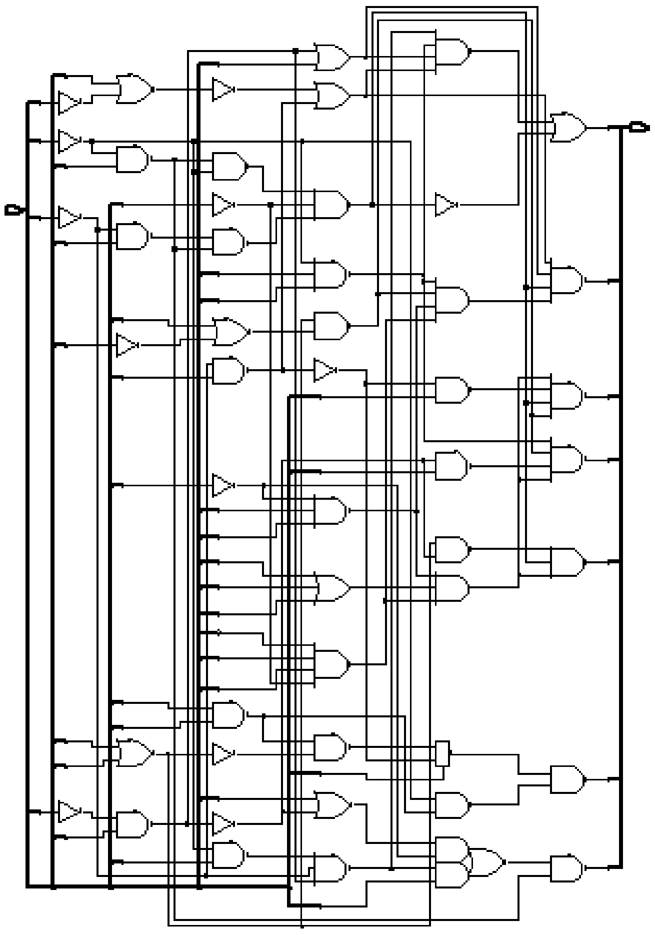
To turn timing-driven structuring off, set the `-timing` option to false before compilation.

```
dc_shell> set_structure -timing false
dc_shell> compile
```

If timing-driven structuring is off or if no timing constraints are on, structuring takes place only to reduce terms such as area.

Figure 4-11 shows the result of timing-driven structuring, with delay constraints, on the design in Figure 4-10.

Figure 4-11 Compiling With Timing-Driven Structuring



Using Boolean Optimization Structuring

Boolean optimization structuring uses Boolean algebra to capture don't care information and reduce circuit area. For example, a * !a = 0, a + a = a, and a + !a = 1.

You can specify don't care information for a design in the existing finite state machine, PLA, or equation input format. You can also specify don't care information in the HDL source code.

Boolean optimization is executed as part of the structuring step during compilation. It is off by default. To turn on Boolean optimization, set the `-boolean` option to true before compilation. For example, enter

```
dc_shell> set_structure -boolean true -boolean_effort medium
dc_shell> compile
```

Design Compiler provides two sets of algorithms for Boolean structuring. The algorithms used are determined by the setting of the `compile_new_boolean_structure` variable.

Algorithms Before v1997.01

The algorithms used before v1997.01 skip the entire optimization step when the memory requirements exceed the memory available.

If the `compile_new_boolean_structure` variable is set to false (the default), Design Compiler uses the Boolean structuring algorithms in effect before v1997.01.

Algorithms Introduced in v1997.01

The algorithms introduced in v1997.01 use automatic test-pattern generation (ATPG) techniques to manipulate logic networks. This Boolean structuring optimizes the circuit by identifying high fanout nodes and attempting to remove them. The algorithms remove them by adding other connections that make the original nodes redundant.

These algorithms are more scalable and memory-efficient than the algorithms used before v1997.01; thus, you can run them on large designs.

Failure to optimize a particular node does not cancel the entire process, because the effect is localized to that one portion of the design.

To enable the new functionality, set `compile_new_boolean_structure = true`. (The default is false.)

Improving Random Logic Designs

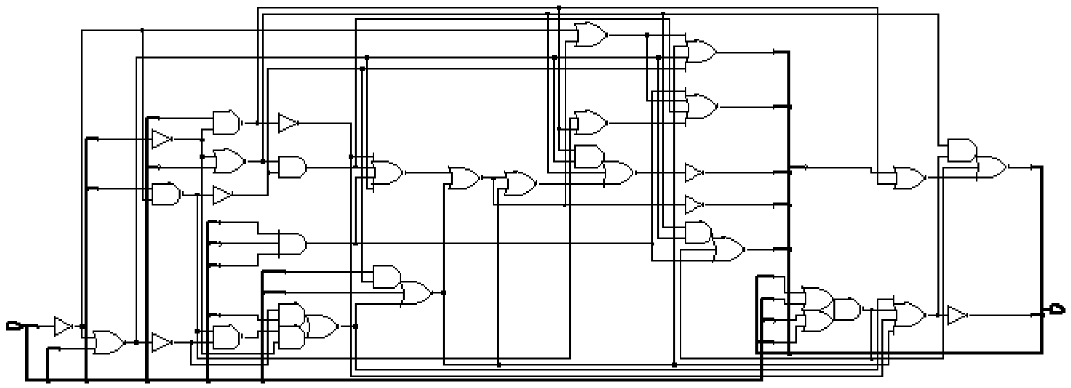
Boolean optimization enhances area optimization. Using the Boolean optimization option enables you to achieve the greatest improvement in random logic designs that have minimal or no structure by creating better structures. State machine designs are also good candidates for Boolean optimization.

Boolean optimization is turned off by default because the tradeoff for the area optimization improvements is a large delay and CPU cost.

If a design has delay constraints and Boolean optimization is enabled, Boolean optimization and timing-driven structuring are both applied during structuring. This technique can sometimes produce a faster design than timing-driven structuring alone.

Figure 4-12 shows the result of Boolean optimization on the design shown in Figure 4-10 and Figure 4-11. This implementation normally achieves the most-area-efficient design but noticeably increases the delay.

Figure 4-12   Compiling With Boolean Optimization



Setting the Boolean Structuring Effort

The Boolean structuring effort option provides flexibility in specifying the amount of CPU effort spent in structuring the design. The `-boolean_effort` option is enabled only when the `compile_new_boolean_structure` variable is set to true (the default is false). If `compile_new_boolean_structure` is set to false, Design Compiler uses the earlier Boolean structuring algorithms.

The algorithms for Boolean structuring introduced in v1997.01 are designed to scale with the size of designs and to work efficiently with memory usage.

The `set_structure -boolean` command, used to specify Boolean structuring, includes the Boolean effort option, `-boolean_effort`. Use this option to specify the amount of CPU effort used for structuring the designs. Set the `compile_new_boolean_structure` variable to true, or Design Compiler will ignore the specified effort level.

`set_structure -boolean true -boolean_effort low`

Is the default when Boolean structuring is enabled. Typically, low is appropriate for structuring most designs. It produces good results and is the least CPU-intensive.

`set_structure -boolean true -boolean_effort medium`

Results in additional structuring steps when Boolean structuring is enabled. The structuring process is carried out in more than one pass. The medium effort level is the recommended effort level and is sufficient for obtaining optimal results for most designs.

`set_structure -boolean true -boolean_effort high`

Combines more CPU-intensive strategies than the low and medium efforts. Multiple passes are used in structuring the design. The structuring process continues until all strategies have been tried.

Controlling Specific Optimization Steps

Design Compiler performs the following six major optimization steps on cells during logic synthesis:

- Global logic structuring. Design Compiler does combinational logic structuring for area and timing.
- Constant propagation. The tool propagates logic constants (0/1) through combinational logic, producing simpler logic.
- Deleting unconnected gates. Design Compiler deletes gates with unconnected (unloaded) output pins from the design.
- Local optimizations. The tool does optimizations “in the neighborhood” of given cells. This step includes sequential and fanout optimizations.
- Critical path resynthesis. In this operation (enabled in high-effort compiles only), Design Compiler groups and reimplements a portion of the critical path of the design, to speed it up.
- Gate sizing. Here, the tool upsizes gates on the critical path to improve performance and downsizes gates on other paths to recover area.

Using set_compile_directives Command

The `set_compile_directives` command, which requires a DC Ultra license, provides close control over these optimizations. You can control which optimizations the Design Compiler performs and does not perform on specific areas of your design.

Whenever you use `set_compile_directives`, Design Compiler disables global logic structuring and enables gate sizing. The other four major steps are controlled by the command options, as shown in the syntax:

```
set_compile_directives object_list
     [-delete_unloaded_gate true | false]
     [-constant_propagation true | false]
     [-local_optimization true | false]
     [-critical_path_resynthesis true | false]
```

object_list

Specifies the objects (cells, references, designs, or library cells) to which the command applies. If you include more than one object, enclose them in quotes or {} braces.

`-delete_unloaded_gate true | false`

Deletes unloaded gates (gates that do not drive a load) from the cells in the *object_list*. The default is true. If you set this option to false, the tool does not delete gates—even if they drive no load.

`-constant_propagation true | false`

Propagates constants through the cells in the *object_list*. The default is true.

`-local_optimization true | false`

Enables buffering, phase assignment, and other local transforms on cells in the *object_list*. The default is true. If you use this option and set it to true, the process might add gates to the path. Those gates do not inherit the same compile directives; that is, Design Compiler performs all optimization steps on them

`-critical_path_resynthesis true | false`

Includes cells in the *object_list* during critical path resynthesis. The default is true.

Example

To allow local optimization but prohibit the deletion of unloaded gates, prohibit constant propagation, and prohibit critical path resynthesis, enter

```
dc_shell> set_compile_directives -del false -const false -crit false
```

Using set_size_only Command

The `set_size_only` command provides a method of gate sizing only. It allows you to set a `size_only` attribute on a cell instance. The `size_only` attribute can only be applied to leaf cell instances.

During optimization Design Compiler will replace these cells with identical functionality.

The syntax for `set_size_only` command is:

```
set_size_only object_list flag
```

Argument	Description
object_list	A list of leaf cell instances on which to set the `size_only` attribute
flag	Determines the value (true or false) to which the `size_only` attribute is to be set. The default is true.

For example,

```
set_size_only U2 false
```

In this example the U2 cell will not have the `size_only` attribute.

The `size_only` attribute can also be specified as a library attribute.

Applying Circuit Strategies

All logic designs and hierarchical modules are one of two types: structured or unstructured, as described in the following sections.

The sections “Applying Strategies for Unstructured Designs” and “Applying Strategies for Structured Designs” later in this chapter, summarize results of various strategies. The results assume that the following are present:

- A reasonable drive at the inputs (the default is infinite drive), set with the `set_driving_cell` command
- A reasonable load on the outputs (the default is zero), set with the `set_load` command

- A wire load model, set with the `set_wire_load_model`, `set_wire_load_mode`, `set_wire_load_min_block_size`, or `set_wire_load_selection_group` commands

The constraint values assigned to these variables are library-dependent.

For minimum area, specify an area constraint, using the `set_max_area` command.

For maximum speed (minimum delay), specify a clock period and input and output delay.

Include your design specifications as constraints. Set realistic constraints. Do not set a max_delay of 0 on an output, because timing-driven structuring allows the creation of repeatable logic to decrease the critical path. An unrealistic constraint, such as max_delay 0, rapidly increases the area. If you are optimizing for area only, do not apply any delay constraints, even if they are loose constraints.

Structured Circuits

Structured circuits contain data-path components, MUXs, XORs, and regular logic. ALUs, adders, multipliers, and comparators all belong to this class. Because this type of circuit usually contains carefully crafted and highly structured logic, do not flatten it unless you also structure it. If you flatten a design without also structuring it, you might remove all or part of the existing structure, resulting in a less than optimal design.

When optimizing a structured design for area or speed, first consider using default options, such as timing-driven structuring and mapping.

Unstructured Circuits

Unstructured circuits contain no data-path components, few XORs or MUXs, and no ALUs or adders. Examples of unstructured logic are random control logic and table-lookup circuits that can be implemented as ROMs. When optimizing an unstructured design for area or speed, consider using the default options first.

Applying Strategies for Structured Designs

These sections describe the results of applying different strategies to a structured design. The strategies concentrate on whether to flatten or structure a design and when and how to use different options, such as

- Flattening effort
- Boolean optimization and timing-driven structuring

Structured circuits contain data-path components, MUXs, XORs, and logic that is well structured. ALUs, adders, multipliers, and comparators belong to this class.

The examples in this section use different strategies to optimize a 32-bit carry-lookahead adder used for fast designs.

Optimizing for Area

When optimizing for area, you can obtain good area results by using the default optimization settings.

```
compile
```

If you do not get good results, use mapping only.

```
set_structure false
compile
```

For further area optimization, use Boolean optimization.

```
set_structure true -boolean true -boolean_effort medium
compile
```

Optimizing for Speed

When optimizing for speed, experiment first with the default settings, structuring, and mapping. Timing-driven structuring—the default structuring option—optimizes the critical paths for delay while maintaining area efficiency on the noncritical paths.

To use the default settings, enter

```
compile
```

If this produces unacceptable results, try mapping only.

```
set_structure false
compile
```

Try flattening with structuring. In general, do not flatten structured designs unless you follow with structuring. The flattening might remove all or part of the existing structures. However, if flattening is followed by structuring, timing-driven structuring often builds good structures that both reduce the area and improve the critical path delays.

```
set_flatten true
compile
```

Area Optimization Analysis

Boolean optimization achieves the best area optimization, as shown by the top three strategies listed in Table 4-3. Flattening without structuring produces the largest area, as shown by the last strategy in the table. Flattening without structuring increases the area because it removes all the structures from a design. The default strategy, with structuring and mapping, achieves a reasonable area.

Table 4-3 Structured Design: Area Optimization

Strategies With Structuring	**Area**	**Critical Path**
set_structure -boolean true	273	98.17
set_structure -boolean true set_flatten true -phase true	302	132.63
set_structure -boolean true set_flatten true	307	109.75
Default	359	53.04

Strategies Without Structuring	Area	Critical Path
set_structure false	428	24.04
set_structure false set_flatten true	1,378	55.82

Speed Optimization Analysis

Timing-driven structuring, as shown by the first two strategies listed in Table 4-4, structures the critical paths for speed, even though flattening removes or partially removes the existing structures.

Table 4-4 Structured Design: Speed Optimization

Strategies With Structuring	Area	Critical Path
set_flatten true -phase true	812	11.76
set_flatten true	827	12.42
Default	646	12.81

Strategies without Structuring	Area	Critical Path
set_structure false	512	16.42
set_structure false set_flatten true	1896	20.15

Note:

> Flattening only, with no structuring, greatly increases both the critical path delay and the area because it removes all structures from a design and does not rebuild a good structure, as shown by the last strategy in Table 4-4.

Applying Strategies for Unstructured Designs

These sections describe the results of applying different strategies to unstructured designs and when and how to use different options, such as

- Flattening effort
- Boolean optimization and timing-driven structuring

These sections also describe other flattening options that fine-tune the results of optimization, such as single-output or multiple-output minimization and phase inversion. These other options have less effect on optimization than flattening and structuring.

Optimizing for Area

When optimizing for area, you can probably obtain good area results using the default optimization settings. To use the defaults, enter

```
compile
```

If you are dissatisfied with the results, try flattening with structuring.

```
set_flatten true
compile
```

For further area optimization, try Boolean optimization.

```
set_structure true -boolean true -boolean_effort medium
compile
```

Optimizing for Speed

When optimizing for speed, experiment first with the default settings, structuring, and mapping.

```
compile
```

If this produces unacceptable results, try flattening with structuring.

```
set_flatten true
compile
```

If area is not an issue, try flattening without structuring. As discussed previously, flattening a design and then mapping it produces a tall, narrow design. If the inputs are not excessively loaded, this strategy might produce the best speed results, although it might take much longer to process large blocks.

```
set_structure false
set_flatten true
compile
```

If the results from the previous strategies are unacceptable and Design Compiler was unable to flatten your design using its low effort, try the previous strategies with a greater flatten effort.

```
set_structure false
set_flatten true -effort medium
compile
```

Fine-Tuning the Optimization

If you are concerned with obtaining the best-possible optimization, you can try improving your results by using multiple-output minimization and phase inversion when flattening is enabled, in addition to using the strategies described previously.

```
set_flatten true -minimize multiple_output
set_flatten true -phase
```

Area Optimization Analysis

The best area optimization strategy for the random logic design, as shown in Figure 4-5, is flattening the design, performing multiple-output minimization, then structuring with Boolean optimization. Table 4-5 shows that the default optimization options also produce reasonably good area optimization results.

In the example, not using structuring logic produced the worst area results. This strategy prevented Design Compiler from sharing terms, thus minimizing product terms and gate count in the circuit’s logic.

Boolean optimization produced the best area optimizations, as shown in Table 4-5.

Table 4-5 Unstructured PLA Design: Area Optimization

Strategies With Structuring	**Area**	**Critical Path**
set_structure -boolean true set_flatten true -minimize multiple_output	800	77.72
set_structure -boolean true set_flatten true -minimize multiple_output -phase true	806	98.87
set_flatten true -minimize multiple_output	847	32.10
Default	959	27.92

Strategies Without Structuring	**Area**	**Critical Path**
set_structure false	2583	128.77
set_flatten true set_structure false	2685	141.48

Speed Optimization Analysis

The best speed optimization strategy for the unstructured design, shown in Table 4-6, is flattening and mapping with no structuring. Strategies with structuring, including the default, also produce fast critical paths without affecting the area, because timing-driven structuring is enabled in all structuring runs.

Table 4-6 Unstructured PLA Design: Speed Optimization

Strategies With Structuring	Area	Critical Path
set_flatten true -phase true	1763	10.51
set_flatten true	1620	10.70
Default	1765	10.58

Strategies Without Structuring	Area	Critical Path
set_structure false set_flatten true	4468	9.98
set_structure false set_flatten true -phase true	3630	10.71

Summarizing the Optimization Strategies

Table 4-7 summarizes the optimization strategies described for these examples.

Table 4-7 Summary of Optimization Strategies by Circuit Type

	Structured Circuit	Unstructured Circuit
Area	1. Structure, map (default) 2. Map 3. Structure, Boolean, map	1. Structure, map (default) 2. Flatten, structure, map 3. Structure, Boolean, map
Speed	1. Structure, map (default) 2. Map 3. Flatten, structure, map	1. Structure, map (default) 2. Flatten, structure, map 3. Flatten, map 4. Medium flatten, map

Structured Logic Summary

This section summarizes the effect of structured logic on area and speed.

Area

The default structuring improves the quality of results. If the results are unsatisfactory, try only mapping the design, without flattening or structuring, or apply Boolean optimization to further improve the area. However, Boolean optimization can increase the delay and the CPU runtime.

In general, if you flatten a structured design, follow up by structuring it. When Design Compiler flattens a design, it removes all or part of the existing structure and does not rebuild the structure unless you direct it to.

Speed

Try the default optimization settings, timing-driven structuring (the default), and mapping. If you are not satisfied with the results, try only mapping the design, without flattening or structuring. For some designs, flattening followed by structuring might result in the best timing optimization.

In general, do not flatten a structured design unless you follow by structuring, because flattening removes all or part of the existing structures and does not rebuild a good structure.

Unstructured Logic Summary

This section summarizes the effect of unstructured logic on area and speed.

Area

Try the default optimization options first. If this strategy does not produce good results, try flattening with structuring. Apply Boolean optimization for increased area enhancements, but keep the delay and CPU tradeoffs in mind.

Speed

Apply the default setting first. Next, try flattening with structuring. For some designs, flattening followed by mapping can result in the best timing optimization. You might need to increase the flattening effort level to medium to complete flattening on some designs.

5

Pipelining Data Paths

Pipelining is a technique commonly used to increase the throughput of designs that have deep levels of logic to meet high sampling rate requirements. The pipelining technique partitions blocks of combinational logic into n stages of equal delays, with the stages separated by banks of pipeline registers.

Register balancing improves the performance of the pipeline design, by moving the registers through combinational logic, adding additional registers as needed, to reduce the clock period.

This chapter contains the following sections:

- Basic Concepts
- Considering Input and Output Delays
- Handling Designs Having Hierarchy
- Enabling Register Balancing

- Inserting Pipeline Registers in a Design
- Hints and Restrictions
- Resetting Flip-Flops
- Replacing Flip-Flops

This chapter includes example methods of inserting the pipeline registers in a design through HDL and a script to convert simple flip-flops to asynchronously reset flip-flops.

Basic Concepts

The `balance_registers` command works by timing a design and moving registers through the logic levels of the design so that the delays between the register banks are equal. The `balance_registers` command affects the state of the flip-flops internal to a design but maintains cycle-to-cycle behavior at all outputs of the design.

Delay balancing involves a series of local transformations known as pipelining, which moves registers across logic. Figure 5-1 shows registers moving from output to input. Figure 5-2 shows registers moving from input to output. The `balance_registers` command uses pipelining to generate a design that has minimum cycle time.

Figure 5-1   Pre-Pipelining and Post-Pipelining, Output to Input

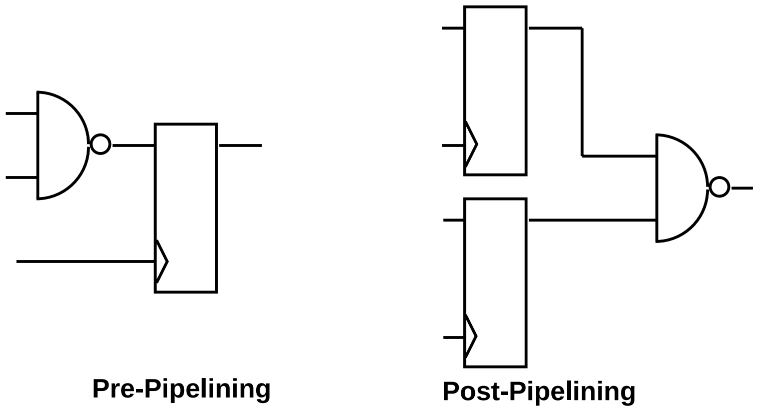
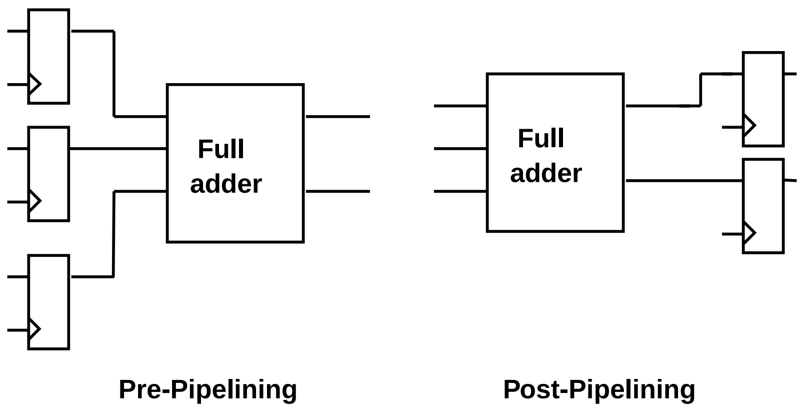


Figure 5-2   Pre-Pipelining and Post-Pipelining, Input to Output



Keep in mind these facts about register balancing when you use it:

- Perform register balancing throughout the entire design.
- Consider using external boundaries to produce optimal cycle times, because `balance_registers` moves register boundaries.
- Be aware of input arrival times and output required times, which define the context of a design. These times affect the position of the pipeline registers relative to input and output ports.
- You cannot verify designs by using compile -verify while the `balance_registers` attribute is set to true.
- Ensure that your design contains no generic logic when `balance_registers` is called during compilation. Running `balance_registers` on a design that contains generic logic is an error. If the `balance_registers` attribute is set, `compile` attempts to optimize the design by running `balance_registers`.
- Subdesigns in the hierarchy are ungrouped into the design unless the `dont_touch` attribute is set.

The pipelined design throughput is equal to n times the throughput of a nonpipelined design and has a latency of n clock cycles. Latency is the number of clock cycles needed to propagate the results out to the output from the input. Pipelined designs have an increased gate count, because of the insertion of pipeline registers, which are used to hold the values of the previous blocks.

An advantage of pipelined designs is that they produce a valid result at the output from the design after every clock cycle. This condition applies only after the first set of data has propagated through the design and when data is entered into the design during every clock cycle. A disadvantage of pipelined designs is the latency between the data input and the resulting output. Pipelining is most useful for systems receiving data every clock cycle in which the clock period is small.

An issue with pipelining is the need to partition the design in blocks of logic that have equal delays (delay balancing). The `balance_registers` command addresses this issue and generates a pipelined design that has equal amounts of combinational logic between pipeline registers.

Considering Input and Output Delays

The circuit shown in Figure 5-3 receives inputs from a register and is followed by combinational circuitry. When balancing registers for the first stage, observe that part of the clock cycle is already taken up by external logic. If you do not consider this condition, the pipelined design will not meet your design goals.

Figure 5-3 shows how external delays can affect achieving the design goal of 10 ns. The 5-ns and 4-ns delays are external to the design to be pipelined, and there are two flip-flops that can be used for pipelining. If the design is analyzed, it can be implemented with a 10-ns clock cycle.

Note:

> Instantiated pipelined designs can derive their timing from the `characterize` command.

Figure 5-3 Design Before Pipelining

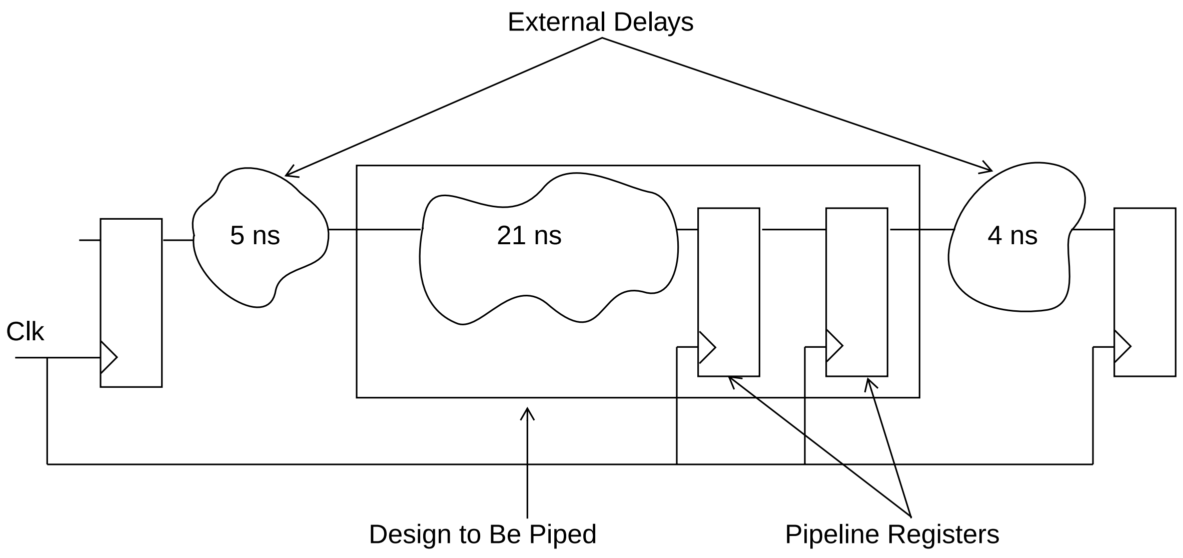


Figure 5-4 and Figure 5-5 show the importance of considering input and output delays when pipelining.

Figure 5-4 shows the result of attempting to balance the internal combinational block without considering external delays. When balanced evenly over three cycles, the 21-ns delay produces an even delay of 7 ns. The system clock cycle is dictated by the first set of delays (5 ns plus 7 ns) and is 12 ns. This delay is more than the 10-ns clock-cycle design goal. However, the design has a suboptimal clock cycle, because the context of the design is considered.

Figure 5-4 Design Goal of 10-ns Period Not Met

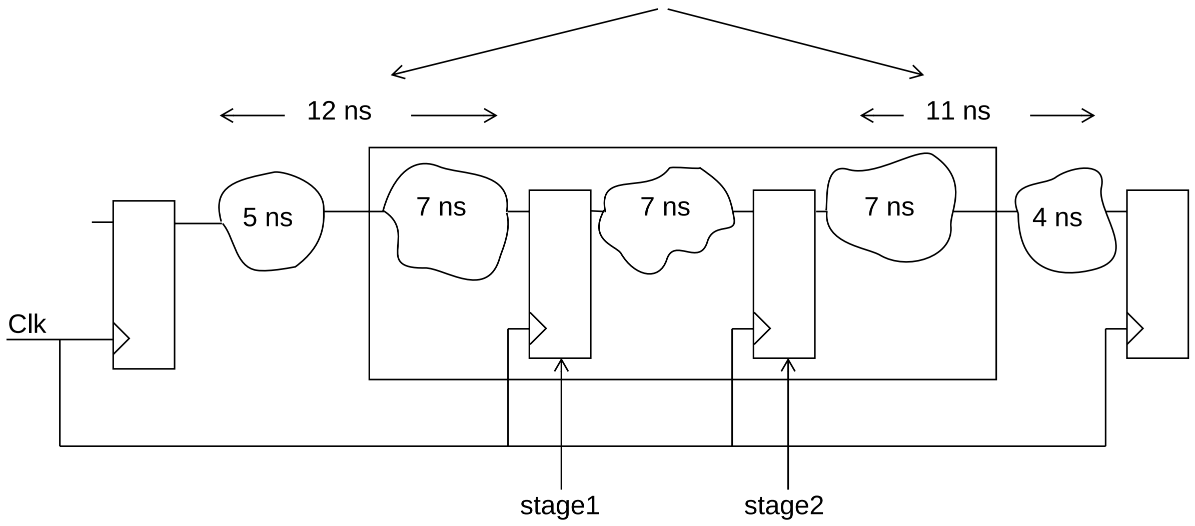
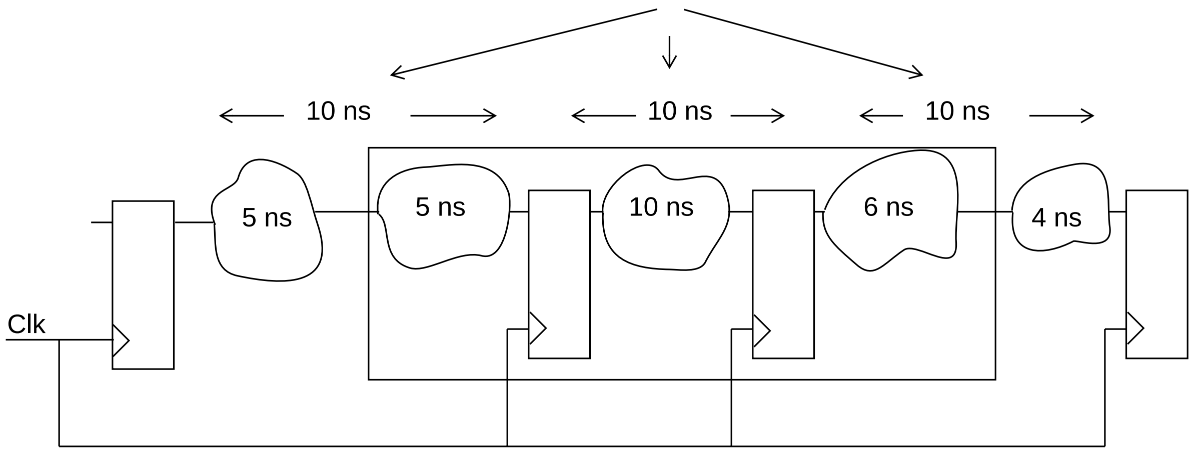


Figure 5-5 shows a design with an optimal clock cycle when external delays are considered. After the external delays are accounted for, delays are balanced among all flip-flops in the design and, therefore, meet the design goal of 10 ns.

Figure 5-5   Design Goal of 10-ns Period Met



Handling Designs Having Hierarchy

The `balance_registers` command first ungroups (removes) the hierarchy, then pipelines the flattened (nonhierarchical) design. This method of pipelining the design ensures that minimum cycle times can be found, because maximum flexibility is allowed in the network transformations (no functional boundaries get in the way). The original circuit structure is lost as a result of ungrouping the hierarchy.

You can maintain the design hierarchy by placing `dont_touch` attributes on selected instances. Instances that have the `dont_touch` attribute are treated as leaf cells and timed through; however, registers are not moved into the dont_touch instances and the hierarchy is maintained. This condition can lead to suboptimal cycle times, because the pipelining algorithm only moves registers around the dont_touch instances. A worst-case scenario occurs when the delay through the dont_touch instance dominates or determines the cycle time.

Enabling Register Balancing

You can enable register balancing in two ways:

- Explicitly invoke register balancing on a design using the `set_balance_registers` command.
- Set the `balance_registers` attribute on a design to invoke register balancing during compile.

The `balance_registers` command moves the registers of the current design to obtain a minimum cycle time.

The `set_balance_registers` command determines whether the registers in a design are retimed during compile. Retiming a design’s registers requires moving registers to achieve a minimum cycle time. A design is not retimed during compile unless `balance_registers` is set to true. Setting `balance_registers` on a design ungroups subdesigns in the hierarchy into the design, unless `dont_touch` is set.

Registers must be edge-triggered flip-flops clocked by the same phase of the same clock. Instances that have the `dont_touch` attribute cannot contain sequential elements.

You need not set the `balance_registers` attribute before you use the `set_balance_registers` command.

The syntax is

```
set_balance_registers true | false [-design design_list]
```

`true | false`

Enables or disables register balancing.

- true (the default) enables register balancing.
- false disables register balancing.

`-design` *design_list*

Specifies a list of designs to retime. If you omit this option, your current_design is used (the default).

To remove the `balance_registers` attribute, use the `remove_attribute`, `reset_design`, or `set_balance_registers false` command.

Use the `balance_registers` command to balance delays on any sequential designs, including state machines. However, only systems that can accept a latency and continuously get data can fully utilize this command. The main obstructions to the performance of the `balance_registers` command are the `dont_touch` attribute (on flip-flops or instances) and the reading of intermediate nonregistered signals through primary outputs.

For example, if an asynchronous reset (see Figure 5-6 on page 5-17) is tied to the flip-flops and is set high in the middle of a cycle, the design outputs zeros for the next two clocks. If you apply the same condition to the post-pipelined design in Figure 5-7 on page 5-18, the circuit does not supply zeros over two clock cycles. Therefore, flip-flops with asynchronous resets are treated as dont_touch flip-flops.

Inserting Pipeline Registers in a Design

You can insert pipeline registers into a design in several ways. You can

- Use signal assignment
- Pipeline registers with depth and width
- Use a single pipeline stage

Using Signal Assignment to Pipeline Registers

This code example is sufficient when the number of stages is fixed. With a bit more detail, you can also create a module that has a configurable pipeline depth and width. One advantage of this method is that it produces reusable code. Also, you can experiment with multiple pipeline depths by changing the generic parameters.

```
--      CODE 1
-- pipe_infer.vhd
Library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity pipe_infer is
  generic ( width : natural);
  port ( clk  : in std_logic;
             a, b  : in signed (width-1 downto 0);
             c     : out signed ( 2*width-1 downto 0));
end pipe_infer ;

architecture two_processes of pipe_infer is
   signal stage0, stage1, stage2 : signed (2*width-1 downto 0);
begin

     -- Process that has the core (multiplier) that is to be pipelined
     process(a,b)
     begin
          stage0 <= a*b;
     end process;
     -- Process that creates three stages of pipelining
     process
     begin
         wait until clk'event and clk='1';
       stage1 <= stage0;
       stage2 <= stage1;
       c <= stage2;
     end process;

end two_processes;
```

Pipelining Registers With Depth and Width

This code example shows another method for inserting pipeline registers into a design.

```
--  Code 2
--  pipe_test.vhd
--
--  Width configurable multiplier and configurable pipeline register
--  stages
--  ****** Design that instantiates the core and the pipeline registers
Library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity pipe_test is
   generic (width   : integer;
            latency : integer);
   port( clk  : in std_logic;
         a, b : in signed (width-1 downto 0);
         c         : out signed(2*width-1 downto 0) );
end pipe_test;

architecture A of pipe_test is
-- This is a structural description that hooks up the lower-level components.
-- Hook up the multiplier to a bank of registers for a multi-level deep bank;
-- else hook multiplier outputs directly to the inputs of a D flip-flop.
   component mult
      generic (width : integer);
      port( a, b : in signed (width-1 downto 0);
            c : out signed(2*width-1 downto 0) );
   end component;
   component delay_stage
      generic(width, latency : integer);
      port(clk     : in std_logic;
           a       : in signed (width-1 downto 0);
           b       : out signed (width-1 downto 0) );
   end component;
   component dff
   port( d  : in  std_logic;
         q  : out std_logic;
         clk  : in  std_logic);
   end component;

   signal raw_mult : signed (2*width-1 downto 0);

   begin
   -- instantiate the part to be pipelined (core)
   inst1: mult generic map (width)  port map (a,b,raw_mult);

   -- instantiate the pipeline registers
   c1: if latency > 1 generate
      inst2: delay_stage generic map(2*width,latency)  port map(clk, raw_mult,
c);
   end generate;
```

Connecting By Using One Pipeline Stage

This code example shows only one pipeline stage.

```
-- If there is only 1 pipeline stage,connect the inputs directly to
   -- the inputs and tie the outputs of the dff directly to the primary
   -- output.
   c2: if latency = 1 generate
      f1: for i in 2*width-1 downto 0 generate
         inst3: dff port map (raw_mult(i), c(i), clk);
      end generate;
   end generate;
   end A;
-
- **********  Core (multiplier) to be pipelined
Library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity mult is
   generic(width : integer);
   port( a, b : in signed (width-1 downto 0);
         c    : out signed(2*width-1 downto 0));
end mult;

architecture A of mult is
   begin
   c <= a * b;
end A;
-- ************  Configurable depth and width pipeline register bank
Library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity delay_stage is
   generic(width : integer;
           latency : integer);
      port(clk : in std_logic;
           a  : in signed (width-1 downto 0);
           b  : out signed (width-1 downto 0) );
end delay_stage;

architecture A of delay_stage is

   component dff
   port( d   : in  std_logic;
         q   : out std_logic;
          clk : in  std_logic);
   end component;
   -- make an array of wires, which will be used to connect successive
   -- ports of registers. latency-2 because no net necessary for input into
   -- first bank and no net necessary for last banks output to primary output
   type intern_type is array (latency-2 downto 0) of \
   std_logic_vector(width-1 downto 0);
   signal intern : intern_type;

   begin
   -- make a word bit by bit
   g1: for i in 0 to width-1 generate

   -- make the number of pipeline stages necessary
       g2: for j in 0 to latency-1 generate

   -- Connect the input bus into the first bank
          c1: if j = 0 generate
             L1: dff port map (a(i), intern(j)(i), clk);
           end generate;

   -- If it is not the first or last bank, connect the last output bit
   -- to the corresponding input bit of the next bank.
```

```
       c2: if j /= 0 and j /= latency-1 generate
          L2: dff port map (intern(j-1)(i), intern(j)(i), clk);
       end generate;

-- If it is the last bank, connect the output last banks output
-- directly out
       c3: if j = latency-1 generate
          L3: dff port map (intern(j-1)(i), b(i), clk);
       end generate;

    end generate;
end generate;
end A;
```

You can use either the configurable depth and width or the single pipeline method to achieve similar results. The pipeline stages do not have to be placed at the outputs. They can be placed anywhere in the circuit: in the middle, at the inputs, or at both the inputs and the outputs. However, keeping all your pipeline registers at the outputs makes writing the code easier.

Script 1

This script can give you good results after the design containing the block to be pipelined and the pipeline registers have been read in.

```
read -f vhdl design_to_be_piped.vhd
create_clock clk -period no_of_stages    /*desired_clk_period*/
compile -map_effort low                                             1
create_clock clk -period desired_clk_period
balance_registers                                                   2
compile -map_effort high                                            3
```

In the script,

- no_of_stages is the number of pipeline banks in the design.
- `compile -map_effort low` is necessary before you issue the `balance_registers` command, because `balance_registers` rearranges the gates and performs another mapping.

- desired_clk_period is the desired post-pipelining clock period.

Include these commands in the script:

1. Perform a quick initial map (using `compile -map_effort low`).
2. Issue the `balance_registers` command.
3. Perform a `compile -map_effort high`.

Script 2

Use this script to pipeline only a portion of a design.

```
read -f vhdl design_to_be_piped.vhd
create_clock clk -period realistic_clk_period* desired_clk_period */             1
compile -map_effort low                                                          2
group [logic_to_be_pipelined and pipeline FFs] -design ret_blk -cell ret_blk   3,4
characterize ret_blk                                                             5
current_design = ret_blk                                                         6
create_clock clk -period desired_clk_period
balance_registers
compile -map_effort high
set_dont_touch ret_blk
current_design = top
create_clock clk -period desired_clk_period
compile -map_effort high
```

In the script,

1. Set a realistic clock constraint.
2. Perform a quick initial (low effort) mapping to gates.
3. Group only the gates that need to be pipelined into another level of hierarchy.
4. Now, pipelining can be performed on a well-defined design. If the subdesign is a separate block (process), use `group -hdl_block block_name` to move the necessary logic and the pipeline flip-flops to a separate level of hierarchy.

5. Using the `characterize` command, extract the context of the design_to_be_pipelined.

6. Annotate the information to the input and output pins.

The remaining steps are similar to those in Script 1.

Figure 5-6 is a design represented by the code example in the section "Using Signal Assignment to Pipeline Registers" on page 5-11. Figure 5-7 is the same design after pipelining.

Figure 5-6 Design Before Pipelining (balance_registers)

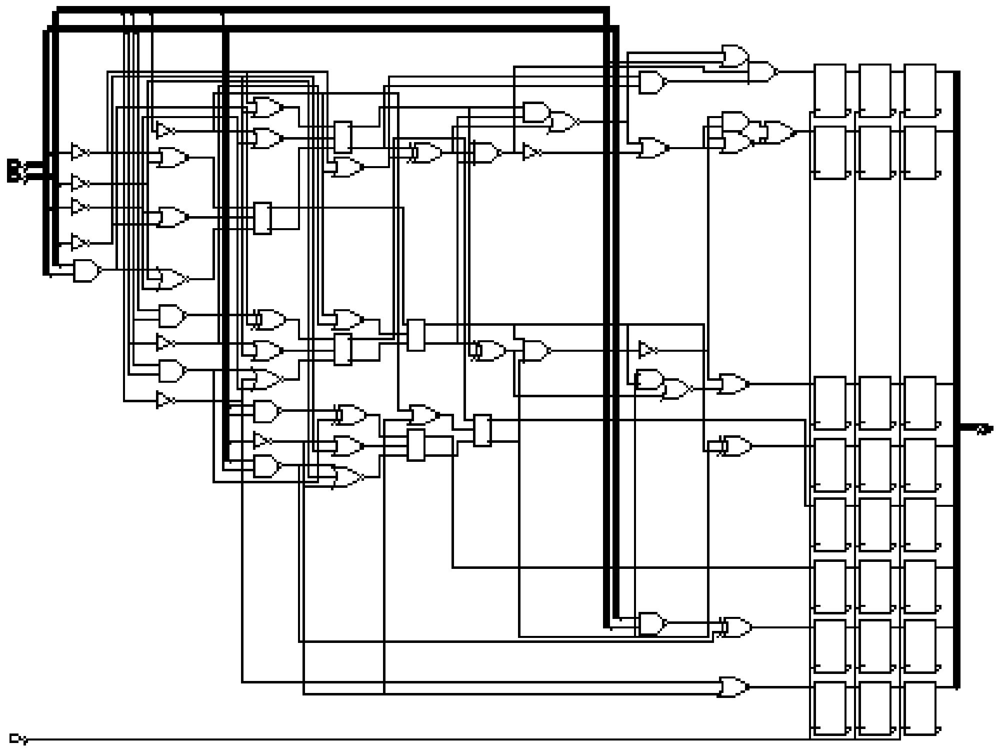
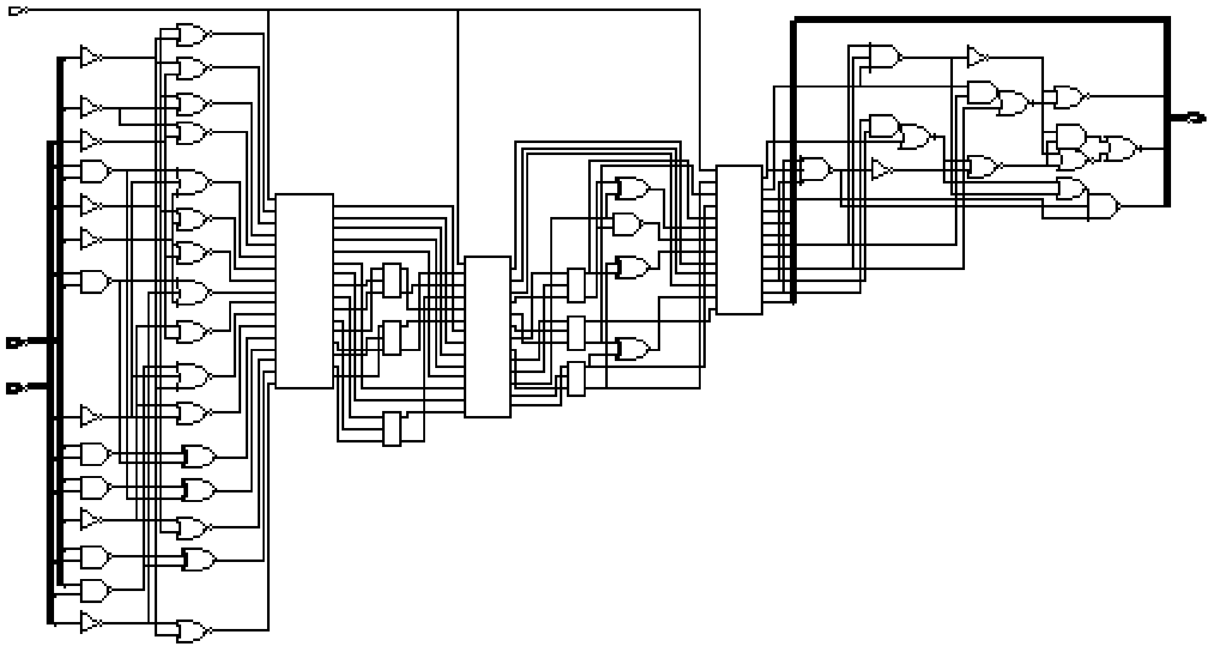


Figure 5-7   Design After Pipelining (balance_registers)



Hints and Restrictions

This section provides hints and restrictions for using the `balance_registers` command.

Hints and Restrictions for Flip-Flops

- The `balance_registers` command treats any flip-flop with an asynchronous set and reset as if the `dont_touch` attribute is in effect and does not move it. This is because `balance_registers` does not determine the initial state of the flip-flop when it is asynchronously set through the set and reset pins.

- Use the `dont_touch` attribute sparingly when you use `balance_registers`. The `balance_registers` command attempts to decrease the clock cycle but might not be able to produce optimal cycle times, because of using `dont_touch` flip-flops, maintaining hierarchy through `dont_touch`, and so on.
- Use only edge-triggered D flip-flops when you use `balance_registers`; neither latches nor asynchronous set and reset flip-flops can be used. All the clock inputs of the flip-flops in the design must be connected to the same edge of the same clock (the same net). Build the clock tree only after the design has been pipelined.

Hints and Restrictions for Input and Output Registering

Figure 5-7 on page 5-18 shows a design after pipelining. Notice the combinational logic after the last register bank, which can be prevented by using either the dont_touch method or the group method.

If the outputs are to be registered, place a `dont_touch` attribute on the last bank of flip-flops in the design after performing the first compile.

If the inputs are also to be registered, register them before pipelining, then place a `dont_touch` attribute on them.

Another technique for registering the inputs and outputs is to first describe the input registers, pipeline registers, output registers, and the block in an HDL file. You can use a strategy similar to the one used in Script 2 (shown previously) to pipeline the necessary block and register the inputs and outputs. Group only the logic to be pipelined and the pipeline flip-flops into another level of hierarchy, then pipeline that design.

Hints and Restrictions for Nonregistered Primary Outputs

Designs having internal signals going directly to the primary outputs cause the `balance_registers` command to give suboptimal results. This condition occurs because `balance_registers` respects the combinational block feeding the primary outputs and cannot place the registers before the gates driving the outputs.

The `balance_registers` command always maintains the pre-pipelined, cycle-to-cycle behavior of the design. If you require all gates in the design to be potential candidates for pipelining, make sure that all primary outputs are registered.

Viewing Hints

Figure 5-7 (shown previously) shows each stage of the pipeline in its own level of hierarchy. This is not done automatically and requires manual intervention. To create these new blocks, use this command:

```
dc_shell> group REG_SX* -design Sx
```

In the command, x is the suffix that `balance_registers` has created.

This command groups all registers in a particular stage to a different level of hierarchy and makes the schematic easier to read. The suffixes do not correspond to the stage number for designs that have feedback or recursion (such as multiplier accumulators and IIR filters).

Resetting Flip-Flops

You might want to reset all your flip-flops through an asynchronous reset and still use `balance_registers` to pipeline your design. This section shows how to do so.

The example design is a 12-bit, 4-level-deep AND tree that has a critical path going through four AND gates. “Connecting By Using One Pipeline Stage” on page 5-13 contains the HDL code that describes this design and its corresponding pipeline registers.

```
--
-- and-pipe.vhd
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all

entity ret_and is
  generic (width : integer);
  port ( clk  : in std_logic;
              a     : in std_logic_vector (width-1 downto 0);
              reset : in std_logic; -- place holder
              c     : out std_logic);
end ret_and;

architecture two_processes of ret_and is

 signal stage0, stage1, stage2 : std_logic;
begin

     -- Create the and_tree core that is to be pipelined
     process(a)
     begin
         stage0 <= and_reduce(a) ; -- function that generates an and tree
     end process;

     -- Process that creates three stages of pipelining
     process
     begin
       wait until clk'event and clk = '1';
       stage1 <= stage0;
       stage2 <= stage1;
       c <= stage2;
     end process;

end two_processes;
```

Figure 5-8 shows the schematic synthesized from the HDL code for a 12-bit, 4-level-deep AND tree.

Figure 5-8 AND Tree With Three Pipeline Stages

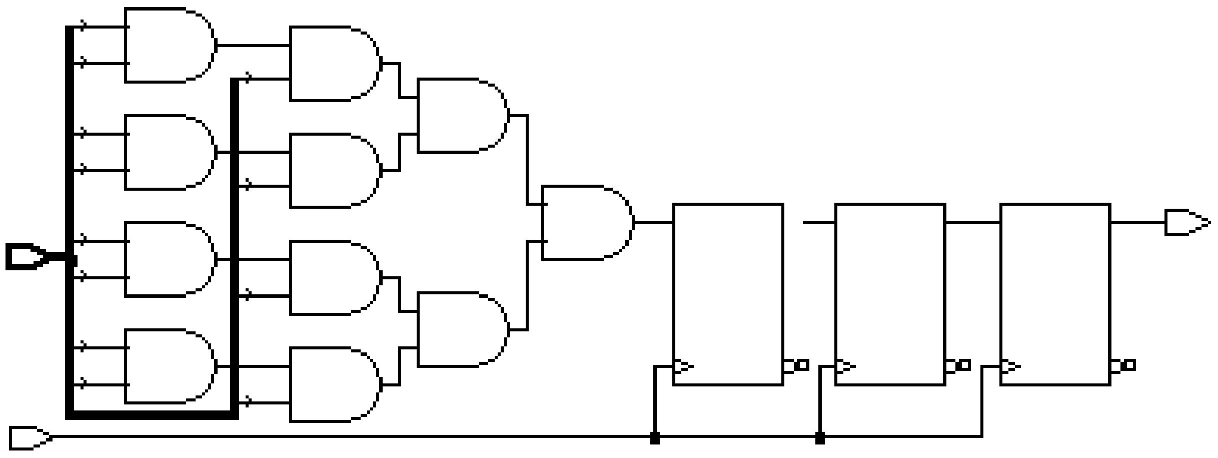
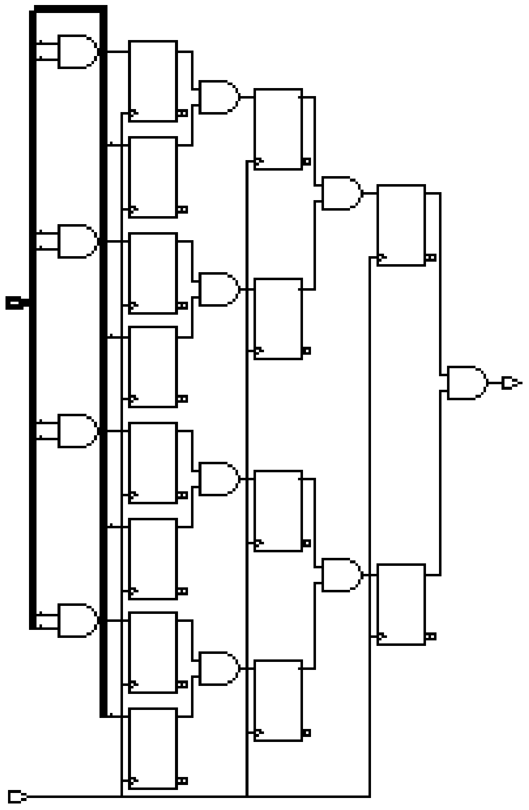


Figure 5-9 shows the result of using the `balance_registers` command on the design in Figure 5-8. The design in Figure 5-9 has three levels of pipelining and a delay of only one AND gate between the stages. All flip-flops in the design are simple edge-triggered D flip-flops.

Figure 5-9   AND Tree After Pipelining



Replacing Flip-Flops

This section shows how to replace all the simple edge-triggered D flip-flops with edge-triggered D flip-flops that have asynchronous reset pins. It also connects all reset pins to a reset port. If you use asynchronous resets in your design, define the reset port in the HDL.

Figure 5-10 shows the code. In the code,

- The first three commands replace all simple edge-triggered D flip-flops (FD1s) in the original design with edge-triggered D flip-flops with asynchronous reset pins (FD2s).

- The command `set_dont_use tech/F*` is necessary because all the flip-flops in this target_library start with the character F.

Figure 5-10 Replacing FD1 Flip-Flops With FD2 Flip-Flops

```
set_dont_use tech/F*
remove_attribute tech/FD2 dont_use
```

These commands tell the compiler that an FD2 is the only flip-flop it can use from the library.

```
  translate
```

At this point, all FD1 cells have been converted to FD2 cells and the async pins have been tied to Vcc (functionally equivalent).

```
   /* script to remove all net connections between pipeline FFs and logic 1.
   The next step is to connect all the asynch_reset pins to a common reset
   port*/

   list_of_regcdpins = " "
   cnt = 0

   list_of_regcdpins = find ( pin REG*_S*/CD )

   foreach ( reg_cdpin, list_of_regcdpins ) {
     net_for_pin = all_connected (reg_cdpin)
     if ((list (net_for_pin)) != "{}") {
       remove_net net_for_pin
  }   else {
        sh echo FF with no net to cd pin found
        cnt = 1
     }
   }
   create_port reset -dir in
```

This command removes the net connected to pin CD of the pipeline registers (REG*_S).

Use this command if there is no reset port in the design.

```
   create_net net_rst
   connect_net (net_rst list_of_regcdpins)
   connect_net net_rst find (port, reset)
```

This command creates a net and connects all pipeline register CD pins to the reset port.

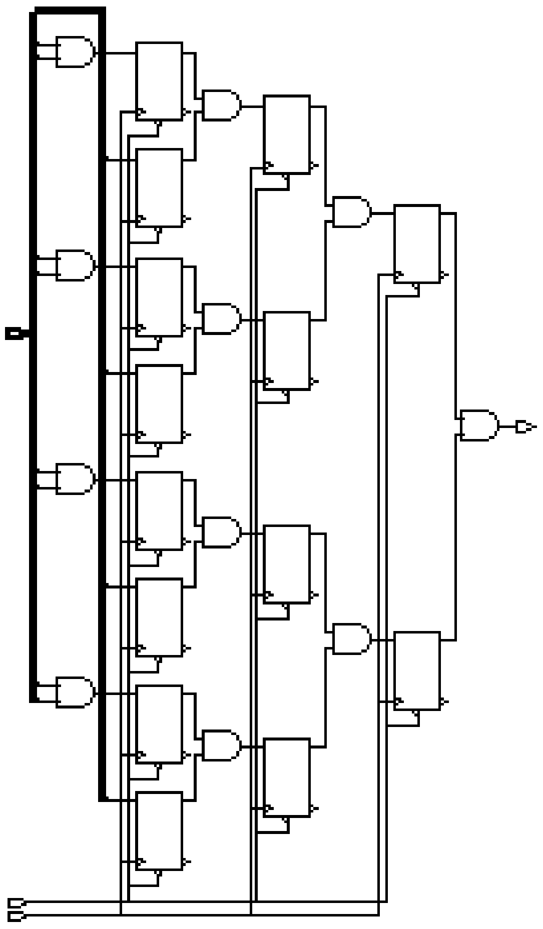
Example 5-1 shows the result (log) of the commands in Figure 5-10.

Example 5-1   Result From Figure 5-10 (Log)

```
/* These three commands will replace all FD1 flip-flops in the original design
with FD2s (FD1 is a simple edge-triggered D FF, FD2 is an edge-triggered D FF
with an asynchronous reset pin). */

set_dont_use tech/F*
Performing set_dont_use on library cell 'tech/FD2SP'.
Performing set_dont_use on library cell 'tech/FJK2SP'.
        •
        •
        •
Performing set_dont_use on library cell 'tech/FJK2'.
Performing set_dont_use on library cell 'tech/FJK3'.
1
remove_attribute tech/FD2 dont_use
Performing remove_attribute on library cell 'tech/FD2'.
{"tech/FD2"}
translate

  Loading target library 'tech'
  Loading design 'ret_and_width12'
  Translating Design 'ret_and_width12'
    from the technology 'tech' (cmos)
    to the technology 'tech' (cmos)

  Transferring Design 'ret_and_width12' to database 'andtree-ret.db'
Current design is 'ret_and_width12'.
1

/* script to remove all net connections between pipeline FFs and logic 1.
The next step is to connect all the asynch_reset pins to a common reset port*/

list_of_regcdpins = " "
" "
cnt = 0
0

list_of_regcdpins = find ( pin REG*_S*/CD )
"{REG4_S1/CD, REG8_S2/CD, REG6_S1/CD, REG2_S3/CD, REG11_S1/CD, REG9_S3/CD,
REG12_S2/CD, REG5_S2/CD, REG1_S2/CD, REG3_S1/CD, REG10_S1/CD, REG7_S1/CD,
REG13_S1/CD, REG0_S1/CD}"



foreach ( reg_cdpin, list_of_regcdpins ) {

net_for_pin = all_connected (reg_cdpin)
  if ((list (net_for_pin)) != "{}") {
      remove_net net_for_pin
  } else {
      sh echo FF with no net to cd pin found
      cnt = 1
  }
}
net_for_pin = "{n97}"
Removing net 'n97' in design 'ret_and_width12'.
net_for_pin = "{n96}"

         •
         •
         •

net_for_pin = "{n84}"
Removing net 'n84' in design 'ret_and_width12'.
1
create_port reset -dir in
Creating port 'reset' in design 'ret_and_width12'.
1
create_net net_rst
Creating net 'net_rst' in design 'ret_and_width12'.
1
connect_net (net_rst list_of_regcdpins)
Connecting net 'net_rst' to pin 'REG4_S1/CD'.
Connecting net 'net_rst' to pin 'REG8_S2/CD'.
         •
         •
         •
Connecting net 'net_rst' to pin 'REG0_S1/CD'.
1
connect_net net_rst find (port, reset)
Connecting net 'net_rst' to port 'reset'.
1
```

Figure 5-11 shows the pipelined design after all FD1 flip-flops have been replaced with FD2 flip-flops. You can easily customize the 12-bit, 4-level-deep AND tree HDL (shown in “Resetting Flip-Flops” on page 5-21) to replace any flip-flop with any other flip-flop. The most important steps are to make sure all flip-flops, except the one needed, have the `dont_use` attribute assigned to them before translation and to define the pin name of the clear pin.

Figure 5-11   Pipelined AND Tree After Flip-Flop Swap



6

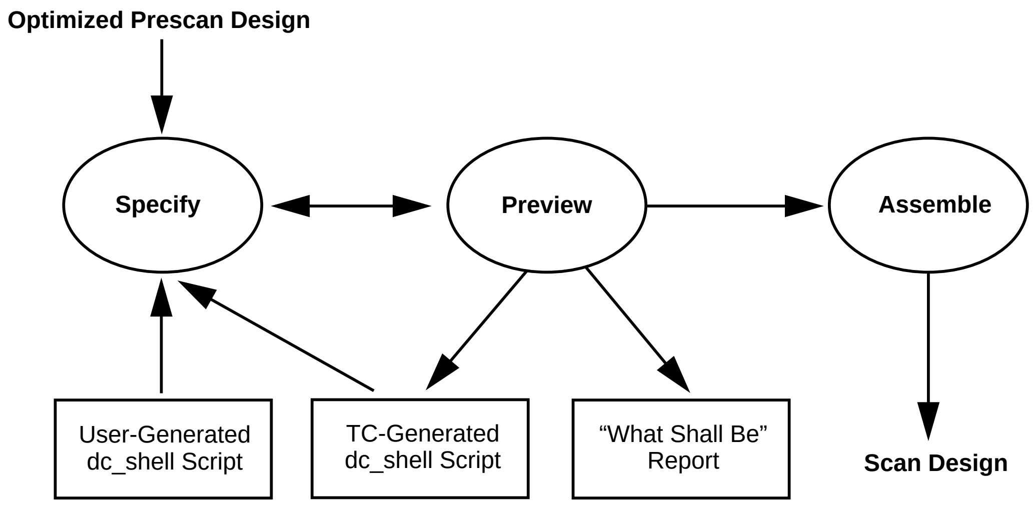
Assembling Scan Structures

If you used test-ready compile, you now have a design that contains unrouted scan cells (prescan design) and you are ready to perform scan assembly. If you did not use test-ready compile, DC Expert *Plus* can perform scan replacement during scan assembly. For best results, use test-ready compile to insert scan cells when your starting point is an HDL description. See Chapter 3, “Optimizing Designs.”

Using the hierarchical scan insertion capability, DC Expert *Plus* supports top-down, bottom-up, and middle-out hierarchical scan insertion flows for full-scan designs. The scan structures in a full-scan design include all sequential cells in your design that pass design rule checking and do not have the `scan_element false` attribute set on them. Although DC Expert *Plus* supports full-scan designs that include nonscan sequential cells, these nonscan cells lower the fault coverage results.

You can provide a complete specification for the scan architecture, a partial specification, or no specification. DC Expert *Plus* generates a complete scan architecture based on the information you provide. You implement the scan architecture by using a specify-preview-assemble process. You might perform multiple specify-preview iterations before you have an acceptable scan architecture. After you have an acceptable scan architecture, invoke the synthesis process to perform scan assembly. Figure 6-1 shows this specify-preview-assemble process.

Figure 6-1 Hierarchical Scan Insertion Design Process



This chapter contains the following sections:

- Specifying the Scan Architecture
- Previewing Scan Structures
- Inserting and Optimizing Scan Structures

Specifying the Scan Architecture

The scan architecture defines the scan style and control-logic requirements and describes how to connect the following scan-path components:

scan element

A scan element is a sequential cell or design instance that will be connected in a scan chain.

scan-data port

A scan-data port is a design port that either drives a scan-input signal or is driven by a scan-output signal.

scan segment

A scan segment is an existing scan structure that will be reused in a scan chain. Scan segments include subdesign scan chains (scan segments inferred by DC Expert *Plus*) and user-defined scan segments.

scan link

A scan link is a connection between scan elements, scan-data ports, or scan segments. Scan links include nets, lock-up latches, and multiplexers. DC Expert *Plus* automatically selects the appropriate scan links but you have the option of overriding the automatic selection.

Figure 6-2 illustrates these scan-path components.

Figure 6-2 Scan-Path Components

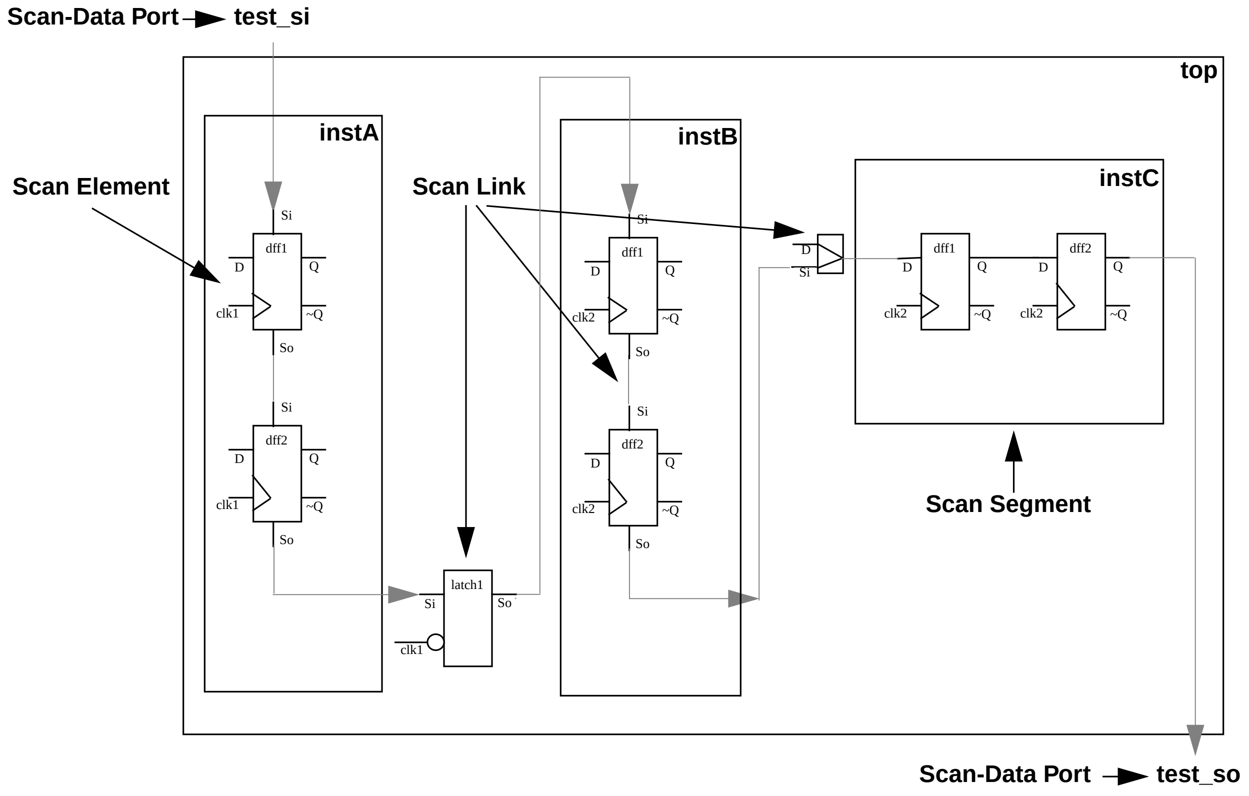


Figure 6-2 shows a single scan path that starts at scan-data port test_si (which receives the scan-input signal) and ends at scan-data port test_so (which drives the scan-output signal). Cells instA/dff1, instA/dff2, instB/dff1, and instB/dff2 are examples of scan elements. The shift register in instC is a user-defined scan segment. If you are using a bottom-up flow and have already processed instances instA and instB, the scan chains in instA and instB are considered subchains or inferred scan segments. The shaded lines represent scan links. The latch (instance latch1) and multiplexer are also scan links.

Specifying and removing the scan architecture consists of the following optional tasks listed in Table 6-1.

Table 6-1 also shows the dc_shell commands that perform each of the scan specification tasks. These commands annotate the database but do not otherwise change the design; they do not cause any logic to be created or any scan routing to be inserted.

Table 6-1 Scan Specification Commands

Task	Commands
Specifying global scan properties	set_scan_configuration
Identifying nonscan sequential elements	set_scan_element set_scan_transparent
Allocating and ordering scan-path components on the scan chains	set_scan_link set_scan_segment set_scan_path
Assigning scan signals	set_scan_signal
Removing a scan specification	remove_scan_specification

You can specify as little, or as much, scan detail as you want. If you do not specify any scan detail, DC Expert *Plus* implements a default full-scan design. If you completely specify the scan design you require, you explicitly define the global scan properties and assign every scan-path component to a specific position in a specific scan chain. You also explicitly define the pins to use as scan control pins. If you issue a partial specification, DC Expert *Plus* creates a complete specification during the preview process. See the “Previewing Scan Structures” section later in this chapter for details on the default full-scan design and completing the scan specification for partially specified designs.

The following sections in this chapter discuss the scan specification tasks and commands.

Specifying Global Scan Properties

Use the `set_scan_configuration` command to specify the global properties of the scan architecture. The global properties include

- Scan methodology
- Scan style
- Number of scan chains
- Handling of multiple clocks
- Bidirectional and three-state control requirements

DC Expert *Plus* provides the following variables that affect the global properties:

test_default_scan_style

Defines the scan style used if you do not specify the `-style` option of the `set_scan_configuration` command.

test_dedicated_subdesign_scan_outs

Controls the creation of dedicated scan-output ports on subdesigns during hierarchical scan assembly.

test_disable_find_best_scan_out = false

Controls whether DC Expert *Plus* uses timing constraints to determine the scan-output pin for each sequential cell in a scan chain.

The global properties constrain all aspects of how DC Expert *Plus* creates scan chains, except specific allocation and ordering of individual scan chains. The `set_scan_path` command defines the allocation and ordering for individual scan chains.

The syntax of the `set_scan_configuration` command is

```
set_scan_configuration
    [-add_lockup true | false]
    [-bidi_mode mode]
    [-chain_count number_of_chains | default]
    [-clock_gating gating_style]
    [-clock_mixing mixing_style]
    [-dedicated_scan_ports true | false]
    [-disable true | false]
    [-existing_scan true | false]
    [-hierarchical_isolation true | false]
    [-methodology scan_methodology]
    [-multiple_test_clocks true | false]
    [-rebalance true | false]
    [-replace true | false]
    [-route true | false]
    [-route_signals all|global|serial|clock|scan_enables]
    [-style scan_style]
```

`-add_lockup true | false`

Instructs the `insert_scan` command to insert lock-up latches between clock domain boundaries on scan chains in multiplexed flip-flop designs. The `insert_scan` command inserts lock-up latches by default, so set this argument to false to disable lock-up latch insertion. The `insert_scan` command ignores this option if the scan style is not multiplexed flip-flop or the scan specification does not allow the mixing of clocks on chains. The Boolean argument is not case-sensitive.

`-bidi_mode mode`

Specifies the direction in which the `insert_scan` command configures bidirectional ports during scan shift. The following list shows the valid values for the mode argument:

- input

  Configures bidirectional ports as inputs during scan shift. This is the default value.

- output

  Configures bidirectional ports as outputs during scan shift.The `insert_scan` command ignores this option if there are no bidirectional ports or if you set the `-disable` option to false.

`-chain_count number_of_chains | default`

Instructs the `insert_scan` command to build the number of chains specified by the `number_of_chains` argument or to revert to its default behavior. By default, the `insert_scan` command builds the minimum number of scan chains consistent with clock mixing constraints. By specifying a positive integer, you instruct the `insert_scan` command to build exactly that number of chains.

`-clock_gating gating_style`

Instructs the `insert_scan` command about the kind of clock gating logic to insert for multiplexed flip-flop partial-scan designs. The following list shows the valid values for the `gating_style` argument:

- entire_design

  Inserts clock gating logic at the top level of the design only. This is the default value.

- leaf_cell

  Inserts clock gating logic throughout the design hierarchy.

- superbuffer

  Inserts a super-buffer in the scan clock tree, inserts the clock gating logic before the super-buffer, and inserts an additional super-buffer at the root of the gated clock tree.The `insert_scan` command ignores this option if the scan methodology is not partial scan or if the scan style is not multiplexed flip-flop.

`-clock_mixing mixing_style`

Specifies whether the `insert_scan` command can include cells from different clock domains in the same scan chain in multiplexed flip-flop designs. The following list shows the valid values for the `mixing_style` argument:

- no_mix

  Scan chains can contain only cells that are clocked by the same clock edge. This is the default value.

- mix_edges

  Scan chains can contain cells clocked by different edges of the same clock.

- mix_clocks

  Scan chains can contain cells clocked by different clocks.The `insert_scan` command ignores this option if the scan style is not multiplexed flip-flop.

`-dedicated_scan_ports true | false`

Specifies whether the `insert_scan` command should always create dedicated scan-output ports. By default, the `insert_scan` command does not create dedicated scan-output ports on the current design. It uses mission-mode ports as scan-output ports whenever possible. Set the option to true to require dedicated scan-output ports. The Boolean argument is not case-sensitive.

`-disable true | false`

Specifies whether the `insert_scan` command should insert three-state disabling logic and bidirectional control logic. By default, the `insert_scan` command inserts disabling logic. The option ensures that, during scan shift, three-state buses have exactly one active driver and that bidirectional port modes are configured properly. Set the option to false if you do not want to insert disabling logic. The Boolean argument is not case-sensitive.

`-existing_scan true | false`

Specifies whether the current design is an existing scan design. By default, the `insert_scan` command assumes that designs it has not processed do not have scan chains. Set the option to true if you are using the `insert_scan` command to process an imported design that already has scan chains. The Boolean argument is not case-sensitive.

`-hierarchical_isolation true | false`

Specifies whether the `insert_scan` command should gate dedicated subdesign scan-output signals with the design's scan enable signal. This gating is called hierarchical isolation logic. Hierarchical isolation logic reduces noise and power dissipation when the scan logic is inactive. By default, the `insert_scan` command does not build hierarchical isolation logic. Set the option to true if you want to insert hierarchical isolation logic. The Boolean argument is not case-sensitive.

`-methodology scan_methododology`

Selects the scan methodology. The following list shows the valid values for the `scan_methodology` argument:

- full_scan

  Includes all sequential cells that do not violate test design rules in the scan chains. This is the default value.

- partial_scan

  Trades off the area and timing penalties of full scan at the expense of testability and test-pattern generation time by leaving valid nonscan cells off scan chains.

- none

  You have not selected a scan methodology for the design.

`-multiple_test_clocks true | false`

Associates dedicated test clocks with distinct clock domains. By default, the `insert_scan` command does not use dedicated test clocks and inserts its own test clocks.

`-rebalance true | false`

Specifies whether the `insert_scan` command should respect existing subdesign scan chains when completing the scan architecture. By default, the `insert_scan` command does not modify existing subdesign scan chains. Set the option to true if you want the `insert_scan` command to split up existing scan chains (created by DC Expert *Plus*) to produce scan architectures with more-balanced scan chains. DC Expert *Plus* never modifies user-defined scan segments. The Boolean argument is not case-sensitive.

`-replace true | false`

Instructs the `insert_scan` command to perform scan replacement. By default, the `insert_scan` command replaces nonscan sequential cells that are not violated by test design rule checking with scan cells. Set the option to false if you do not want scan replacement. The Boolean argument is not case-sensitive.

`-route true | false`

By default, the `insert_scan` command routes scan signals you specify in the `-route_signals` option. If you do not use the `-route_signals` option, `insert_scan` routes all scan signals. Set the `-route` option to false if you do not want scan-signal routing.

`-route_signals all | global | serial | clock | scan_enable`

By default, routes all scan signals (scan clocks, scan enables, and serial signals such as scan_in and scan_out). Select global to route scan clocks and scan enables only.

`-style` *`scan_style`*

Selects the scan style. The following list shows the valid values for the `scan_style` argument that implement a scan design:

- multiplexed_flip_flop

  Selects the multiplexed flip-flop scan style.

- clocked_scan

  Selects the clocked scan style.

- lssd

  Selects the Level-sensitive scan design (LSSD) scan style.

- aux_clock_lssd

  Selects the auxiliary-clock LSSD scan style.

For details on these scan styles, see the *Design Compiler Reference Manual: Constraints and Timing*.

The following list shows other possible values for the scan_style argument:

- combinational

  Specifies that the design is combinational. Use this value to enable Test Compiler automatic test-pattern generation (ATPG) for combinational designs.

- none

  You have not selected a scan style for the design. By default, the `insert_scan` command uses the scan style specified by the test_default_scan_style variable in your .synopsys_dc.setup file.

Identifying Nonscan Sequential Elements

You can explicitly exclude sequential elements from the scan chains, using the `set_scan_element` command. In addition, you can select the pseudocombinational transparent latch model for nonscan latches, using the `set_scan_transparent` command.

Excluding Sequential Elements From the Scan Chain

For full-scan designs, the default scan architecture includes all valid sequential cells in the scan chain. You can use the `set_scan_element` false command to exclude sequential elements from the scan chain. The `set_scan_element` command places the `scan_element` attribute on the specified sequential elements. If a cell has a `scan_element` attribute of false, DC Expert *Plus* does not perform scan replacement on the cell or include the cell in a scan chain.

The syntax of the `set_scan_element` command is

```
set_scan_element
     true | false
     cell_design_ref_list
```

`true | false`

A Boolean value that determines whether or not to include sequential elements in a scan chain. This value is not case-sensitive.

`cell_design_ref_list`

A list of sequential elements, such as

- Cells (flip-flops or latches)
- Hierarchical cells (containing flip-flops or latches)
- References
- Library cells
- Designs

Identifying Transparent Latches

Use the `set_scan_transparent` command to select the pseudocombinational transparent latch model for test design rule checking and ATPG. The `set_scan_transparent` command places the `scan_transparent` attribute on the specified latch elements. If an element has a `scan_transparent` attribute of true, DC Expert *Plus* uses the pseudocombinational transparent latch model during test design rule checking and ATPG.

The syntax of the `set_scan_transparent` command is

```
set_scan_transparent true | false cell_design_ref_list
-existing
```

`true | false`

A Boolean value that determines whether or not to use the pseudocombinational transparent latch model for the specified latch elements during test design rule checking and ATPG. This value is not case-sensitive.

cell_design_ref_list

A list of latch elements, such as

- Latch cells
- Hierarchical cells (containing latches)
- References
- Library cells
- Designs

`-existing`

An option that indicates that DC Expert *Plus* does not need to add logic to force a latch transparent.

Allocating and Ordering Scan Chains

Use the `set_scan_path` command to allocate and order the scan-path components on each scan chain. Before specifying the allocation and ordering, you can define scan links or scan segments you will use in the specification.

Defining Scan Links

Use the `set_scan_link` command to explicitly define a scan link or to override a default scan link. To see the default scan links, preview the scan specification. The scan specification report shows the location of lock-up latch links. If the scan specification report does not show a link, the default scan link is a wire. You cannot control the insertion of multiplexer links.

The syntax of the `set_scan_link` command is

```
set_scan_link
     scan_link_name
     wire | scan_out_lockup
```

`scan_link_name`

Gives the scan link a name.

`wire | scan_out_lockup`

Specifies the scan link type. This can be a wire or a lock-up latch that retimes internal scan-output signals.

Defining Scan Segments

DC Expert *Plus* supports inferred scan segments and user-defined scan segments.

An inferred scan segment (also called a subdesign scan chain) is an existing scan structure that uses subdesign ports for all scan signals. DC Expert *Plus* infers subdesign scan chains during execution of the `check_test` command.

A user-defined scan segment is an existing scan structure that uses internal pins for one or more scan signals. Use the `set_scan_segment` command to identify user-defined scan segments.

For details on using scan segments, see the *Scan Synthesis User Guide*.

The syntax of the `set_scan_segment` command is

```
set_scan_segment
     scan_segment_name
     [-access signal_type_pin_pair_list]
     [-contains member_list]
```

`scan_segment_name`

Gives the scan segment a name. You use this name to identify the scan segment during scan chain allocation and ordering.

`-access signal_type_pin_pair_list`

Instructs the `insert_scan` command how to access the segment. Consists of a list of ordered pairs, each consisting of a scan signal type and a design pin. Valid signal types include test_clock, test_scan_clock, test_scan_clock_a, test_scan_clock_b, test_scan_enable, test_scan_enable_inverted, test_scan_in, and test_scan_out. Validation ensures that specified signal types are consistent with the design's scan style.

`-contains member_list`

Defines scan segment components. This ordered list can include sequential cells and design instances. You can use wildcards and dc_shell command expressions such as all_registers(), filter(), and find() to identify design objects. You must explicitly specify each component of the scan segment.

DC Expert *Plus* includes user-defined scan segments in scan chains by connecting their access pins but does not validate user-defined scan segments before insertion. If you do not completely specify a user-defined scan segment, DC Expert *Plus* can create an invalid scan architecture.

If the test_scan_in access pin is functionally connected, the `insert_scan` command inserts a multiplexer before the test_scan_in access pin. The `insert_scan` command does not recognize a pin as functionally connected if the pin:

- Is not connected to a net
- Is connected to a net that is held at a constant logic value
- Was created with the `insert_scan` command
- Is a driver and does not have a load, or is connected to a subdesign output port created with the `insert_scan` command
- Is a load and does not have a driver, or is connected to a subdesign input port created with the `insert_scan` command

Specifying Scan Chains

Use the `set_scan_path` command to specify a scan chain for the current design. The `set_scan_path` command allocates scan-path components (scan elements, scan segments, and scan links) to scan chains and specifies scan chain orderings. Subject to ordering consistency, you can define any ordering of scan links and scan elements in a scan chain. You can define the order so that a scan chain has multiple reentries into a level of hierarchy.

A scan-path component cannot belong to more than one chain. Where `set_scan_path` commands conflict, DC Expert *Plus* uses the most recent command.

The syntax of the `set_scan_path` command is

```
set_scan_path
     scan_chain_name
     [ordered_list]
     [-dedicated_scan_out true | false]
     [-complete true | false]
```

`scan_chain_name`

Gives the scan chain a name.

`ordered_list`

Defines the scan-path components. This ordered list can include scan elements, scan segments, and scan links. You can use wildcards and dc_shell command expressions such as all_registers(), filter(), and find() to identify design objects.

`-dedicated_scan_out true | false`

Specifies the requirement for a dedicated scan-output port for the chain. By default, if the last cell in a scan chain drives an output port, the `insert_scan` command uses this port as the scan-output port. Set the option to true if you want the `insert_scan` command to always build a dedicated scan-output port for the chain.

`-complete true | false`

Specifies whether DC Expert *Plus* can add more components to the chain. By default, DC Expert *Plus* might add components to a specified scan chain. Set the option to true to indicate to DC Expert Plus that the chain specification is complete and additional components must not be added.

Identifying and Assigning Scan Signals

DC Expert *Plus* can use existing design ports to transmit scan signals. If you want to control the allocation of existing design ports to scan signals, use the `set_scan_signal` command. You can assign each scan port to a specific scan chain, or you can allow DC Expert *Plus* to assign the port.

The `set_scan_signal` command places a `scan_signal` attribute on the specified design port. DC Expert *Plus* first uses ports with `scan_signal` attributes to complete the scan chains. If unassigned scan signals remain, DC Expert *Plus* completes the scan architecture by using the port selection process described in the “Determining Scan Ports” section later in this chapter.

Note:

The `set_signal_type` command identifies scan port to scan signal connections in existing scan designs and does not affect the determination of the scan architecture for nonscan or prescan designs.

The syntax of the `set_scan_signal` command is

```
set_scan_signal
    test_signal_type
    -port port_list
    [-chain scan_chain_list]
    [-hookup pin_name [-sense inverted | non_inverted]]
```

`test_signal_type`

Describes the type of the signal. See Table 6-2 for valid values. When you run the `set_scan_signal` command, DC Expert *Plus* ensures that signal types are consistent with the design's scan style.

`-port port_list`

Identifies design ports used to transmit signals of the specified type. When you run the `set_scan_signal` command, DC Expert *Plus* ensures that the port direction is consistent with the signal type (see Table 6-2).

`-chain scan_chain_list`

Identifies the scan chain connected to this scan signal. By default, the `insert_scan` command assigns scan signals to scan chains. This option lets you make the assignment.

`-hookup pin_name`

Identifies the connection pin. By default, the `insert_scan` command connects wires to the core side of identified signal ports, jumping pads and buffers as needed. The `-hookup` option overrides this behavior and instructs the `insert_scan` command to connect wires to a specific pin. When you run the `set_scan_signal` command, DC Expert *Plus* ensures that the pin direction is consistent with the signal type. You can associate a specific access pin with only one design port.

```
-sense inverted | non_inverted
```

Describes whether the logic on the path between the port and the hookup pin inverts the signal sense. By default, the sense is noninverted.

Table 6-2 Valid Values for the test_signal_type Argument

Scan Signal	**Attribute Value**	**Applicable Scan Styles**	**Port Direction**
scan input	test_scan_in	all	input bidirectional
scan output	test_scan_out	all	output three-state bidirectional
scan enable	test_scan_enable or test_scan_enable_inverted	multiplexed_flip_flop	input bidirectional*
test clock	test_clock	aux_clock_lssd	input
a scan clock	test_scan_clock_a	lssd, aux_clock_lssd	input
b scan clock	test_scan_clock_b	lssd, aux_clock_lssd	input
test scan clock	test_scan_clock	clocked_scan	input

*Not recommended: Complex methodologies required.

Example

If you enter

```
dc_shell> set_scan_signal test_scan_in -port scan_in
```

DC Expert *Plus* responds with

```
Performing set_scan_signal on port ‘scan_in’.
```

Where scan_in is the name of the scan-input port that DC Expert *Plus* uses.

Removing Scan Specifications

Use the `remove_scan_specification` command to remove scan specifications from the current design. This command deletes specifications made with the `set_scan_configuration`, `set_scan_link`, `set_scan_path`, `set_scan_segment`, and `set_scan_signal` commands. You can use the `-all` option to delete all such specifications and use other command options to delete specific specifications.

The `remove_scan_specification` command does not change your design. The command deletes specifications you have made.

The syntax of the `remove_scan_specification` command is

```
remove_scan_specification
     [-all]
     [-chain chain_name_list]
     [-configuration]
     [-link link_name_list]
     [-segment segment_name_list]
     [-signal port_name_list]
```

`-all`

Removes all `set_scan_path`, `set_scan_configuration`, `set_scan_link`, `set_scan_segment`, and `set_scan_signal` command specifications from the current design.

`-chain chain_name_list`

Removes all specifications that reference scan chains listed in the `chain_name_list` argument. Removes associated `set_scan_path` command specifications from the current design and updates `set_scan_signal` command specifications.

`-configuration`

Restores `set_scan_configuration` command options to default values.

`-link` *link_name_list*

Removes all specifications that reference scan links listed in the `link_name_list` argument. Removes associated `set_scan_link` command specifications from the current design and updates `set_scan_path` command specifications.

`-segment` *segment_name_list*

Removes all specifications that reference scan segments listed in the `segment_name_list` argument. Removes associated `set_scan_segment` command specifications from the current design and updates `set_scan_path` command specifications.

`-signal` *port_list*

Removes all specifications that reference scan signals associated with the design ports listed in the `port_list` argument. Removes associated `set_scan_signal` command specifications from the current design.

Previewing Scan Structures

The `preview_scan` command generates a scan architecture definition that satisfies the scan specifications on the current design and reports the scan architecture for you to preview. If you do not like the proposed implementation, you can iteratively adjust and preview the specification until you are satisfied with the proposed design. This allows you to preview your scan designs without synthesizing and to change your scan specification to explore the design space as necessary.

The syntax for the `preview_scan` command is

```
preview_scan
[-script]
[-show cells|scan|scan_clocks|scan_signals|segments|all]
```

`-script`

Instructs the `preview_scan` command to produce a dc_shell script that specifies the defined scan design. The script is written to standard output.

`-show cells|scan|scan_clocks|scan_signals|segments|all`

Instructs the `preview_scan` command to report information of various types, in addition to the summary report it produces by default. The option takes a list of case-insensitive arguments in cells, scan, segments, scan_clocks, and scan_signals or the keyword all.

The preview_scan command performs the following tasks:

- Checks test design rules
- Validates your scan specification
- Completes the scan architecture
- Reports the resulting scan architecture
- Generates a dc_shell script

The following sections discuss each of these tasks.

Checking Test Design Rules

If you did not run test design rule checking before running the `preview_scan` command, the `preview_scan` command invokes test design rule checking to identify valid (nonviolated) scan cells. For information about test design rule checking, see the *Scan Synthesis User Guide*.

Validating Your Scan Specification

A valid scan specification must meet the following conditions:

- The scan specification is consistent.

  Two scan specification commands must not impose mutually exclusive conditions on the scan design. For example, DC Expert *Plus* does not support the creation of complex test and maintenance architectures where individual scan cells are members of more than one chain. Therefore, two specification commands that place the same element in two different scan chains are mutually incompatible.

- The scan specification yields a functional scan design.

  You cannot impose a specification that leads to a nonfunctional scan design. For example, if your specification for a multiplexed flip-flop design does not allow clock mixing, DC Expert *Plus* requires one scan chain per clock domain. In this situation, a specification that mandates fewer scan chains than the number of clock domains is invalid.

Completing the Scan Architecture

The preview_scan command proposes a scan architecture, including

- The number of scan chains to be implemented
- The allocation of scan cells to scan chains
- The ordering of scan cells within a scan chain
- The type of scan links to use between scan cells
- The design ports used as scan ports

Determining the Number of Scan Chains

Figure 6-3 shows the process DC Expert *Plus* uses to determine the number of scan chains in your design. Figure 6-4 shows the process for determining minimum chain count.

Figure 6-3 Determining the Number of Scan Chains

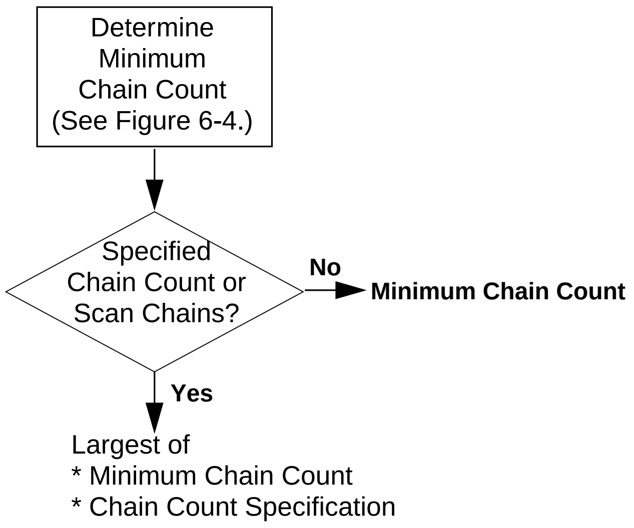
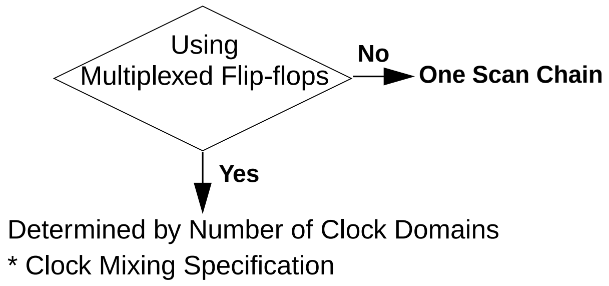

Determine Minimum Chain Count (See Figure 6-4.)
Specified Chain Count or Scan Chains?
No
Minimum Chain Count
Yes
Largest of
* Minimum Chain Count
* Chain Count Specification

Figure 6-4 Determining Minimum Chain Count


Using Multiplexed Flip-flops
No
One Scan Chain
Yes
Determined by Number of Clock Domains
* Clock Mixing Specification

For multiplexed flip-flop designs, DC Expert *Plus* creates one scan chain per clock domain by default. When performing scan assembly, DC Expert *Plus* considers each edge of the clock as a separate clock domain. Although you increase the risk of timing problems, you can reduce the number of scan chains by allowing mixing of edges or clocks within a scan chain. Use the `set_scan_configuration -clock_mixing` command to define the clock mixing specification. If you request fewer scan chains than the minimum chain count, DC Expert *Plus* generates the following warning message during `preview_scan` execution:

```
Warning: Cells with %n new incompatible clock domains have not been assigned to
scan chains. Cannot honor -chain_count specification of %n. Try using
set_scan_configuration -clock_mixing mix_edges or -clock_mixing mix_clocks.
(TEST-355)
```

Allocating Scan Chains

If your design has multiple scan chains, DC Expert *Plus* uses the following criteria (in priority order) to allocate cells to scan chains:

1. Have you specified an allocation?

   If you specify an allocation using the `set_scan_path` command, DC Expert *Plus* follows this allocation.

For multiplexed flip-flop designs, DC Expert *Plus* generates the following warning message if your specified allocation violates the clock domain requirement:

```
Warning: Chain '%s' has elements clocked by different clocks. (TEST-353)
```

DC Expert *Plus* creates the requested scan chain, inserting a scan lock-up latch between clock domains unless you have disabled scan lock-up latch insertion.

2. What clock does the scan cell use?

   For multiplexed flip-flop designs, DC Expert *Plus* allocates cells to scan chains based on clock domain, unless you have explicitly allowed mixing of edges or clocks. If your design has more clock domains than the number of scan chains you have requested, DC Expert *Plus* generates the following warning message during preview_scan execution:

   ```
   Warning: Cells with %n new incompatible clock domains
   have not been assigned to scan chains. Cannot honor
   -chain_count specification of %n. Try using
   set_scan_configuration -clock_mixing mix_edges or
   -clock_mixing mix_clocks. (TEST-355)
   ```

3. Are the scan chains balanced?

   By default, DC Expert *Plus* balances the number of cells in each scan chain, but it might generate unbalanced scan chains in the following situations:

   - You specify complete scan chains by using the `set_scan_path` command.
   - A multiplexed flip-flop design has multiple clock domains.
   - Your design contains scan segments.

Ordering Scan Chains

DC Expert *Plus* uses the following criteria (in priority order) to order cells within a scan chain:

1. Will the scan chain shift properly?

   DC Expert *Plus* uses ideal test clock timing (ideal clock timing does not consider clock skew) to determine whether scan shift occurs properly. If you do not explicitly define the test clock timing, DC Expert *Plus* uses the default values. See the *Scan Synthesis User Guide* for more information about test clock timing.

   If the scan chain has multiple edges or clocks, DC Expert *Plus* orders the cells so cells clocked later in the cycle are placed earlier in the scan chain.

2. Have you provided explicit routing specifications?

   If your routing specification meets the scan shift requirement, DC Expert *Plus* follows your explicit routing specifications. To shift properly, each cell pair must meet the following criteria: The active clock edge of the next scan cell must occur concurrently with or before the active clock edge of the current scan cell (or can be synchronized with a scan lock-up latch). If a cell pair violates this requirement, DC Expert *Plus* reorders the scan chain.

3. Does the design have multiple clock domains?

   If the design has multiple clock domains and clock mixing is supported, DC Expert *Plus* orders the cells to minimize the number of clock domain crossings within a scan chain.

In the absence of explicit routing specifications, DC Expert *Plus* places the elements in each scan chain in alphanumeric order using the full hierarchical path name of the element. The only exception occurs when the output of a scan cell directly drives an output port in the current design. In such a case, DC Expert *Plus* uses this cell as the last cell in the scan chain, regardless of its instance name.

Determining Scan Links

In most cases, DC Expert *Plus* uses nets (wires) as scan links.

DC Expert *Plus* uses scan lock-up latches as scan links in the following situations:

- You explicitly specify scan lock-up latches as scan links, using the `set_scan_link` and `set_scan_path` commands.
- Your multiplexed flip-flop design changes clock domains between scan cells, and you have not disabled scan lock-up latch insertion.

DC Expert `Plus` uses multiplexers as scan links in the following situations:

- A signal path does not exist between the last scan-output pin and the scan-output port.
- A signal path does not exist between the previous scan-output pin and the scan-input pin of a scan segment.

Note:

> The preview_scan report does not indicate the presence of multiplexer scan links.

A scan link connects the scan-output pin on the current cell to the scan-input pin of the next cell (or the scan-output port). DC Expert *Plus* uses the following process to determine the scan-output pin:

- If the functional path through the scan cell has timing constraints, DC Expert Plus selects the scan-output pin with the most timing slack. In cae of a tie, the tool selects the pin with the greatest drive strength.

  To disable this behavior, set the test_disable_find_best_scan_out variable to true.

- If the functional path through the scan cell does not have timing constraints, DC Expert *Plus* selects the noninverted scan-output pin.

Determining Scan Ports

A scan design requires a scan-input port and a scan-output port for each scan chain, as well as the scan-control ports appropriate for the selected scan style. The following sections describe how DC Expert Plus determines the scan ports.

Determining the Scan-Input Port

The `insert_scan` command executes the following steps to determine the scan-input port:

1. Checks for ports with the `test_scan_in scan_signal` attribute (specified with the `set_scan_signal` command)
2. Checks for ports with `test_scan_in signal_type` attributes (specified with the `set_signal_type` command)
3. Creates a dedicated scan-input port

Determining the Scan-Output Port

The `insert_scan` command executes the following steps to determine the scan-output port:

1. Checks for ports with the `test_scan_out scan_signal` attribute (specified with the `set_scan_signal` command)
2. Checks for ports with `test_scan_out signal_type` attributes (specified with the `set_signal_type` command)
3. Checks for an output port directly driven by a sequential cell
4. Creates a dedicated scan-output port

Determining the Scan-Control Port

When generating the scan architecture, DC Expert *Plus* assigns a single port to drive each scan-control signal required by the selected scan style. Table 6-3 shows the scan-control signal requirements for each supported scan style.

Table 6-3 Scan-Control Signals

Scan Style	**Scan-Control Signals**	**Attribute Value**
Multiplexed flip-flop	scan enable	scan_enable
Clocked scan	test scan clock	test_scan_clock
LSSD	a scan clock b scan clock	test_scan_clock_a test_scan_clock_b
Auxiliary clock LSSD	a scan clock b scan clock test clock	test_scan_clock_a test_scan_clock_b test_clock

The `insert_scan` command executes the following steps to determine the scan-control ports:

1. Checks for a port with the appropriate `scan_signal` attribute (specified with the `set_scan_signal` command)
2. Checks for a port with appropriate signal_type attribute (specified with the `set_signal_type` command)
3. Creates a dedicated scan-control port

Note:

DC Expert *Plus* uses the existing slave clock as the b scan clock for all LSSD master-slave latch equivalents.

Reporting the Scan Architecture

If the scan architecture reported by the `preview_scan` command does not meet your requirements, modify your scan specification as discussed in the “Specifying the Scan Architecture” section earlier in this chapter.

Example

To generate a scan architecture summary report, enter the following command:

```
dc_shell> preview_scan
```

The result is the following report:

```
****************************************
Preview scan report
Design: P
Version: 1997.08
Date: Fri May 9 11:25:53 1997
****************************************

Number of chains: 1
Test methodology: full scan
Scan style: multiplexed_flip_flop
Clock domain: no_mix
Scan chain ‘1’ (test_si -> test_so) contains 4 cells
```

Generating a dc_shell Script

Use the `-script` option to generate a dc_shell script that completely specifies the resulting scan architecture. You can use this script to

- Modify the scan architecture.

  One method of modifying the scan architecture is to edit and run the generated script.

- Annotate the design database with the complete scan specification.

  The `preview_scan` command does not annotate the database. However, you can run the generated script to annotate the database with the completed scan specification.

Inserting and Optimizing Scan Structures

Use the `insert_scan` command to assemble the scan architecture. The `insert_scan` command inserts and optimizes the scan structures defined in the scan architecture, using design constraints to guide the optimization. See the *Design Compiler Reference Manual: Constraints and Timing* for information about specifying design constraints. The `insert_scan` command does not use timing values specified by the `create_test_clock` command for optimization. The `insert_scan` command supports hierarchical scan insertion. You can use top-down, bottom-up, or middle-out scan insertion methodologies.

The syntax for the `insert_scan` command is

```
insert_scan
    [-map_effort low | medium | high]
    [-ignore_compile_design_rules]
```

`-map_effort low | medium | high`

Specifies the relative amount of CPU time spent during the mapping phase of scan insertion. The default is medium. Table 6-4 shows the effect of each of the map effort settings.

Table 6-4 insert_scan -map_effort Settings

Map Effort	Description
low	Performs critical path optimizations on logic added to the design by the `insert_scan` command.
medium	Performs low-effort optimizations plus critical path optimizations on the entire design. This is the default value.
high	Performs medium-effort optimizations plus sequential mapping on nonscan sequential cells on critical paths. Also tries to reduce design area by eliminating inverters and downsizing cells off the critical paths.

`-ignore_compile_design_rules`

Instructs the `insert_scan` command to exit before correcting compile design rule violations.

This option allows you to check the results in a constraint report before correcting the violations. By default, the `insert_scan` command corrects compile design rule violations (for example, max_transition, max_fanout, max_capacitance, and min_path) and performs mapping optimizations before exiting.

Example

To assemble the scan chains in design WC66, enter the following commands

```
dc_shell> current_design WC66
dc_shell> insert_scan
```

Controlling the Behavior of the insert_scan Command

DC Expert *Plus* provides several variables that control the behavior of the `insert_scan` command. The following list defines these variables and their default values:

compile_fix_multiple_port_nets = false

Defines whether the `insert_scan` command adds extra logic to prevent multiple output ports from being connected to the same net.

test_dont_fix_constraint_violations = false

Defines whether the `insert_scan` command ignores optimization constraints.

insert_test_design_naming_style = %s_test_%d

Defines the naming convention for subdesigns modified during the scan assembly process. The `insert_scan` command does not change the name of the current design.

test_clock_port_naming_style = test_c%s

Defines the naming convention for test clock ports created during the scan assembly process.

test_scan_clock_a_port_naming_style = test_sca%s

Defines the naming convention for test master clock ports created during the scan assembly process.

test_scan_clock_b_port_naming_style = test_scb%s

Defines the naming convention for test slave clock ports created during the scan assembly process.

test_scan_clock_port_naming_style = test_sc%s

Defines the naming convention for test scan clock ports created during the scan assembly process.

test_scan_enable_port_naming_style = test_se%s

Defines the naming convention for scan enable ports created during the scan assembly process.

test_scan_enable_inverted_port_naming_style = test_sei%s

Defines the naming convention for inverted scan enable ports created during the scan assembly process.

test_scan_in_port_naming_style = test_si%s%s

Defines the naming convention for scan-input ports created during the scan assembly process.

test_scan_out_port_naming_style = test_so%s%s

Defines the naming convention for scan-output ports created during the scan assembly process.

Preparing for Scan Assembly

Before running the `insert_scan` command, you must

- Set the design constraints
- Specify the scan architecture
- Specify the test configuration

Setting Design Constraints

If you have not already set the design constraints, you should set constraints before running the `insert_scan` command, because DC Expert *Plus* minimizes constraint violations. For information about setting constraints on your design, see the *Design Compiler Reference Manual: Constraints and Timing*.

Avoiding the Default Scan Architecture

Unless you want the default scan architecture generated by DC Expert *Plus*, you must specify the scan architecture before running the `insert_scan` command. For information about the default scan architecture and how to specify a scan architecture, see the “Specifying the Scan Architecture” section earlier in this chapter.

Defining a Test Configuration

If your design requires a static test configuration to satisfy design rules or to enable circuit paths, you must define a test mode using the `set_test_hold` command. For more information, see the information in the *Scan Synthesis User Guide* about defining a test protocol.

Understanding the Scan Assembly Process

The `insert_scan` command performs the following tasks as it assembles the scan architecture:

- Checks test design rules
- Generates the scan architecture
- Performs scan replacement
- Inserts disabling logic for three-state buses
- Inserts control logic for bidirectional ports
- Assembles scan chains
- Optimizes the scan design

The following sections describe these tasks.

Checking Test Design Rules

Whenever you modify your design or the test protocol, the check_test information becomes out of date. If the check_test information is out of date (or if you have never run test design rule checking), the `insert_scan` command invokes test design rule checking. Test design rule checking infers a test protocol and violates cells that cannot belong to scan chains. For information about test protocols and test design rule checking, see the *Scan Synthesis User Guide*.

Generating the Scan Architecture

If you have modified the scan specification since you last invoked the preview_scan command (or if you have never run the `preview_scan` command), the `insert_scan` command completes the scan architecture. For information about specifying a scan architecture, see the “Specifying the Scan Architecture” section earlier in this chapter.

Performing Scan Replacement

If your design contains nonscan sequential cells that do not have test design rule violations or attributes that prevent scan replacement, the `insert_scan` command replaces these cells with their scan equivalents. For information about scan replacement, see “Using Test-Ready Compile” in Chapter 3.

To prevent scan replacement on specific cells, use the `set_scan_element false` command. To prevent scan replacement on the current design and all subdesigns, use the `set_scan_configuration -replace false` command.

Caution!

> Be careful when using the `dont_touch` attribute on sequential cells or designs. This attribute also prevents scan replacement.

Disabling Three-State Buses

If your design contains three-state buses that can generate a bus contention or bus float condition during scan shift, the `insert_scan` command inserts disabling logic. The `insert_scan` command does not insert disabling logic in the following cases:

- A bus driver belongs to a design that has a `dont_touch` attribute.
- The `insert_scan` command could not disable any of the three-state drivers.
- The three-state drivers on a bus have conflicting disabling requirements.
- The disabling requirements conflict with those of other buses.

After identifying candidate buses, the `insert_scan` command iterates through all three-state bus drivers and computes the logic values that must be applied to input pins to configure the bus during scan shift. The `insert_scan` command inserts generic disabling logic that uses the scan enable signal to apply the necessary logic values. The `insert_scan` command maps this generic logic during the optimization phase.

To prevent the insertion of three-state disabling logic, use the `set_scan_configuration -disable false` command.

Configuring Bidirectional Ports

If your design contains bidirectional ports, the `insert_scan` command inserts logic to control the mode of these ports during scan shift. The `insert_scan` command does not add control logic in the following cases:

- The bidirectional port drives the scan enable signal.
- The bidirectional port is configured as an input or output by constant logic values (degenerated bidirectional port).
- The bidirectional port has exactly one three-state driver, and the design that contains the driver has a `dont_touch` attribute.
- The configuration requirements conflict with those of other bidirectional ports.

After identifying candidate ports, the `insert_scan` command determines the mode for these ports during scan shift. The scan signal type and the bidirectional mode specification determine the mode. By default, the `insert_scan` command places bidirectional ports in input mode during scan shift. The `insert_scan` command places bidirectional ports in output mode only if the bidirectional port is a scan output port or you have specified -bidi_mode output, using the `set_scan_configuration` command.

The `insert_scan` command computes the logic values that must be applied to input pins to configure each port during scan shift and inserts generic configuration logic that uses the scan enable signal to apply the necessary values. The `insert_scan` command maps this generic logic during the optimization phase.

To prevent the insertion of bidirectional control logic, use the `set_scan_configuration -disable false` command.

Assembling Scan Chains

To assemble the scan chains, the `insert_scan` command connects the scan chain components and the global scan signals.

For each scan chain, the scan chain components are of

- Scan-input port
- Scan cells
- Scan connection elements, such as scan lock-up latches and multiplexers
- Scan-output port

The global scan signals include

- Scan-enable signals
- Test clock signals

The scan specification and preview processes determine the actual components and the ordering of the components for each scan chain. For information about the specification process, see the “Avoiding the Default Scan Architecture” section earlier in this chapter. For information about the preview process, see the “Previewing Scan Structures” section earlier in this chapter.

When connecting scan chain components, the `insert_scan` command jumps over input pads, single-fanout buffers, and single fanout inverters that occur between two components. When jumping over inverting cells, DC Expert *Plus* scan toggles the polarity of the expected scan-output results generated by Test Compiler ATPG.

The `insert_scan` command does not touch optional methodology signal pins that are functionally connected. It reconnects mandatory methodology signal pins that are functionally connected and generates a warning.

To prevent assembly of the scan chains, use the `set_scan_configuration -route false` command.

Optimizing the Scan Design

The `insert_scan` command performs the following mapping optimizations on the scan design:

- Generic-logic mapping
- Critical path optimizations
- Nonscan replacement

Generic-Logic Mapping

If your scan architecture included scan connection elements, three-state disabling logic, or bidirectional control logic, the `insert_scan` command inserted generic logic for these components. DC Expert *Plus* maps this generic logic during the optimization phase of the `insert_scan` command execution.

Critical-Path Optimizations

If your design has constraints, the `insert_scan` command performs the following critical path optimizations on the scan design:

- Sizing one or more drivers or loads for appropriate drive strength
- Replacing the driver with one that has an opposing phase

- Inserting buffers or inverter pairs: transforming inverter pairs to buffers, buffers to inverter pairs, or buffer inverters
- Downsizing and swapping out loads
- Isolating noncritical loads, using inverter pairs or buffers
- Off loading noncritical loads to other driver outputs by using a driver output with a greater drive strength, using drivers with more outputs, or moving loads to nets with earlier arrival times
- Balancing by redistributing loads among drivers or balance buffers or by balancing loads across driver outputs
- Splitting duplicate drivers, distributing loads, or dedicating one driver to the critical path

The scope of the optimization depends on the `-map_effort` option setting. See the command syntax information in the “Inserting and Optimizing Scan Structures” section earlier in this chapter for details on the `-map_effort` option.

Nonscan Replacement

If you used test-ready compile, your design might contain unrouted scan cells that have test design rule violations or the `scan_element false` attribute. In this case, the `insert_scan` command replaces these cells with their nonscan equivalents.

Saving the Scan Designs

The `insert_scan` command creates scan versions of the subdesigns and top-level design and stores these new designs in the database. DC Expert *Plus* names the new subdesigns according to the value of the `insert_test_design_naming_style` variable but does not change the name of the current design.

DC Expert *Plus* creates these scan designs and stores them in memory. Use the `write` command to save these new designs to disk.

Understanding the insert_scan Output

The `insert_scan` command generates status information as it runs. The status information includes

- The current task being performed

  The "Understanding the Scan Assembly Process" section earlier in this chapter describes the tasks performed by the `insert_scan` command. As the `insert_scan` command starts each task, DC Expert *Plus* generates a status message including.

- The optimization cost function status

  See the *Design Compiler Reference Manual: Constraints and Timing* for information about the optimization cost function.

- The design update information

  DC Expert *Plus* generates a message indicating the name of each scan design it created.

Example 6-1 shows the output from a sample `insert_scan` run.

Example 6-1 insert_scan Status Information

```
Loading design 'WC66'
Checking test design rules
Architecting Scan Chains
Inserting Scan Cells
Routing Scan Chains
Routing Global Signals
Mapping New Logic
Beginning Mapping Optimizations

        OPTIMIZATIONDESIGN RULE
TRIALS    AREA   DELTA DELAY   COST   COST
-------- ------  -----------  ------  ------
108      506.0     0.93        544.0    0.0
186      499.0     0.90        538.4    0.0
 91      483.0     0.87        521.7    0.0
. . .
209      446.0     0.00        505.0    0.0
------
2684


Transferring design 'MOORE' to database 'MOORE.db'
Transferring (new) design 'CHARLTONS_test_1' to database 'CHARLTONS.db'
Transferring (new) design 'STILES_test_1' to database 'STILES.db'
Transferring (new) design 'HURST_test_1' to database 'HURST.db'
Transferring design 'WC66' to database 'WC66.db
```

7

Adding I/O Pad Cells

I/O pad cells are necessary to connect core logic to an external environment. These cells contain logic and attributes that facilitate their interface applications.

I/O pad cell logic ranges in complexity from a simple buffer to special scan cells containing flip-flops and combinational logic. During design optimization, core logic can be brought into the I/O pad cells. The I/O pad cell logic also provides boundary-scan test and design partitioning capabilities.

You can add pads in two ways:

- Instantiate through HDL (preferred method)

  Instantiation allows explicit control of pad cells. Instantiate I/O pads after the core is implemented and simulated and before floorplanning.

- Synthesize with Design Compiler (alternative method)

  If the vendor library contains pad cells, you can use Design Compiler to insert the pad cells automatically. Design Compiler does not map to pad cells during regular compilation if the pad cells have required attributes. Otherwise, set the `dont_touch` attribute on the pad cells during compilation.

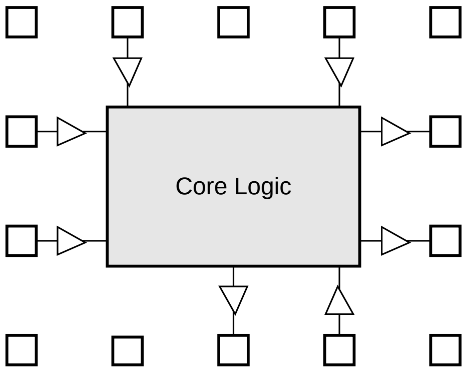
This chapter describes how to add pad cells through synthesis, in the following sections:

- Determining Availability of Library Pad Cells
- Creating I/O Pads
- Describing I/O Pad Characteristics
- Inserting I/O Pad Cells

I/O pad insertion adds I/O buffers to the primary inputs and outputs of a design. Instantiated I/O pads have the characteristics (voltage levels, current levels, presence of pull-ups and pull-down resistors, slew-rate control, and so forth) that you requested. I/O pad optimization can modify previously inserted I/O pads to meet constraints. I/O pad optimizations can include sizing and can incorporate core logic functionality into the pads.

Figure 7-1 shows a sample design.

Figure 7-1 Design Having I/O Pads



Input and output buffers are added to each port in the top-level design. The buffers are sized to meet the port-to-port timing constraints when the delays through the core logic are known.

I/O buffers consume significant current; therefore, gate-level optimization selects the smallest I/O buffer that meets the timing specification of the design.

Determining Availability of Library Pad Cells

Before you can insert pad cells, determine whether the technology library contains appropriately modeled pad cells and determine which pin on the pad has the `is_pad` attribute.

The `pad_cell` attribute must be set to true, and one or more pins of the pad cell must have the `is_pad` attribute set.

Example

To determine whether the technology library contains pad cells, enter

```
filter find(cell,libA.db/*) "@pad_cell == true"
```

To determine which pin on the pad cell has the `is_pad` attribute (the previous `filter find` command returned the pad cell padA), enter

```
filter find (pin,"libA.db/padA/*") "@is_pad == true"
```

If Design Compiler cannot find the library and issues a message, use the `list -libraries` command to list the system file name and the library name.

Creating I/O Pads

The `set_port_is_pad` command sets the `port_is_pad` attribute on specified ports to indicate that you want I/O pads inserted on the ports.

The `set_port_is_pad` command is required for inserting I/O pads; if it is omitted, no pads are inserted. I/O pad cells are inserted and optimized only at ports that have the `port_is_pad` attribute.

The syntax is

```
set_port_is_pad [list_of_ports_or_designs]
```

If you specify a design or a list of designs, every port in each design becomes an I/O pad.

If you use `set_port_is_pad` with no argument, Design Compiler sets the attribute on all the ports in the current design.

For bidirectional ports, set the attribute once as either an input port or an output port.

Describing I/O Pad Characteristics

The `set_pad_type` command describes the requirements for I/O pads.

Use `set_pad_type` to control the kind of pad cell inserted. For greater control, use the `-exact` option.

By setting I/O pad cell attributes, you can define many characteristics associated with typical pad cells, such as

- Pull-up or pull-down resistors
- Schmitt triggers
- Slew-rate control
- Open drain or open source
- Voltage levels
- Output current level

The syntax is

```
set_pad_type [-example example_pad] [-exact exact_pad]
    [-schmitt] [-pullup] [-pulldown] [-slewrate value]
    [-open_drain]  [-open_source]
    [-vih value]  [-vilvalue]
    [-vol value] [-voh value]
    [-vimin value] [-vimax value]
    [-vomin value] [-vomax value] [-clock] [-no_clock]
    [-currentlevel value] design_or_port_list
```

`-example` *example_pad*

Defines the nonfunctional characteristics of the pad by inference from the library information for the example pad. Use `-example` to define the characteristics of a pad without explicitly listing each attribute.

`-exact` *exact_pad*

Defines the exact I/O pad cell to be instantiated at the listed ports. Because `-exact` defines an I/O cell, no optimization is performed on the target ports. Use this option with care, and only with pads that are inverters, buffers, or simple three-states.

`-schmitt`

Specifies that the input or inout pad needs a Schmitt trigger.

`-pullup`

Specifies that the pad has a pull-up resistor.

`-pulldown`

Specifies that the pad has a pull-down resistor.

`-slewrate [none | low | medium | high]`

Defines slew-rate control for output and inout ports. If you omit `-slewrate`, no slew rate control is implemented.

The low, medium, and high values are optional. Use a value when there are different gradations of slew-rate control. If you omit a value, the slew-rate defaults to high.

`-open_drain`

Specifies that the output or inout pad has an open drain.

`-open_source`

Specifies that the output or inout pad has an open source.

`-vil value`

Defines Vil on input or inout pads.

`-vih value`

Defines Vih on input or inout pads.

`-vol value`

Defines required Vol on output or inout pads.

`-voh value`

Defines required Voh on output or inout pads.

`-vimin value`

Defines the minimum allowed input voltage on input or inout pads.

`-vimax value`

Defines the maximum allowed input voltage on input or inout pads.

`-vomin value`

Defines the minimum output voltage at output or inout pads.

`-vomax value`

Defines the maximum output voltage at output or inout pads.

`-clock`

Uses a clock driver pad for the listed input port, if available.

`-no_clock`

Directs Design Compiler not to use a clock driver for the listed input port.

`-currentlevel value`

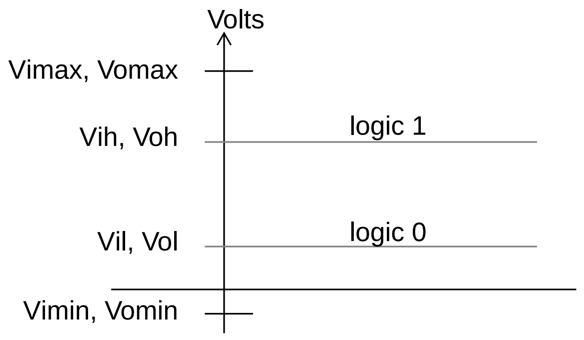
Defines the minimum current rating.

`design_or_port_list`

Defines the list of target objects for `set_pad_type`.

Figure 7-2 shows the voltage levels.

Figure 7-2 Voltage Levels



Inserting I/O Pad Cells

The insert_pads command inserts I/O pad cells in a design.

Invoke insert_pads at the top level, which inserts pads on top-level ports that have the port_is_pad attribute.

The insert_pads command does not bus together inputs into the same pad, using bused pad cells; you need to instantiate such pad cells.

The syntax is

```
insert_pads [-thru_hierarchy] [-respect_hierarchy]
         [-verify] [-verify_effort low | medium | high]
```

-thru_hierarchy

Inserts pads regardless of hierarchical boundaries. The pad is inserted at the level that contains the driving cell of the port (for an output pad) or the loading cell of the port (for an input pad). Use -thru_hierarchy to map sequential logic into complex pads.

-respect_hierarchy

Inserts pads without moving pads through hierarchical boundaries. Use this option to preserve pads at the top level of hierarchy. This is a default option for ASIC technologies.

-verify

Copies the original design and compares it with the design after insert_pads completes processing.

`-verify_effort [low | medium | high]`

Use `-verify_effort` only with the `-verify` option. The effects of the values are as follows:

- low uses the least time trying to verify the design. Parts of the design that are difficult to verify are not pursued.
- medium uses more CPU time to verify the design.
- high tries to verify the design completely. This process might never terminate for some designs.

During pad insertion, a pad is inserted at each input, output, or inout port that has a `port_is_pad` attribute. The characteristics of the inserted pad match those defined by the `set_pad_type` command. For output pads, the program selects the pad with the minimum output current that satisfies the defined characteristic. The type of gate driving the port determines which output pad is inserted.

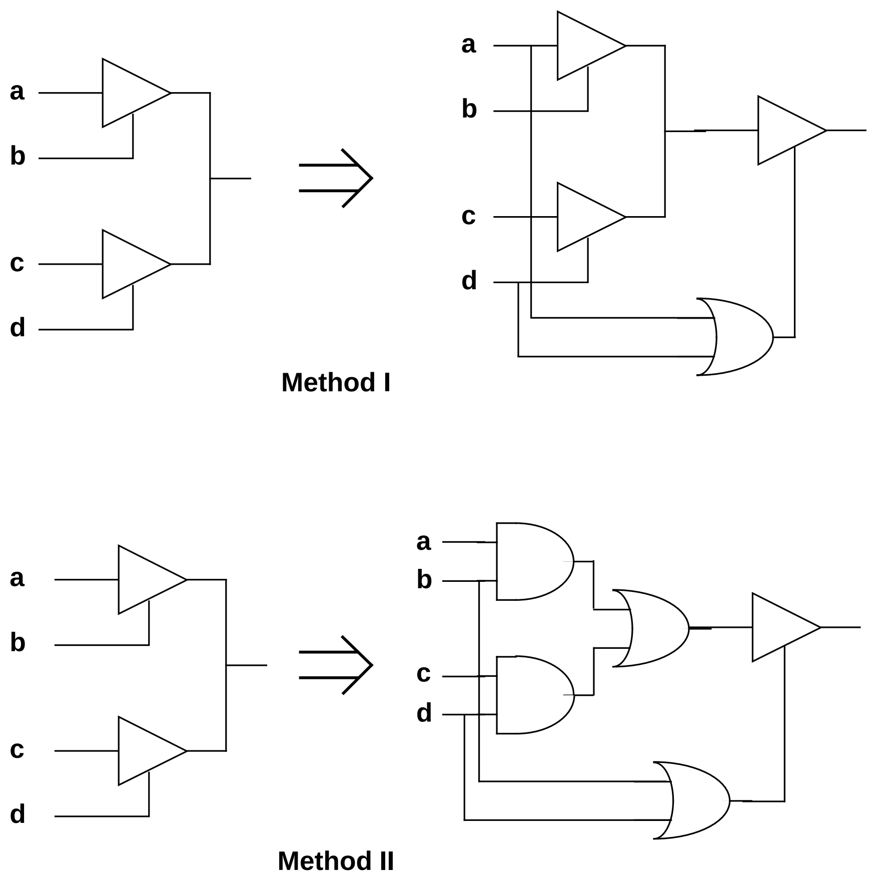
- If a single three-state gate is driving the output port, a three-state I/O cell replaces the gate in the netlist.
- If a single three-state gate is driving a bidirectional port and the port drives other gates in the network, a bidirectional I/O cell replaces the gate in the netlist.
- If none of the gates driving the output port is three-state, the program uses a combinational I/O cell. The simplest combinational I/O cells are a buffer or an inverter. If cells with more-complex logic are available, however, the program moves portions of the core logic into the I/O cell if it finds matching logic.

If the program does not find I/O cells that meet the characteristics you defined, the system issues an error message and searches for cells that have only matching current and voltage level characteristics. If no I/O cells have matching electrical characteristics, the program chooses a cell.

If multiple three-state gates drive a bidirectional port, the system issues an error message to indicate that multiple three-state buffers cannot be replaced by any single I/O cell without changing the functionality of the design. In this case, you must modify the design so each output port has no more than one three-state gate connected to it. If all the enable signals are logically disjoint, the modification is relatively easy to do.

If multiple three-state gates drive an output port, the system uses a simple buffer pad. This decision is based on the assumption that the three-stated net does not receive contention from outside the design.

Figure 7-3 Converting Outputs to a Single Three-State Drive



Input pads are selected according to the load they are driving and the timing requirements. If the input signal fans out to at least one clock pin of a sequential element, a clock buffer is used (if available). You can prevent insertion of this buffer by using the `-no_clock` option with the `set_pad_type` command. Similarly, you can force a clock buffer on a nonclock port, using the `set_pad_type` command with the `-clock` option.

8

Verifying Designs for Functionality

After optimization, the resulting design can be verified for functional equivalence against the previous design or a “golden” design. Verification ensures that the synthesis process or manual design changes did not introduce logical errors. Logic verification ignores timing considerations.

You can compare large designs of a wide variety, including combinational circuits, sequential circuits, three-state drivers, and adders. Designs can be hierarchical and need not be mapped.

This chapter contains the following sections:

- Conforming to Verification Criteria
- Satisfying Verification Requirements
- Running Verification
- Creating and Using a Script of Commands

- Removing a Script of Commands
- Saving a compare_design Script for Reuse

Verification performs a Boolean comparison to determine whether two designs are functionally equivalent — for example, comparing a new HDL model against an old netlist or an old design against a newly synthesized design. Because a Boolean comparison is a complex process, it is more efficient on small designs. Partition large designs into smaller hierarchical blocks, then compare the blocks.

The `compare_design` command runs the comparison. Optionally, you can prepare a script file containing commands for `compare_design` to use during the comparison.

Conforming to Verification Criteria

Designs to be compared must conform to these criteria to be successfully verified with the `compare_design` command:

- Verification sourcepoints and endpoints must have the same name in both designs.

  Verification sourcepoints consist of top-level input ports and noncombinational gate output pins (such as outputs from flip-flops, latches, and three-state components).

  Verification endpoints consist of top-level output ports and noncombinational gate input pins (such as inputs from flip-flops, latches, and three-state components).

- Flip-flops, latches, and three-state components must have a one-to-one correspondence and the same name in both designs. Sequential components do not need to be the same type. For example, you can verify a design that uses JK flip-flops against a design that uses D flip-flops.
- Combinational feedback loops are detected, and all endpoints driven by the feedback loop are set to logic 0 during the remainder of verification. In effect, these endpoints are verified.
- Hierarchical designs must have the same hierarchy, including hierarchical instance names.

Satisfying Verification Requirements

Keep these facts in mind when you verify designs:

- Extra runtime is required. When comparing a synthesized design against the original design, Design Compiler must keep a copy of the original design. The additional time Design Compiler spends on the verification itself contributes to extra runtime compared to a synthesis-only runtime. The verification time is significant compared to the optimization time. Verification time can exceed compile time.
- Additional virtual memory space is required to keep a copy of the original design for comparison.
- Verification requires stronger design methodology than optimization; the process of comparing designs has more-restrictive rules. For example, a warning message appears if a combinational feedback loop is encountered in a design, whether or not it violates any design rules.

Running Verification

The `compare_design` command compares two designs for functional equivalence.

The `compare_design` command performs a Boolean comparison to determine whether two designs are functionally equivalent. This command can be used with `compile` to compare the old design with a newly synthesized design.

You can input a script file of commands to direct `compare_design` during comparison, using the `include` command. The script file is described later in this chapter.

The `compare_design` command returns one of these values:

0	Improper command usage or bad arguments were found.
1	Verification succeeded.
2	Verification was too costly (modify the effort parameter).
3	Verification failed.

When the designs are found not to be equivalent, the command returns a pattern for the cases in which the designs are different. The command does not return an exhaustive list of conflicting patterns.

The syntax is

```
compare_design [-effort low | medium | high]
     [-jtag] [-verbose]
     [-hierarchical] design1 design2
```

`-effort low | medium | high`

Sets the relative amount of CPU time for verification. The default is low. Medium uses more CPU time, and high causes Design Compiler to try to verify the design completely.

`-jtag`

Runs JTAG verification on the two top-level designs. All three-states must be in the top level designs; insertion must have been performed on design2, and the core logic of both design1 and design2 must be grouped into subdesigns.

`-verbose`

Displays messages about JTAG processing and assignment of constants to ports for verifying scanned designs.

`-hierarchical`

Disables flattening the designs before comparison. If you omit this option, the hierarchies are flattened (the default).

`design1`

Represents the original design. This design must already exist in dc_shell. design1 can have don't care conditions associated with it, which are created during execution of the `read` command.

`design2`

Represents the design being compared with design1. design2 must exist in dc_shell. Verification determines whether design2 is a valid implementation of design1. Don't care conditions associated with design2 are ignored.

Example 1

These commands compare two designs. The results show that the designs are functionally equivalent.

```
dc_shell> read -f verilog {old.v new.v}
dc_shell> compare_design old new
Verifying Design (Low effort)
Verification Succeeded
```

The `compare_design old new` command establishes that design new is a valid implementation of design old. It assumes that design old is a specification. (design old can contain don’t care values that are resolved in design new.) When design old contains don’t care values that are resolved in the new design,

- The command `compare_design old new` succeeds.
- The command `compare_design new old` fails.

Example 2

These commands compare two designs. The results show that the designs are not functionally equivalent.

```
dc_shell> read -f verilog {old.v new.v}
dc_shell> compare_design old new
Verifying Design (Low effort)
Information: Verification Failed.
Verification endpoint Z is different.
A distinguishing pattern is:
    A:0
    B:1
All other inputs may be assigned any value to distinguish
the circuits.
```

Creating and Using a Script of Commands

Before you run `compare_design`, you can optionally prepare a script file of `set_compare_design_script` commands using a text editor. The commands guide `compare_design` during the comparison. Include the script file before running `compare_design`.

The `compare_design` command applies the point-specific options `-ignore` and `-only` before it applies the subdesign-specific option `-accept`, enabling acceptance on pairs of subdesigns that have different numbers of outputs.

The primary outputs of the designs being compared must have the same names in both designs. The sourcepoints do not need to have the same names.

The `set_compare_design_script` is added to the compare_design script, which contains dc_shell commands to be applied during the verification phase of the `balance_buffer`, `compare_design`, `compile`, `insert_pads`, `reoptimize_design`, `replace_fpga`, and `translate` commands.

For more information, see the set_compare_design_script man page.

The syntax is

```
set_compare_design_script [-ignore end_point_list]
         [-only end_point_list] [-accept sub_design_list]
```

`-ignore` *end_point_list*

Disables comparison of the endpoints listed in end_point_list with any points in the other design. The first and second strings are path names to subdesigns in design1 and design2. One string must be subfunctions, and the other must be nonnull. The nonnull string indicates the design to which this command refers. Both design names can be nonnull if all the listed endpoints exist in both designs. Subsequent strings are names of endpoints (output ports, sequential cells, and hierarchical input pins) to be ignored. You cannot use `-ignore` with `-only` for the same subdesign.

`-only end_point_list`

Compares only the specified endpoints against points in the other design. The first and second strings are path names to subdesigns in design1 and design2. One string must be subfunctions and the other must be nonnull. The nonnull string indicates the design to which this command refers. Both design names can be nonnull if all the listed endpoints exist in both designs. Subsequent strings are names of endpoints (output ports, sequential cells, and hierarchical input pins) to be compared. You cannot use `-only` with `-ignore` for the same subdesign.

`-accept sub_design_list`

Accepts the specified subdesigns in the original design as functionally equivalent without verification. The strings are path names to subdesigns in design1. The accepted subdesigns must have the same number of output ports in design1 and design2. Use the -ignore option to remove extraneous output ports on either subdesign.

Removing a Script of Commands

The `reset_compare_design_script` command removes the script of commands applied during the verification phase of the `balance_buffer`, `compare_design`, `compile`, `insert_pads`, `reoptimize_design`, `replace_fpga`, and `translate` commands.

If no compare_design script is defined, this command returns 0.

To see the contents of the script, use the `write_compare_design_script` command.

The syntax is

```
reset_compare_design_script
```

For example, enter

```
dc_shell> reset_compare_design_script
```

Saving a compare_design Script for Reuse

The `write_compare_design_script` command saves for reuse the compare_design script of commands to apply during the verification phase of the `balance_buffer`, `compare_design`, `compile`, `insert_pads`, `reoptimize_design`, `replace_fpga`, and `translate` commands.

By default, the `write_compare_design_script` command writes dc_shell commands to the screen. Use the redirection operator (>) to redirect the output to a disk file. When no compare_design script is defined, `write_compare_design_script` returns 0.

You can create a compare_design script before defining the current design. The script is an ordered set of dc_shell commands used only for verification. It is not associated with a particular db design. You can reuse the script for comparing an original specification design against several compile results.

The syntax is

```
write_compare_design_script
```

Example

This example shows how to create a compare_design script and save it. The script is used while comparing an original specification design, MY_SPEC, with an implementation design, IMPL1, that resulted from a previous compilation. Both MY_SPEC and IMPL1 must have an output port out1 and a subdesign whose instance name is BOTTOM.

According to the script,

- out1 is ignored (not verified) in both designs.
- Subdesign BOTTOM is accepted as equivalent (not verified) in both designs.

The null is a placeholder to indicate which design is intended for each `-ignore` option.

```
dc_shell> set_compare_design_script -ignore {MY_SPEC null out1}
dc_shell> set_compare_design_script -ignore {null IMPL1 out1}
dc_shell> set_compare_design_script -accept {MY_SPEC/BOTTOM}
dc_shell> write_compare_design_script

/**********************************************************************
Created for compare_design on Tue Feb 21 15:32:16 1995
***********************************************************************/

set_compare_design_script -ignore {MY_SPEC null out1 }
set_compare_design_script -ignore {null IMPL1 out1 }
set_compare_design_script -accept {MY_SPEC/BOTTOM }
dc_shell> compare_design MY_SPEC IMPL1

Accepting Designs MY_SPEC/BOTTOM and IMPL1/BOTTOM.
Verifying Designs MY_SPEC and IMPL1 (Low effort)
Verification Succeeded
```

9

Linking to Physical Design Tools

Getting physical design information into the design flow early lets you optimize your design for layout. To meet the area and speed constraints defined for a design, Design Compiler considers the placement of the cells and the routing of the nets as well the cells and nets used. As process feature sizes shrink, the effects of net routing (wiring) on area and delay increase.

Design Compiler uses a several of techniques to achieve high-accuracy wiring estimates. This chapter contains the following sections:

- Interacting With Layout Tools
- Forward-Annotating Constraints to Physical Design Tools
- Saving (Forward-Annotating) Delay Data
- Creating a Constraints File

- Providing Constraint Coverage for the Entire Design
- Writing Constraints in the Standard Delay Format
- Writing Constraints in the Synopsys Format
- Using a Unified Constraint Set
- Using Physical Design Tools to Improve Wiring Estimation
- Back-Annotating From Layout
- Defining Net and Cell Delays
- Defining Timing Check Values Between Pins
- Reporting Annotated Values and Checks
- Removing Back-Annotated Values
- Determining Post-Placement Optimization Strategies

Note:

> If you are using the Synopsys Floorplan Manager, see the Floorplan Manager User Guide. You need a Floorplan Manager license or a DC Ultra license to use the `reoptimize_design`, `create_wire_load`, `read_clusters`, and `write_clusters` commands.

In the absence of physical design information, Design Compiler uses statistically generated wire load models to estimate wire lengths in a design. For information about wire load models, see *Design Compiler Reference Manual: Constraints and Timing*.

Interacting With Layout Tools

After Design Compiler optimizes a design to meet its constraints, the design is sent to a layout tool for implementation. Until this point, the goal in synthesis is to estimate the timing and area characteristics of the final layout as accurately as possible so that after layout the design still meets its constraints.

To help ensure a predictable layout implementation, you must

- Make intelligent use of the constraint mechanisms the target layout tool provides to help ensure that the final design still meets its constraints
- Make use of the parasitics and timing generated by the layout tool to perform a more accurate final timing verification
- Make use of incremental resynthesis capabilities to correct any constraint violations that might have arisen during layout

Forward-Annotating Constraints to Physical Design Tools

Design Compiler provides the means to create a variety of constraints that influence floorplanning or layout tools to create an implementation likely to meet the design constraints. Design Compiler creates constraint files in Standard Delay Format (SDF) or Synopsys format, using the `write_constraints` command.

The types of constraints that can be produced are

- Maximum path timing—defines the maximum delay allowed from a startpoint to an endpoint for all the critical paths selected.
- Net priority—identifies high-priority nets along the selected critical paths to be routed first.
- Net timing—defines a target net delay for each net found on the critical paths selected (the net delay Design Compiler calculates).
- Net capacitance—defines a target net capacitance for each net found on the critical paths selected (the net capacitance Design Compiler estimates).
- Cell groups—defines the cell instances with the same `group_name` attribute to be grouped together in the final layout.
- Design hierarchy—defines the cell instances in the same design hierarchies to be grouped together in the final layout.

Maximum path timing constraints depend on the value of the delay constraints. Other constraint types do not depend on the values of the constraints. Use `report_constraint` to see which paths are constrained because the `write_constraints` command writes constraints only for nets on constrained paths.

Saving (Forward-Annotating) Delay Data

The `write_timing` command saves leaf cell pin-to-pin timing information to a disk file. Use `write_timing` to capture delay data.

Forward-annotation is the process of transferring delay and constraint data from a synthesized design to a text file in an SDF or other format. This data is useful in guiding floorplanners or place and route tools.

Design Compiler can write timing information in SDF and Synopsys VHDL format files. (Design Compiler can write constraints in SDF and Synopsys format). Conditions defined for timing arcs in the technology library are written in SDF.

- To create a timing file in a format compatible with the Synopsys VHDL simulator, use `-format synopsys_vhdl`.
- To create a timing file in SDF format, use `-format sdf` or `-format sdf-v2.1`.

The timing file contains data associated with the netlist from which it is created. Names in the SDF must match names in the db netlist. Names must be simple, so use the `change_names` command to simplify the design's names. You can change instance names in the design, using the `define_name_rules` and `change_names` commands.

Load delay (also known as extra source gate delay) is the contribution to the overall cell propagation delay caused by capacitive loading. Load delays can be written out as part of the cell delay or as part of the net delay. The default is to include load delays in the cell delays. Some place and route tools expect load delay to be part of the net delay rather than part of the cell delay. Use the `write_timing -load_delay` option to specify whether load delay is part of the net delay or part of the cell delay.

`Write_timing` writes rise and fall net delays in the timing file. If your simulation library is not v2.2b or later, only one net delay value is supported in simulation and you must set the variable `simulation_library` to v2.2a before using `write_timing`.

Writing Delays in Synopsys VHDL Format

In the synopsys_vhdl format, timing information is written in 1-ns, 1-ps, or 1-us units. The time unit chosen is the time unit specified in the technology library. For example, if the library specifies the time unit in nanoseconds, the timing data will be written in nanoseconds. If the library time unit is 10 ns, the scaling is 10 and the unit is nanoseconds; the timing data is multiplied by 10 and is written in nanoseconds.

- If the library does not specify a time unit, the unit is assumed to be nanoseconds and the scale is the value of the environment variable `vhdlout_time_scale`. For example, if the library has no time unit specified but describes delay in units of 0.1 ns, set `vhdlout_time_scale` to 0.10 to scale the data to nanoseconds.
- If `vhdlout_time_scale` is not specified, 1 is the default, written in multiples of ns, ps, and us. The time unit in the SDF file is the time unit specified in the technology library and will be written in the timing file under TIMESCALE.

Writing Delays in Standard Delay Format

In SDF format, timing information is written in 1-ns, 1-ps, or 1-us units. The time unit chosen is the time unit specified in the technology library. For example, if the library specifies the time unit in nanoseconds, the timing data will be written in nanoseconds. If the library time unit is 10 ns, the scaling is 10 and the unit is nanoseconds; the timing data is multiplied by 10 and is written in nanoseconds.

- If the library does not specify a time unit, the unit is assumed to be a multiple of nanoseconds and the multiple is the value of the environment variable `sdfout_time_scale`. For example, if the library has no time unit specified but describes delay in units of 0.1 ns, set `sdfout_time_scale` to 0.1 to specify the time unit as 0.1 ns and write a timing file in 0.1 ns.
- If `sdfout_time_scale` is not specified, 1 is the default. With the sdf and sdf-v2.1 formats, timing conditions described in the technology library are written to the timing file.

The `write_sdf` command writes a Standard Delay Format (SDF) back-annotation file. It writes the timing information in the SDF format using version v1.0 or v2.1. The timing file contains data associated with the netlist from which it is created.

Note:

> The `write_sdf` command does not write back-annotated conditional timing (for example, the COND construct in SDF back-annotated by the `read_sdf` command).

The syntax is

```
string write_sdf [-version sdf_version]
     [-no_cell_delays]  [-no_timing_checks]
     [-no_net_delays] [-input_port_nets]
     [-output_port_nets]
     [-significant_digits digits]
     [-enabled_arcs_only]
     [-no_internal_pins]
     [-instance inst_name]
     [-annotated]  file_name
```

`-version 1.0`

Selects which SDF version to use. Supported SDF versions are 1.0 and 2.1. SDF 2.1 is the default.

`-no_cell_delays`

Specifies that no cell delays be written in the SDF file. By default, all cell pin-to-pin delays are written to the SDF file. Cell delays include the load delay of the cell. Following the SDF conventions, only cell input pin to cell output pin delays are written. In case one cell output is unbuffered, delays are usually represented in libraries by a delay from an output pin to another output pin. Because this is not allowed by the SDF convention, the SDF delay for an unbuffered output is specified from cell inputs.

`-no_timing_checks`

Specifies that no cell timing checks be written in the SDF file. By default, all cell timing checks such as setup, hold, recovery, and removal are written to the SDF file.

`-no_net_delays`

Specifies that no net delay be written in the SDF file. By default, all net pin-to-pin delays are written to the SDF file.

`-input_port_nets`

Specifies that the SDF include delays of nets connected to input ports of the current design. By default, these delays are not written to the SDF file, because the external connectivity information for ports is not available.

`-output_port_nets`

Specifies that the SDF include delays of nets connected to output ports of the current design. By default, these delays are not written to the SDF file, because the external connectivity information for ports is not available.

`-significant_digits` *`digits`*

Specifies the number of digits to the right of the decimal point that are to be written in SDF delay triplets. Allowed values are 0-13; the default is 3.

`-enabled_arcs_only`

Specifies that the SDF should not include delays of timing arcs that are currently disabled. By default, all timing arcs of the design are written to the SDF file, whether they are disabled or enabled.

`-no_internal_pins`

Specifies that the SDF should not include delay timing arcs from or to internal pins. Timing arcs to or from internal pins are expanded into delays from or to primary input and output of the given cell.

`-instance` *`inst_name`*

Specifies that the SDF must be written only for the instance named *inst_name*.

`-annotated`

Specifies that the SDF should include only timing arcs that have been annotated, with either the `read_sdf`, `set_annotated_delay`, or `set_annotated_check` command.

`file_name`

Represents the name of the SDF file to write.

Examples

This example writes timing information for the design MULT16 to a disk file called mult16.sdf.

```
dc_shell> write_sdf -version 2.1 mult16.sdf
```

Creating a Constraints File

The `write_constraints` command creates a constraints file for physical design tools in SDF or Synopsys format. Timing constraints are written for critical paths. A critical path is a path that has a user-defined constraint.

If you do not use any options, the `write_constraints` command writes all constraint types to the file. You can use any combination of constraint-type options.

If a design has no timing constraints, the `write_constraints` command writes the constraints not related to timing (such as cell grouping and design hierarchy) and the following warning message appears:

```
Warning: This design has no timing constraints selected.
(RPT-12)
```

If the targeted place and route tools do not use the same naming convention as Design Compiler, use the `define_name_rules` and `change_names` commands before using `write_constraints`.

The syntax is

```
write_constraints [-output file_name]
     [-format sdf | sdf-v2.1 ]
     [-max_paths number] [-nworst number]
     [-max_path_slack value]
     [-cover_design | -cover_nets]
     [-net_priorities]
     [-min_net_priority number] [-max_net_priority number]
     [-low_priorities] [-max_path_timing] [-net_timing]
     [-load_delay net | cell] [-net_capacitance]
     [-subtract_pin_cap] [-cell_groups] [-hierarchy]
     [-by_input_pin_name] [-by_output_pin_name]
     [-max_nets value] [-from start_point_list]
     [-to end_point_list] [-through through_point_list]
```

`-output file_name`

Specifies the file to which constraints are written. If you omit this option, the default output file name is designname_constraints.format. designname is the current design, and format is the format specified by `-format`.

`-format sdf | sdf-v2.1`

Specifies the output file format type. Required for specifying an SDF format. If you omit this option, the output format is Synopsys (the default). If you specify SDF format, only `-max_path_timing` constraints are written.

`-max_paths` *`number`*

Specifies the maximum number of constrained paths to write for each constraint group. The paths are the most critical paths (the paths with the smallest timing slack). If you omit this option, one path is selected per constraint group. Usually you do not use this option with `-cover_design`, because `-cover_design` obtains a path set that covers the entire design.

`-nworst` *`number`*

Specifies the maximum number of paths considered at each endpoint. If you omit this option, one worst critical path per endpoint is written. *Number* is an integer greater than 0. If you use `-cover_design`, this option is ignored.

`-max_path_slack` *`value`*

Specifies the maximum slack value to be used in selecting paths. Paths with slack values greater than the specified maximum value are not selected. Slack is calculated as the required arrival time minus the actual arrival time, so a negative slack value implies a constraint that has not been met. If you define *value* as 0.0, only paths with negative slack are generated. If you omit this option, paths are not filtered on the basis of slack.

`-cover_design`

Generates only enough unique paths to provide constraint coverage for the entire design. The worst path through every driver-load pin pair is generated, and nonunique paths are pruned out. Path and net timing constraints are then based on this path set. If you use this option, the `-nworst`, `-to`, `-from`, and `-through` options are ignored. You cannot use this option with `-cover_nets`. Usually, if you use `-cover_design`, do not use `-max_paths`.

`-cover_nets`

Generates the worst path through every net in the design. This is useful for tools that convert path-based constraints to net-based constraints, because each net is constrained by its worst path. Path and net timing constraints are then based on this path set.

Use this option when generating net constraints, because it results in better net coverage and runs faster than a similar run using `-max_paths` to determine the path set. Constraints are actually generated for each net driver, so multidriver or three-state nets might require multiple paths in order to be considered fully constrained. If you use this option, the `-nworst`, `-to`, `-from`, and `-through` options are ignored. You cannot use this option with `-cover_design`.

`-net_priorities`

Writes net priority constraints. Net priority constraints are written in order of decreasing priority, starting with the number specified with `-max_net_priority`.

`-min_net_priority` *`number`*

Specifies the lowest-priority number of nets to write to the file. The default minimum net priority is 0 (nets with the highest timing slack have lowest priority). number is an integer greater than or equal to 0 (the default).

`-max_net_priority` *`number`*

Specifies the highest-priority number of nets to write to the file. The default is 100 (nets with the lowest timing slack have highest priority). *Number* is an integer greater than the integer specified in the `-min_net_priority` option.

`-low_priorities`

Writes unconstrained nets in the constraint file in order of increasing priority. With this option, Design Compiler writes nets with the highest timing slack (lowest priority). The total number of nets written is determined by the `-max_nets` option.

`-max_path_timing`

Writes only the maximum timing paths.

`-net_timing`

Writes net delays on the most-critical paths. The net timing written is independent of the timing slack of the path.

`-load_delay net | cell`

Allows the load delay location to be described as part of the command. This makes the option more obvious (it appears as part of the `-help` message). It is also useful in the `write_script` command, because the script can now force load delay to go one place or another without depending on environment variable settings.

If you omit this option, an information message appears, saying that it is assumed that load delay is included. Under normal circumstances, load delay is assumed to be in the cell.

`-net_capacitance`

Writes only estimated net capacitance constraints in the file. The net capacitance written is independent of the timing slack of the path. Net capacitance is the estimated lumped capacitance, including pin capacitances on each net along selected paths.

`-subtract_pin_cap`

Excludes pin capacitances from the net capacitances that are written. By default, net capacitances include pin capacitances on the net. If you omit this option, the estimated net capacitances written include the sum of all pin capacitances.

`-cell_groups`

Writes constraints only for grouped cells.

`-hierarchy`

Writes constraints only for cell groups in the current design hierarchy.

`-by_input_pin_name`

Writes constraints with cell input pin names instead of net names. Use this option when net names in the Synopsys software are different from the net names in your layout tools.

`-by_output_pin_name`

Writes constraints with cell output pin names instead of net names. Use this option when net names in the Synopsys software are different from the net names in your layout tools.

`-max_nets` *`value`*

Specifies the maximum number of nets to write. The value is a percentage of the total number of nets for each constraint type you want to write expressed as a real number from 0 to 1. The default is 0.05 (writes 5 percent of all nets). To write all nets on the selected paths, use the value 1 (100 percent).

`-from start_point_list`

Specifies a list of starting pins or ports of the constrained paths. If you use `-cover_design` or `-cover_nets`, this option is ignored.

`-to end_point_list`

Specifies the ending pins or ports of the constrained paths. If you use `-cover_design` or `-cover_nets`, this option is ignored.

`-through through_point_list`

Specifies pins or ports that pass through the constrained paths. Constraints are written for paths that go through the pins or ports on `through_point_list`. Use this option to select different paths with the same startpoints and endpoints. If you use `-cover_design` or `-cover_nets`, this option is ignored.

If negative or zero-valued constraints are allowed in the SDF file, set the variable `sdfout_allow_non_positive_constraints` to true. The default is false. If `sdfout_allow_non_positive_constraints` is set to false, paths with a negative or zero-valued constraint are written out with a constraint value of 0.001.

Timing Slack

Timing slack is the difference between the required time and the arrival time from the start to the end of a path.

For each path you select, if the timing slack is negative (meaning the timing constraint is violated), the following type of warning message appears:

```
Warning: Writing violated constraints for the path: ‘ffb/
CP’-’CO’. (WC-9)
```

The most critical path is the path with the smallest timing slack. By default, `write_constraints` writes the constraints for the most critical path in each constraint group in a design. For example, if a design has two constraint groups, `write_constraints` writes constraints for two timing paths (by default). If more than one path has the same timing slack, `write_constraints` arbitrarily selects one of the paths.

You can define the number of critical paths written for each constraint group by using the `-max_paths` option. If you do not use `-max_paths`, `write_constraints` selects only one critical path per constraint group.

Defining Paths

You can define paths by using the `-from_pin` and `-to_pin` options. When you define `-from` or `-to` pins, only paths starting and finishing at the defined points that have timing constraints are written.

Path startpoints are

- Input ports
- Flip-flop clock pins
- Pins having the `set_input_delay` constraint

Path endpoints are

- Output ports
- Flip-flop data pins
- Pins having the `set_output_delay` constraint

When you use the `-from_pin` and `-to_pin` options and no path exists between the specified startpoints and endpoints, `write_constraints` issues a warning message.

Most layout tools can handle cell grouping constraints. You can group cells by setting a `group_name` attribute on them. Cells with the same `group_name` are in the same placement group. Use the `set_attribute` or `group -soft` command to set the `group_name` attribute.

In SDF format, only path timing constraints can be written to the output file. If you specify other constraint-type options with `-format sdf`, a warning message appears.

The `-by_input_pin_name` and `-by_output_pin_name` options are for physical design tools that require pin names instead of net names.

Nets common to different paths appear only once in each of the constraint types. Fully instantiated pins and nets are written. If a selected path goes through a hierarchical block of the current design, pins and nets inside the block are written.

Generating All Constraint Types for the Most Critical Paths

The following set of commands creates a constraint file with all constraint types for the most critical path of each constraint group:

```
dc_shell> read counter.db
dc_shell> max_delay 10 all_outputs()
dc_shell> write_constraints
Information: Writing timing constraints to file
‘counter_constraints.txt’. (WC-8)
```

Generating Path Timing Constraints for the Most Critical Path

The following command creates a constraint file with path timing constraints for the most critical path:

```
dc_shell> write_constraints -max_path_timing
Information: Writing timing constraints to file
‘counter_constraints.txt’. (WC-8)
```

Generating the Ten Most Critical Paths

The following command creates a constraint file for the ten most critical paths in each constraint group:

```
dc_shell> write_constraints -max_paths 10
```

Providing Constraint Coverage for the Entire Design

To fully constrain a design for placement, every net or every edge (connection between two cells) generally requires a constraint. You can ensure that a design is fully constrained by forcing the `write_constraints` command to generate every path in the design. To do this, you must know the number of paths in the design and the largest number of paths to a single endpoint. Generation of the resulting constraints file requires a large amount of CPU time and disk space.

Some place and route tools require only the worst path through any point (other data is not used). In these cases, generating the worst path through every driver-load pin pair in the design to provide full constraint coverage is sufficient. Therefore, the total number of paths for the design is less than or equal to the total number of leaf cell input pins plus the number of primary outputs (often, points are covered by multiple paths).

The `write_constraints` command `-cover_design` option generates just enough unique paths to provide constraint coverage for the entire design. The worst path through every driver-load pin pair is generated, and nonunique paths are pruned out. The overall runtime with the `-cover_design` option is greater, but memory and disk space requirements are significantly less.

Note:

> The `write_constraints -cover_design` option ensures that a constraint is placed on every driver-load pin pair in the design, but the design is not fully constrained. The coverage is minimal; it contains less information than a constraint file with all paths enumerated. It is theoretically possible that a place and route tool could meet all the given constraints and still have timing violations.

The `-cover_design` option is most useful for large designs in which generating data for all paths is not reasonable or when the targeted place and route tool uses only the worst path through any point. A point can be a net or an edge, where an edge is any connection between two leaf-level cells.

For large designs in which generating data for all paths is not reasonable, using the `-cover_design` option might be the only way to ensure that every net is constrained at least once.

When the targeted place and route tool uses only the worst path through any point, any constraints other than the worst are thrown away by the place and route tool.

Example

Assuming that the constraints are to be written in SDF and that the design has a single clock name (clk), the following commands generate the minimal cover set of constraints:

```
dc_shell> group_path -default -to clk
dc_shell> write_constraints -format sdf -cover_design
dc_shell> group_path -to clk -name clk
```

The final command restores the path groups to their original state. For multiple clock designs, there generally are two `group_path` commands per clock (one to ungroup paths associated with a given clock and the other to regroup them). The regrouping performed after writing the constraints is not necessary unless the design will be written out or reoptimized.

Writing Constraints in the Standard Delay Format

Path timing constraints can be written in SDF and used to constrain layout tools to meet critical timing goals. Use the `write_constraints -format sdf` command.

Examples

This command creates an SDF file with the path timing constraint for the most critical path in the current design:

```
dc_shell> write_constraints -format sdf
```

This command writes an SDF file for the ten most critical paths in each constraint group of the current design:

```
dc_shell> write_constraints -format sdf -max_paths 10
```

If you specify a constraint-type option other than `-max_path_timing` with SDF, a warning message appears.

Path timing constraints are written in the file with the PATHCONSTRAINT SDF construct.

The construct format is

```
(PATHCONSTRAINT port_instance1 port_instance_list
port_instance2 value)
```

`port_instance1`

The beginning of the path.

`port_instance_list`

The list of pins between port_instance1 and port_instance2.

`port_instance2`

The end of the path.

Example

This command sequence creates and displays a constraint file in SDF for design counter. The most critical path is from pin ffb/CP to pin CO/Z, and the timing constraint on this path is 100 ns.

```
dc_shell> current_design = counter
dc_shell> max_delay 100 all_outputs()
dc_shell> write_constraints -format sdf
dc_shell> sh cat counter_constraints.sdf

(DELAYFILE
(SDFVERSION)
(DESIGN “counter”)
(DATE “Fri Mar 6 18:52:38 1992”)
(VENDOR “”)
(PROGRAM “Synopsys Design Compiler cmos”)
(VERSION “v3.4”)
(DIVIDER /)
(VOLTAGE 0.00:0.00:0.00)
(PROCESS)
(TEMPERATURE 0.00:0.00:0.00)
(TIMESCALE 1ns)
(CELL
 (CELLTYPE “counter”)
 (INSTANCE)
 (TIMINGCHECK
 (PATHCONSTRAINT ffb/CP ffb/QN m/C m/Z CO/A CO/Z ((100.000000)) )
 )
)
```

Writing Constraints in the Synopsys Format

The constraint file is a generic, tabular text file. This Synopsys format allows you to use an awk or perl script to reformat the data easily.

Constraint files can contain comments. Comments start with the /* characters, beginning in the first column in a line, and terminate with the */ characters.

Each construct in the file starts with a special word. A special word is a predefined string. Special words begin with a dollar sign ($) in the first column in a line. A script that reads a constraint file needs to recognize special words and process the related information (the rest of the file is ignored).

A construct can continue on more than one line with the continuation character (@) in the first column of a new line. The `write_constraints` command usually generates a file with a maximum width of 80-characters. Because names are not split over two lines, names longer than 80 characters extend the width of the line.

Special Words

This section defines special words and includes an example of each special word.

`$DATE` *string_list*

Creation date and time of the constraint file.

```
$DATE  Fri Aug 21 16:34:14 1999
```

`$VERSION` *string_list*

Version of the software used to generate the file.

```
$VERSION  v1997.08
```

`$DESIGN` *string*

Name of the design for which constraints are generated.

```
$DESIGN  counter
```

`$LIBRARY` *string*

Library used to generate the constraints.

```
$LIBRARY  lsi_10k
```

`$HIERARCHY_DIVIDER` *character*

Character used in hierarchical instance names. The hierarchical divider is a slash (/).

```
$HIERARCHY_DIVIDER  /
```

Note:

Use the `change_names` command before `write_constraints` to make sure no names contain the slash character.

`$VOLTAGE` *real_number*

Operating voltage of the design.

```
$VOLTAGE 5.00
```

`$TEMPERATURE` *real_number*

Ambient temperature of operation, in Centigrade.

```
$TEMPERATURE 25.00
```

`$TIME_UNIT` *real_number time_scale*

Timing unit from the technology library. time_scale is seconds (s), nanoseconds (ns), or picoseconds (ps). If no timing unit is specified in the technology library, the default is 1 ns.

```
$TIME_UNIT 10.00ns
```

`$PIN_CAPACITANCES_INCLUDED` *Boolean*

Pin capacitances are included in net capacitance constraints. The default is true.

```
$PIN_CAPACITANCES_INCLUDED 1
```

Constraint Types

The types of constraints written in Synopsys format include:

- Maximum path timing
- Net priorities
- Net timing
- Net capacitance
- Cell groups
- Design hierarchy

Maximum Path Timing Constraints

Maximum path timing constraints are the maximum delays allowed along the selected paths between the startpoints and endpoints.

The construct format is

```
$MAX_PATH_TIMING edge_type value
   from_pin to_pin through_pin list
```

edge_type

The signal at the endpoint with a value r for rise, f for fall, or * for both rise and fall.

value

A real number that represents the maximum delay allowed on the path.

from_pin

The pin or port that represents the startpoint of the path.

to_pin

The endpoint of the path.

through_pin list

A list of pins between the path startpoints and endpoints that contains zero or more pin names.

Net Priorities

Nets are assigned a priority integer value based on timing slack.

- Timing slack for a pin is the difference between the required arrival time at the pin (determined by constraint values) and the actual arrival time.
- Timing slack for a net is the minimum timing slack at the fanout pins of the net.
- Nets with the smallest slack (negative slack is possible) are assigned maximum net priority.

- Nets with the highest slack are assigned minimum net priority.

Design Compiler calculates net priority based on the minimum and maximum timing slack of nets on the selected paths. For example, if you specify a net priority range of 50:100 and the selected nets have a timing slack ranging from 0 ns to 5 ns, net priorities are assigned as listed in Table 9-1.

Table 9-1 Net Priorities

Net timing slack	**Net priority**
0	100
1	90
2	80
3	70
4	60
5	50

The priority for a net with timing slack of 2.4 is calculated as

```
max_net_priority -(slack -min_slack) x
     (max_net_priority - min_net_priority)/(max_slack
     - min_slack)
```

or

```
100 - (2.4 - 0) x (100 - 0)/(5 - 0) = 76
```

The $NET_PRIORITY construct has three construct types. Nets are written in order of decreasing priority.

The default format is

`$NET_PRIORITY` *`net_name priority`*

`net_name`

The hierarchical net name from the current design.

`priority`

An integer that represents the priority.

With the `-by_input_pin_name` or the `-by_output_pin_name` option, the format is

`$NET_PRIORITY` *`{cell_instance_name} pin_name priority`*

`cell_instance_name`

A hierarchical instance name.

`pin_name`

The cell pin name (this is not a hierarchical name).

`priority`

An integer that represents the priority.

With the `-by_input_pin_name` or the `-by_output_pin_name` option, constraints on nets connecting to output or input ports at the top-level design are in the following format:

`$NET_PRIORITY` *`port_name priority`*

Net Timing

Net timing is the estimated net delay for nets along the selected path. The net delay constraint is the maximum net delay between each cell output pin and cell input pin on the path. The delay is independent of the constraint applied to the path.

The $NET_TIMING construct has three construct types.

The default format is

```
$NET_TIMING net_name rise_delay fall_delay
```

`net_name`

String that represents the hierarchical net name from the current design.

`rise_delay` and `fall_delay`

Real numbers that represent the rise and fall delays.

With the `-by_input_pin_name` or the `-by_output_pin_name` option, the format is

```
$NET_TIMING {cell_instance_name} pin_name rise_delay
fall_delay
```

`cell_instance_name`

A hierarchical instance name.

`pin_name`

The string that represents the cell pin name (this is not a hierarchical name).

With the `-by_input_pin_name` or the `-by_output_pin_name` option, constraints on nets connecting to output or input ports at the top-level design are in the following format:

```
$NET_TIMING rise_delay fall_delay
```

Net Capacitance Constraints

Estimated net capacitance constraints are written for all nets along the selected paths.

The estimated capacitances are calculated from the wire load model or determined by back-annotation from the `set_load` command. The capacitances written are independent of the constraint applied to the path.

The $NET_CAPACITANCE construct has two formats.

The default format is

`$NET_CAPACITANCE` *`net_name capacitance`*

`net_name`

The hierarchical net name from the current design.

`capacitance`

A real number and, by default, the sum of the wire capacitance and all pin capacitances on the net. (Use the `-subtract_pin_cap` option of the `write_constraints` command to write only wire capacitance constraints.)

With the `-by_input_pin_name` or the `-by_output_pin_name` option, the format is

`$NET_CAPACITANCE` *`cell_instance_name pin_name capacitance`*

`cell_instance_name`

A hierarchical instance name.

`pin_name`

The cell pin name.

Cell Groups Constraints

Cell group constraints are written for the current design. The `-cell_groups` option of the `write_constraints` command lists cells in the same groups. Cell instances in the same group have the same `group_name` attribute. The `group_name` attribute can be set with the `set_attribute` command or with the `group -soft` command. The `group -soft` command is described later in this section.

Cell instances having the `group_name` attribute are written in the output file and sorted by `group_name`. The total cell area for each group is also written. A cell is written only in one group.

The format is

```
$CELL_GROUP group_name total_area cell_instance_name list
```

`group_name`

A string that defines the group name.

`total_area`

A real number that represents the area of all cells in the group. If the group contains hierarchical blocks, the area of nets fully included in the hierarchical blocks is also added. This area can be used to constrain placement tools. The *total_area* unit is the same as the area unit in the technology library.

`cell_instance_name_list`

The name that defines the cell instance. Each group contains at least one cell instance name. The hierarchy divider is slash (/).

The `group -soft` command sets the `group_name` attribute on cells that would normally be grouped with the `group` command but does not change the design hierarchy. The value of the `group_name` attribute is set with the `-cell_name` option of the `group` command. Usually, you use the `-soft` option with group options such as `-logic`, `-pla`, or `-fsm`.

Note:

> The `-soft` option is not available with the HDL options of group such as `-hdl_block`, `-hdl_all_block`, or `-hdl_bussed`.

If you attempt to group a cell that belongs to a `-soft` group (for example, it already has a `group_name` attribute), the initial `group_name` is overwritten by the new one.

The `ungroup` command also has a -soft option. With `-soft`, `ungroup` removes the `group_name` attribute from the cells that would otherwise be ungrouped but does not modify the design hierarchy. If you use the `-soft` option with the `-flatten` option, the `group_name` attribute is removed recursively down the hierarchy of the selected cells.

Design Hierarchy Constraints

Design hierarchy constraints are written for the current design. The output file lists the leaf cells in the current design and in each hierarchical block of the current design.

The `write_constraints` command lists all leaf cells, using hierarchical instance names, for each hierarchical block.

The design hierarchy constraint allows you to create placement constraints so that the cells in one level of hierarchy in Design Compiler are placed close together.

A cell is written in only one group.

The format is

```
$DESIGN_HIERARCHY cell_instance_name design_area
cell_name_list
```

cell_instance_name

The full hierarchical instance name of the hierarchical block. The hierarchy divider is a slash (/).

design_area

A real number that represents the total cell and net area of the hierarchical block.

cell_name_list

The name of cells at the hierarchical level of *cell_instance_name*. *cell_name_list* does not include hierarchical names. Each hierarchical group contains at least one cell name.

Example

To build a simple hierarchy, enter

```
dc_shell> current_design = counter
dc_shell> group find(cell,ff*) -design_name FLOPS -cell_name flops
dc_shell> current_design = FLOPS
dc_shell> group { ffa ffb} -design_name FLOPSAB -cell_name flopsab
dc_shell> current_design = counter
dc_shell> write_constraints -design_hierarchy
```

The file counter_constraints.txt contains

```
$DESIGN_HIERARCHY counter 78.00 a zero flops b c d e f g h i j k
$DESIGN_HIERARCHY flops 34.00 flopsab ffc ffd
$DESIGN_HIERARCHY flops/flopsab 8.00 ffa ffb
```

Using a Unified Constraint Set

Driving physical design from synthesis, using a single, unified constraint set can give more-predictable results. The constraint process is divided into delay-based constraints and floorplan or physical-based constraints.

Delay-based constraints fall into three classes:

- Net constraints
- Path constraints
- External environment constraints

Net constraints, such as maximum net length, net priority, and net capacitance, overly constrain place and route tools. Constraining a path, instead of each net along a path, gives the tool more freedom to obtain a reasonable placement.

Tools that support path-based constraints using the SDF fall into two general categories: those that require ordered constraints for every path in the design and those that require ordered constraints for the most-critical paths. Generating a constraint for every path is not reasonable, due to the large number of paths.

Similarly, with only a handful of paths, it is likely that some critical path will not get constrained. The reasonable solution to both of these problems is to generate SDF that does not contain every path in the design but does constrain enough paths to cover the entire design. In this case, the word cover implies that every source-load pin pair in the design has at least one path constraint going through it. If this is still too many paths, use the `-cover_nets` option to generate the worst path through every net in the design.

Using Physical Design Tools to Improve Wiring Estimation

Floorplanning tools place groups of cells into regions on a chip. Because Design Compiler has access to information that defines the placement of cells in a region, it can make accurate estimates of wiring and refine a design to improve the chances for a successful layout.

Design Compiler contains features that allow it to closely interact with floorplanning tools.

Using Floorplanners With Synthesis

Floorplanning tools provide early access to more-accurate wiring estimates. Because the quality of designs produced by Design Compiler depends on the accuracy of its wire estimation, the earlier this information is provided, the better.

The depth to which a design is floorplanned affects both synthesis and layout. The deeper the floorplan, the smaller the blocks. A deep floorplan

- Allows for accurate wire estimates but limits the magnitude of changes that can be made
- Can overconstrain the layout tool and cause routability or utilization problems
- Can provide wire delay information, which allows an iterative approach to compilation and chip layout

When you use a floorplanner, you need to understand these tradeoffs.

Floorplanning Before Synthesis

Sometimes a floorplanning tool is used before a design is synthesized to partition a complex design into large blocks and assign the blocks to specific regions on the chip.

This methodology serves two purposes for synthesis:

- The nets that connect the major blocks on a chip are typically the most difficult to estimate with wire load models. A high-level floorplan provides early back-annotation for the global nets and leads to more-accurate wire estimation. See “Back-Annotating From Floorplanners” on page 9-38 for details on how to annotate global wiring estimates into Design Compiler.
- The major blocks can be assigned to specific regions. This action enables Design Compiler to associate more-accurate wire load models with each design, reflecting the size of the region to which it has been assigned. See information about wire load modeling in *Design Compiler Reference Manual: Constraints and Timing* for details about how to associate wire load models with designs.

Communicating Constraints to Physical Design Tools

Some physical design tools can accept timing or other types of constraints to help produce an implementation that meets the design objectives. These tools are invoked after an initial synthesis run is completed.

The most important constraint physical design tools support is logical hierarchy. This constraint allows a tool to create an initial floorplan based on logical hierarchy. By constraining the floorplan in this way, more-accurate wire load models are used during the initial synthesis run for each design assigned to a contiguous region. This implies better wiring estimates, which results in higher probability of the final design meeting the defined constraints.

Some physical design tools support timing or capacitance constraints. Design Compiler can generate a wide variety of constraints for floorplanners and for layout tools. See “Forward-Annotating Constraints to Physical Design Tools” on page 9-3.

Back-Annotating From Floorplanners

After an initial floorplan is created for the mapped netlist, Design Compiler provides features for back-annotation more-accurate wiring information for timing analysis and incremental optimization. Design Compiler accepts the following information from floorplanning tools:

- Net parasitics (lumped capacitance and total net resistance)
- Net and cell delay through the Open Verilog International (OVI) Standard Delay Format (SDF)
- Physical groupings of cells in the floorplan (in the Synopsys PDEF format) if you are using Floorplan Manager

If you created custom wire load models with an external floorplanning or layout tool, take the following steps:

1. Create a file containing a wire load description (use Library Compiler syntax).
2. Add the custom wire load to the target technology library, using the `update_lib` command.

3. Associate the wire load with the appropriate design or physical grouping of cells, using the `set_wire_load_model` command.

Example

```
update_lib my_lib wireld.lib

/* Add the wire load model defined in ‘wireld.lib’ to the technology library
‘my_lib’ */

set_wire_load_model custom_model -library my_lib design_name

/* The newly added wire load model ‘custom_model’ gets associated with the design
‘design_name’ */
```

Back-Annotating SDF Pin-to-Pin Timing and res and cap Values

When pin-to-pin timing and res and cap values are back-annotated, timing information is obtained from the following sources in the following order of precedence:

1. SDF pin-to-pin values
   - The delay information in the SDF is the most accurate.
   - The timing reports have an asterisk (*) on timing arcs read from an SDF file.
   - The SDF data is sufficient if all the timing arcs in the design have a value back-annotated from an SDF file and the design meets timing.
2. Annotated resistance and capacitance values on a net
   - Because lumped resistance and capacitance values are considered, the RC delay numbers are pessimistic and not as accurate as SDF pin-to-pin delays.

- For the following reasons, you need the resistance and capacitance values:

  The technology or place and route library does not contain certain arcs. For example, the technology library has timing arcs for CLK -> QB and QB -> Q, but the technology library has a CLK -> Q arc. The tool requires the back-annotated resistance and capacitance values in order to compute the timing.

  Timing is not met, by a small margin, and you want to do in-place optimization. In such a case, you can use the resistance and capacitance values to compute timing with the new cells.

  Large timing violations exist, and you want to recompile but with accurate wire load models (using `create_wire_load`). The tool uses the capacitance values when it creates the wire load models.

- Missing resistance or capacitance value is extracted from the wire load model.

3. Wire load models

   - Wire load models are based on historical data; they can be the least accurate.

   - Both resistance and capacitance values are generated, then the delay is calculated.

Optimizing With Floorplan Information

After the more accurate floorplan wiring estimates are back-annotated, the design might need to be reoptimized. Design Compiler contains features that support refining a design after floorplanning. Chapter 3, “Optimizing Designs,” describes these features.

- If subdesigns are constrained with the `characterize` command, the `compile -incremental` command can be used. Compile in its pure form does not retain back-annotated information. However, the global net back-annotation used during characterization, combined with accurate wire load models, makes block-level reoptimization with this command possible.
- If a block experiences congestion problems, you can ease the congestion and make the design easier to route by using the `-routability` option to the `compile` command.

After all design constraints are again met, the changes made to the design must be communicated so that the floorplanner can update its representation of the design.

Reporting Wire Loads

The `report_wire_load` command provides a list of all wire load model characteristics set on a design. This command reports the characteristics of a specific wire load model set on a design or library. By default, all wire load models set on the current_design are reported.

The `report_wire_load` command is similar to the `report_lib` command, but it also reports the statistics of how the wire load model was created.

The syntax is

```
report_wire_load [-design design_name] [-name model_name]
    [-libraries] [-nosplit]
```

`-design` *`design_name`*

Specifies a design name. If you omit this option, the current_design is used.

`-name` *`model_name`*

Specifies a model name. The logical hierarchy of the design is searched first; then the physical hierarchy; and, finally, the libraries linked to the design. The first wire load model found that has a matching name is used for the report.

`-libraries`

Specifies the target library name. The wire load models stored in the libraries linking the design are reported, as are as those on the design.

`-no_split`

Disables line splitting when the column fields overflow.

Example

The highlighted fields in the report are described following the report.

```
dc_shell> report_wire_load -libraries
****************************************
Report : wire loads
Design : counter
Version: v1997.01
Date   : Tue Jan 14, 1997
****************************************
Wire load model:   counter_wl
Location       :   counter (design)
Resistance     :   100
Capacitance    :   1
Area           :   0
Slope          :   1.19097
                             Average   Standard  % Standard
Fanout   Length   Points         Cap  Deviation   Deviation
--------------------------------------------------------------
     1     1.00       20        1.00     0.00       0.00
     2     2.31        3        2.31     0.27      11.78
     3     3.50        3        3.50     0.24       6.73
     4     4.69        6        4.69     0.45       9.62
     5     5.89        1        5.89     0.11       1.95
--------------------------------------------------------------
Weighted Average Standard Deviation:                3.49

Wire load model:   top_cluster/cluster_1_wl
Location       :   top_cluster/cluster_1 (cluster)
Resistance     :   100
Capacitance    :   1
Area           :   0
Slope          :   1
                             Average   Standard  % Standard
Fanout   Length   Points         Cap  Deviation   Deviation
--------------------------------------------------------------
     1     1.00        3        1.00     0.00       0.00
--------------------------------------------------------------
Weighted Average Standard Deviation:                0.00

Wire load model:   10x10
Location       :   my_lib (library)
Resistance     :   100
Capacitance    :   1
Area           :   0
Slope          :   0.311

                             Average   Standard  % Standard
Fanout   Length   Points         Cap  Deviation   Deviation
--------------------------------------------------------------
     1     0.53
```

In the wire loads report,

Location

Specifies where the wire load model was found. If a wire load model is on a design and in a library, the design location is indicated.

Resistance, Capacitance, and Area

Reports the values per unit length.

Slope

Reports the slope of the wire load model after the last fanout value.

Points

Specifies the number of points used for a fanout. A high number of points implies that a good population was used to create a statistically more accurate fanout length estimate. If the model was not created from back-annotation, only fanout and length are reported.

Average Cap

Lists the average capacitance, which is directly proportional to length. This is the average capacitance back-annotated and modified by smoothing and trimming.

Standard Deviation

Reports the average difference between each point and the average.

Back-Annotating From Layout

Back-annotation is the process of reading resistance, capacitance, and delay values from a layout file into Design Compiler. Using back-annotation, you can more accurately analyze your circuit timing in Design Compiler after each phase of physical design (floorplanning, placement, global routing, and detailed routing). Back-annotation is also useful if a design requires additional optimization steps to meet its constraints.

Design Compiler provides commands to do the following:

- Back-annotate net load.
- Identify the net to which a pin is connected.
- Replace estimated wire capacitance values with actual values.
- Back-annotate net resistance.
- Reduce runtime of scripts containing `set_load` and `set_resistance`.
- Back-annotate delay values and timing checks for an entire design.
- Read load delay.
- Back-annotate timing information from an SDF file.

Back-Annotating Net Load

Design Compiler calculates total load for nets as follows:

```
total_net_load = pin_capacitance + wire_load
```

pin_capacitance

Sum of capacitances of all pins on a net, defined in the technology library description.

wire_load

Wire capacitance for a given net, usually estimated by Design Compiler from the wire load model.

You can back-annotate the wire_load parameter of the total net load equation by using the `set_load` command.

The syntax for the `set_load` command is:

```
set_load load_value object_list [-min][-max]
[-subtract_pin_load][[-pin_load][-wire_load]]
```

load_value

Defines external load in technology library load units, such as picofarads or standard loads. The *load_value* is a positive floating-point number. It specifies the capacitance value and must be greater than or equal to zero (> = 0).

object_list

Lists one or more net or port names.

`-subtract_pin_load`

Subtracts the pin capacitances of the net from *load_value* before the net load value is set. If the resulting net load value is a negative number, sets net load value to zero. Use `-subtract_pin_load` with nets only.

`-min`

Uses *load_value* to specify the minimum capacitance. If you do not specify a `-min` value, Design Compiler uses the `-max` value for both maximum and minimum analysis.

`-max`

Uses *load_value* to specify maximum capacitance. If you do not specify a `-max` value but do specify a `-min` value, Design Compiler ignores the `-min` value.

`-pin_load`

Defines the pin capacitance outside the port. Use `-pin_load` only with ports.

`-wire_load`

Defines the wire capacitance outside of the port. Use `-wire_load` only with ports.

Examples

```
dc_shell> set_load 6.53 cell57/n15
Performing set_load on net ‘cell57/n15’.
```

Identifying the Net to Which a Pin Is Connected

Sometimes physical design tools produce capacitance reports that associate wire load with the name of a pin connected to the net. Use the `all_connected` command with `set_load` to identify the net to which the pin is connected, and set a load value on it. For example,

```
dc_shell> set_load 6.53 all_connected(cell57/inv2/Z)
Performing set_load on net ’cell57/n15’.
```

CPU overhead is involved in processing the previous command. A design with 100,000 nets can take a long time to process.

Sometimes a net name equals a port name. In such cases, use the following command:

```
dc_shell> set_load 6.53 find(net, cell57/n15)
```

The `all_connected` and `find` commands are described in *Design Compiler User Guide*.

Replacing Estimated Wire Capacitance Values With Actual Values

To replace estimated wire capacitance values in Design Compiler with actual values, create a file containing one `set_load` command for each net.

```
set_load  2.739  INPUT7
set_load  2.101  net204
set_load  3.433  cell33/n28
set_load  1.007  FF21_Q
```

You then include the file as a Design Compiler command script. In some systems, your place and route tool can produce this file.

Use the following procedure to replace the estimated net loads with actual values:

1. Edit the file generated by your place and route tool to use the following line format:

   ```
   set_load load_value net_name [-subtract_pin_load]
   set_load  0.052   cell109/n29
   set_load  0.052   net251
   set_load  0.052   net266
   set_load  0.2152  net798
   set_load  0.1052  net800
   set_load  0.1052  net802
   set_load  0.3052  net803
   set_load  0.052   net804
   ```

2. Run the script file by using the `include` command.

   ```
   include load_file_name
   ```

3. Display the resulting annotated nets with the `report_net`, `report_attribute`, or `report_annotated_delay` command.

See “Reporting Annotated Values and Checks” on page 9-71 for more information on the `report` commands.

Back-Annotating Net Resistance

The `set_resistance` command defines the wire resistance for nets in an optimized design. This command overwrites the Design Compiler internally estimated net resistance values.

Resistance is estimated on the basis of the tree type used in the current operating conditions.

- For a balanced tree, the resistance annotated is assumed to be balanced across all loads. The resistance (R) from the driver to each of the N loads is R/N.
- For a worst-case tree, the resistance from the driver to each load is assumed to be R.
- For a best-case tree, resistance is ignored (not used). The net delay is zero, resulting in optimistic delay values.

The syntax is

`set_resistance` *`resistance_value`* `[-min][-max]` *`object_list`*

`resistance_value`

Specifies the resistance value for nets in the object list, in the units the technology library uses during optimization (usually ohms).

`-min -max`

Uses `resistance_value` for minimum or maximum timing delay analysis. You can specify either `-min`, `-max`, or both; to specify both, use two separate command lines. If you specify a `-max` value but not a `-min` value, Design Compiler uses the `-max` value for both maximum and minimum timing delay analysis.

`object_list`

Lists the nets for which the specified resistance values are set.

Examples

To set a resistance of 200 units on nets a and b, enter

```
dc_shell> set_resistance 200 {a,b}
```

To set a resistance of 300 units on the net U1/U2/Net3, enter

```
dc_shell> set_resistance 300 U1/U2/Net3
```

Sometimes, physical design tools produce resistance reports that associate values with the name of a pin connected to the net. Use the `all_connected` command with `set_resistance` to identify the net to which the pin is connected. For example,

```
dc_shell> set_resistance 6.53 all_connected(cell57/inv2/Z)
```

CPU overhead is involved in processing the previous command. A design with 100,000 nets can take a long time to process. In such cases, use the `find(net, name)` command. For example,

```
dc_shell> resistance 6.53 find(net, cell57/n15)
```

Reducing Runtime of Scripts Containing set_load and set_resistance

Because the `set_load` and `set_resistance` commands verify the design links each time you run them, you can significantly reduce the runtime of a script if you link the design and disable the autolink feature at the beginning of the script. For example, enter

```
dc_shell> auto_link_disable = true
dc_shell> include -quiet load_res.file
```

The include command -quiet option prevents the script commands from echoing to the screen (error messages do print to the screen). As a result, the script runs faster and you can easily find messages about unfound nets in the output list. After the script has finished running, enable the auto-link feature. Enter

```
dc_shell> auto_link_disable = false
```

Back-Annotating Delays and Timing Checks

Design Compiler provides the following commands for back-annotating delay values and timing checks:

read_timing

> Annotates the delay values and timing checks for an entire design from an SDF file.

set_annotated_delay

> Sets the delay values for specified nets and cells in the current design.

set_annotated_check

> Sets an annotated timing check between two or more pins.

SDF does not support internal pins, so read_timing ignores timing to internal pins.

Back-annotating from an SDF file can be ten times as fast as than using the `set_annotated_delay` commands. Use SDF whenever possible.

The `set_annotated_delay` and `read_timing` commands do not perform a 100 percent input validity check; some checking is done during `update_timing`. Therefore, run `update_timing` after you have run `set_annotated_delay` and `read_timing` to verify that all back-annotation is correct.

Reading Load Delay

Before you use the `set_annotated_delay` and `read_timing` commands, you need to know how Design Compiler reads load delay. Design Compiler uses the following delay definitions:

Cell delay

The delay between a state transition on an input pin and the resulting state transition on the output pin of a gate.

Load delay

The amount of cell delay introduced by the load of the net being driven. You calculate this by computing the loaded cell delay and subtracting what the delay would have been if the net had a total capacitance of zero. Load delay is also known as extra source gate delay.

Connect delay (or net delay)

The delay between the output pin changing state and a subsequent input pin changing state. This is the time-of-flight delay on the net.

Design Compiler correctly interprets SDF files with IOPATH and INTERCONNECT delays as

```
IOPATH = cell delay + load delay
INTERCONNECT = connect delay
```

However, some layout tools define the delays as

```
IOPATH = cell delay – load delay
INTERCONNECT = connect delay + load delay
```

Back-Annotating Timing Information From an SDF File

The `read_timing` command reads leaf cell net and cell delay values from an SDF v1.0 or v2.1 timing file and annotates the values to the current design.

Instance-specific pin-to-pin cell and net delays greater than zero are read from the timing file and annotated on the current design.

Note:

> The `read_timing` command saves only the worst time of all timing conditions; `read_timing` does not annotate all timing conditions.

Load delay is the contribution to the overall cell propagation delay caused by capacitive loading. Load delays can be written out as part of the cell delay or as part of the net delay. The default assumes that the load delays are included in the cell delays in the timing file being read. Some delay calculators consider load delay to be part of the net delay, and others consider it part of the cell delay. Use the `read_timing -load_delay` option to specify whether the load delay is part of the net delay or part of the cell delay.

Setup, Hold, Recovery, and Removal Timing Checks

Setup, hold, recovery, and removal timing checks, when present in the timing file, are used to annotate the current_design. The SDF constructs are

SDF constructs	
setup and hold	SETUP, HOLD, and SETUPHOLD
recovery	RECOVERY

SDF constructs	
removal	HOLD

Back-Annotating Information Created for an Instance

To annotate the information from a timing file created for an instance of a subdesign of the current design, use the `-path` and `-design` options. When a subdesign is specified, the net delays to the ports of the subdesign cannot be used to annotate the current design.

Design Compiler annotates only delays between leaf cell pins. When you use the `-path` and `-design` options, net delays up to the subdesign ports cannot be annotated on the current design (because lower-level ports are not leaf cell pins).

Back-Annotating the Worst Timing Delay

Use the `remove_annotated_delay -all` command and the `remove_annotated_check` command before using the `read_timing -worst` command to annotate the worst timing delay and timing checks of all timing conditions for each pin-to-pin annotation.

This behavior does not depend on the type of timing arc being read (whether the arc is a SETUP arc or a HOLD arc). This is particularly important because worst might lead to the belief that for HOLD timing arcs minimum values will be read. The correct behavior is one where all the values read from are either minimum, typical, or maximum, regardless of the type of timing arc.

Instance Names

Instance names in the design must match instance names in the timing file. For example, if the timing file was created from a design using VHDL naming conventions, `design_name` must use VHDL naming conventions. To modify design names, use the `change_names` command.

Cell and Pin Names

Cell and pin names in the SDF file and the current design must be the same. Object renaming does not occur when an SDF file is being read and loaded. If the names do not match, an error message similar to the following appears:

```
Error: Pin ’B1/C1’/’INC1’ could not be found. (SDFN-10)
```

DESIGN Field Names

The DESIGN field in the SDF file must have the same name as the subdesign. If the names do not match, an error message similar to the following appears:

```
Error: The SDF file contains delays for the design ’SYSTEM’,
they cannot be annotated on design ’BAR’. (SDFN-4)
```

Verifying Back-Annotation

To verify that the back-annotation done by `read_timing` is correct, run `update_timing`.

Removing Annotated Delays and Timing Checks

To remove delays read and annotated with `read_timing`, use `reset_design` or `remove_annotated_delay`. The `compile` command also removes annotated delays (except delays in cells and nets that have the `dont_touch` attribute). To remove timing checks annotated with read_timing, use the `remove_annotated_check` command.

The `read_timing` command syntax is

```
read_timing [-load_delay net | cell]
     [-min_triplet min_triplet_name]
     [-max_triplet max_triplet_name]
     [-worst]
     [-design design_name -path path_name]
     timing_file_name
     [-context vhdl | verilog]
```

`-load_delay net | cell`

Specifies whether load delays are included in net delays or in cell delays in the timing file being read. The default is cell.

`-min_triplet` *min_triplet_name*

Specifies which value from the SDF delay triplet is annotated for minimum delay analysis. Valid values for `min_triplet_name` are none (the default), minimum (use the leftmost delay triplet), and typical (use the middle delay triplet). If you choose none (the default), Design Compiler uses the same values for minimum delay analysis that it uses for maximum delay analysis.

`-max_triplet` *max_triplet_name*

Specifies which value from the SDF delay triplet is annotated for maximum delay analysis. Valid values for `max_triplet_name` are maximum (the default), typical (use the middle delay triplet), and none (do not annotate delays for maximum delay analysis). If you choose maximum, Design Compiler annotates the rightmost delay value in each triplet for maximum delay analysis.

`-worst`

Annotates net and cell delays and annotated timing checks if the delays are worse than the current annotated delay. Use `-worst` when the timing file contains conditional timing.

`-design` *design_name*

Specifies the name of the subdesign timing file used to annotate the current_design. Use `-design` when a timing file was written for a subdesign of the current_design. If you omit this option, Design Compiler reads the timing for the current design. If you use `-design`, you must use `-path`.

`-path path_name`

Specifies the hierarchical path from the current design to design_name. If you use `-design`, you must use `-path`.

`timing_file_name`

Specifies the timing file from which to read timing information. The timing file must be SDF v1.0 or v2.1.

`-context vhdl | verilog`

Specifies the context in which the timing file is created. The default context is VHDL.

Example 1

To read the SDF file cir.sdf for the design cir, enter

```
dc_shell> current_design = cir
dc_shell> read_timing cir.sdf
```

or enter

```
dc_shell> current_design = foo
dc_shell> read_timing -format sdf foo.sdf
```

To read the SDF file bar2.sdf created for an instance of design BAR (a subdesign of CIR) with a path from CIR to BAR of U1/U2, enter

```
dc_shell> current_design = CIR
dc_shell> read_timing -design BAR -path U1/U2 bar2.sdf
```

Example 2

To read maximum delay values from one file and minimum delay values from another file, enter

```
dc_shell> remove_annotated_delay -all
dc_shell> read_timing -min_triplet none -max_triplet max\
              -worst max.sdf
dc_shell> read_timing -min_triplet min -max_triplet none\
              -worst min.sdf
```

Example 3

To read timing information of instance u1 of design MULT16 from the disk file mult16_u1.sdf and have the timing annotated on the design MY_DESIGN, enter

```
dc_shell> current_design MY_DESIGN
dc_shell> read_timing -load_delay net -path u1 -design MULT16\
              mult16_u1.sdf
```

Supported SDF Constructs

Supported constructs are INTERCONNECT and PORT, for net delay; IOPATH, for cell delay; and SETUP, HOLD, and SETUPHOLD, for timing checks.

The DEVICE construct is parsed but ignored for compilation. (DEVICE is used by VSS.)

Note:

> Design Compiler reads only absolute delays with SDF—incremental delays are not read.

SDF files are case-sensitive. The DIVIDER (hierarchy divider), TIMESCALE (time unit), and DESIGN (design name) constructs are read from the SDF header.

The hierarchy divider specified in the DIVIDER construct must be either "." or "/".

The `read_timing` command scales the timing data from the SDF time unit, as defined in the TIMESCALE construct, to the unit defined in the library. If no time unit is defined in the library, the default time unit is 1 ns.

If the delay statement in the SDF file has two or more timing expressions, the first two expressions are read as the rise and fall values. If the delay statement has only one timing expression, `read_timing` assumes that the rise and fall times are the same.

Examples

- From this delay statement with one expression,

  ```
  (INTERCONNECT A1/Z A2/B  (2:3:4))
  ```

  (2:3:4) is read as rise delay and fall delay.

- From this delay statement with two expressions,

  ```
  (INTERCONNECT A1/Z A2/B  (2:3:4) (1:1:2))
  ```

  (2:3:4) is read as rise delay and (1:1:2) is read as fall delay.

- From this delay expression, no rise delay is read and (2:3:4) is read as fall delay.

  ```
  (INTERCONNECT A1/Z A2/B  ()(2:3:4))
  ```

- From this delay statement with more than two expressions,

  ```
  (INTERCONNECT A1/Z A2/B  (2:3:4) (1:1:2) (4:5:6))
  ```

(2:3:4) is read as rise delay; (1:1:2) is read as fall delay; and (4:5:6) is not read, as this describes 0-to-Z and 1-to-Z transitions.

Design Compiler does not support all three-states when reading SDF. Only rise (01 and Z1) and fall (10 and Z0) delays are supported; 1Z and 0Z are not supported.

Reading SDF Instance Names

The variable `sdfin_top_instance_name` defines the name prepended to all instance names in the SDF file. This variable is set, by default, to an empty string. Design Compiler assumes that cell instance names do not already contain prepended names. If cell instance names have prepended names, set `sdfin_top_instance_name` to the prepended name.

For example, if the SDF file contains

```
(DESIGN “fifo”)
(DIVIDER  .)
(CELL
  (CELLTYPE “fifo”)
  (INSTANCE  stim\.cell1)

)
(CELL
  (CELLTYPE “AND2”)
  (INSTANCE  stim\.cell1.U1)
)
```

set the following before reading the SDF file:

```
sdfin_top_instance_name = stim.cell1
```

Note:

> The `read_timing` command does not support INSTANCE `*`. You must define each instance timing separately.

Defining Net and Cell Delays

The `set_annotated_delay` command sets actual net and cell delay values in the current design or the net delay between two or more pins or ports on the same net.

To verify that the back-annotation done by `set_annotated_delay` is correct, run `update_timing`. If the current_design is hierarchical, you must link it, using `link -all`, before using `set_annotated_delay`.

Load delay is the portion of cell delay caused by the capacitive load of the net being driven. Some delay calculators consider load delay part of the net delay and others consider it part of the cell delay.

- If your annotated delay value assumes that load delay is part of the cell delay, use `-load_delay cell`.
- If your delay value assumes that load delay is included in the net delay, use `-load_delay net`.

By default, load delays are assumed to be in cell delays and appear in `report_timing` path listings.

The specified delay value overrides the internally estimated cell and net delay value. If the specified pins are not in the same cell or on the same net, an error message appears when the timing is updated and `delay_value` is discarded for those pins. `set_annotated_delay` can be used for pins at lower levels of the design hierarchy. Pins are specified as INSTANCE1/INSTANCE2/PIN_NAME.

To list annotated delay values, use `report_annotated_delay`.

To remove the annotated cell or net delay values from a design, use `remove_annotated_delay` or `reset_design`.

The syntax is

```
set_annotated_delay -net|-cell [-load_delay net | cell]
    [-rise | -fall] [-min] [-max] delay_value
    -from from_list -to to_list [-worst]
```

`-net | -cell`

Specifies whether the annotated delay is a net or cell delay.

If you use `-net`, you must use both `-from` and `-to`. Pins in *from_list* are cell output or inout pins, and pins in *to_list* are cell input or inout pins.

If you use `-cell`, you must use both `-from` and `-to`. Pins in *from_list* are cell input or inout pins, and pins in *to_list* are cell output or inout pins.

`-load_delay net | cell`

Includes load delay location as part of the command. This makes the option more obvious (it appears as part of the -help message). It is also useful in the `write_script` command, because the script can now force load delay to go in one place or another without depending on environment variable settings.

If you omit this option, an information message appears, informing you that Design Compiler assumes that you want to include the load delay. Under normal circumstances, the tool assumes that load delay is in the cell. However, if you have set one or both of the old variables, the tool uses that information to make the assumption.

`-rise | -fall`

Defines the delay as logic high or logic low. If you omit this option, both that rise and fall delay values are set to `delay_value`.

`-min -max`

Specifies whether Design Compiler uses *`delay_value`* for minimum or maximum timing delay analysis. By default (if you enter neither `-min` nor `-max`), the tool uses *delay_value* for *both* minimum and maximum analysis. If you specify -min only or if you specify `-max` only, the tool again uses *`delay_value`* for *both* minimum and maximum delay analysis. You can specify different values for minimum and maximum analysis (using two separate command lines), but you cannot specify only a minimum or only a maximum value.

`delay_value`

Specifies the delay value between pins in a cell or on a net, expressed in the technology library units used during optimization.

`-from` *from_list*

Lists the leaf cell pins or the top-level input ports that are the startpoints of the timing arcs for which delays are annotated.

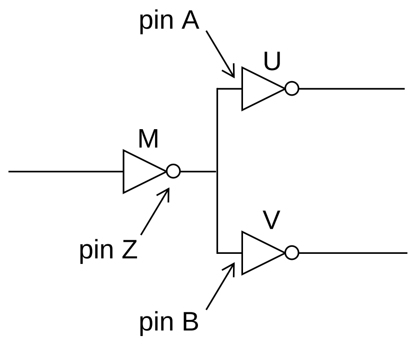
-to_list

Lists the leaf cell pins or the top-level output ports that are the endpoints of the timing arcs for which delays are annotated.

`-worst`

Overwrites the previously annotated delay if the delay value specified is worse than the previously annotated delay value. By default, *`delay_value`* overwrites any previously annotated value.

Example 1



For this circuit, to annotate a net delay of 4 on the net connected to cell output pin M/Z, enter

```
dc_shell> set_annoted_delay -net 4 -from M/Z
```

The net delay between M/Z and U/A is 4, and the net delay between M/Z and V/B is also 4.

To annotate a net delay of 5 on the net connected to cell input pin U/A, enter

```
dc_shell> set_annotated_delay -net 5 -to U/A
```

The net delay between M/Z and U/A is 5, and the net delay between M/Z and V/B is not changed.

To annotate a net delay of 6 between cell output pin M/Z and cell input pin U/A, enter

```
dc_shell> set_annotated_delay -net 6 -from U/Z -to U/A
```

The net delay between M/Z and u/A is 6, and the net delay between M/Z and V/B is not changed.

Defining different net delays for pins on the same net causes ambiguity— for example, setting delay on a net connecting the output pin Z of cell M to the input pin A of cell U.

```
dc_shell> set_annotated_delay -net -rise 4 -from m/Z
dc_shell> set_annotated_delay -net -rise 5 -to u/A
```

In this case, when timing is updated, a warning message appears.

```
(Warning) Overwriting the rise delay between pin ‘M/Z’ and
pin ‘U/A’ with (4). (OPT-835)
```

Example 2

To annotate a rise net delay of 12.3 units for minimum delay analysis between the output pins of two circuits, enter

```
dc_shell> set_annotated_delay -net -rise -min -load_delay\
         net 12.3 -from U1/Z -to U2/A
```

Defining Timing Check Values Between Pins

The `set_annotated_check` command sets an annotated timing check between two or more pins.

Use `set_annotated_check` after place and route for technologies in which the timing check value varies on different instances and the library timing check values do not provide sufficient accuracy.

If the design is not already linked, `set_annotated_check` links it automatically. You can use `set_annotated_check` for the pins at lower levels of the design hierarchy. Specify pins as INSTANCE1/INSTANCE2/PIN_NAME.

To list annotated timing check values, use `report_annotated_check`.

To see the effect of `set_annotated_check` for a specific instance, use `report_timing`.

The syntax is

```
set_annotated_check check_value
     -from from_pins -to to_pins
     -setup | -hold | -removal | -recovery | -nochange_high
| -nochange_low
     [-rise -fall]
     [-clock clock_check]
     [-worst]
```

check_value

Identifies the timing check value between pins on a cell, expressed in the units in the technology library used during optimization. The *check_value* can be negative.

`-from from_pins`

Lists leaf clock pins that are the startpoints of the arcs for which the check is annotated.

`-to to_pins`

Lists leaf data pins that are the endpoints of the arcs for which the check is annotated.

`-setup | -hold` | -removal | -recovery | -nochange_high | -nochange_low

Specifies the timing check type. A corresponding timing arc must exist between the from and to pins. You must specify one of these options.

- `-setup`

  Specifies that data is stable for the amount of time specified by the check value before the closing clock edge.

- `-hold`

  Specifies that data is stable for the amount of time specified by the check value after the closing clock edge.

- `-removal`

  Specifies that an asynchronous set or clear inactive edge cannot occur until the check value time after the closing clock edge. This is essentially a hold requirement on an asynchronous pin, but only the inactive edge is considered.

- `-recovery`

  Specifies that an asynchronous set or clear inactive edge cannot occur within the check value before the closing clock edge. This is essentially a setup requirement on an asynchronous pin, but only the inactive transition is considered.

- `nochange_high`

  Indicates that the data must not rise within the time period specified by *check_value* before the clock becomes active and must not fall until the time period specified by *check_value* after the clock becomes inactive.

  A rise data transition corresponds to the check before the activating clock edge. A fall data transition corresponds to the check after the deactivating clock edge. A rise clock transition indicates that the clock is active-high. A fall clock transition indicates that the clock is active-low.

- `nochange_low`

  Indicates that the data must not fall within the time period specified by *check_value* before the clock becomes active and must not rise until the time period specified by *check_value* after the clock becomes inactive.

A fall data transition corresponds to the check before the activating clock edge. A rise data transition corresponds to the check after the deactivating clock edge. A rise clock transition indicates that the clock is active-high. A fall clock transition indicates that the clock is active-low.

`-rise | -fall`

Specifies whether the check is for data rise or fall transition. If you omit both options, both the rise and fall values are set.

`-clock clock_check`

Specifies whether the check is for clock rising or falling. Valid values for *clock_check* are rise and fall. The default sets checks for both clock rise and clock fall.

`-worst`

Causes overwriting of previously annotated information if the specified check value is worse than the previously annotated check value. By default, `set_annotated_check` overwrites any previously annotated value.

Example

To annotate a setup time of 2.1 units between clock pin CP of cell instance u1/ff12 and data pin D of the same cell instance, enter

```
dc_shell> set_annotated_check -setup 2.1 -from u1/ff12/CP -to u1/ff12/D
```

To remove annotated timing check values from a design, use `remove_annotated_check` or `reset_design`.

Reporting Annotated Values and Checks

Reports provide information about the implementation of your design. Design Compiler provides commands to report the following:

- Load and resistance values
- Annotated delay values
- Annotated timing checks
- Back-annotated values command summary

Reporting Load and Resistance Values

The `report_net` command displays the net load and resistance values. Annotated values appear in the Attributes column with a c or an r.

Attributes such as `dont_touch` are displayed for each net. The `dont_touch` attribute can be present on a net as an implicit attribute. This can happen when the `set_dont_touch_network` command is used. All nets in the transitive fanout of a port are affected, but the `dont_touch` cannot be removed independently on these nets.

The syntax is

```
report_net [-nosplit] [-noflat] [-transition_times]
     [-cell_degradation] [-min]
     [-connections [-verbose]] [net_list]
```

`-nosplit`

Disables line splitting when information exceed its column’s width.

`-noflat`

Displays fanin and fanout information throughout the hierarchy for each net as if the net were flattened. Use `-noflat` for backward compatibility to produce net reports that trace only fanin and fanout for the current level of hierarchy.

`-transition_times`

Displays the rise and fall transition times for each net.

`-cell_degradation`

Displays the actual net capacitance, the net capacitance limit, and the net capacitance violation. (The net capacitance limit is calculated from the `cell_degradation` table for net drivers.)

`-min`

Displays minimum capacitance, resistance, or transition time values, rather than maximums. (The default is maximum values only.)

`-connections`

Reports information about leaf cell pins connected to the nets.

`-verbose`

Reports verbose connection information. If you use `-verbose`, you must use `-connections`.

`net_list`

Displays only the listed nets. If you omit a *net_list*, Design Compiler displays all nets in the current instance.

Example

```
dc_shell> report_net
Information: Updating design information... (UID-85)
****************************************

Report : net
Design : DESIGN
Version: v1997.01
Date   : Tues Jan 14 1997
****************************************
. . .
Attributes:
    d - dont_touch
    c - annotated capacitance
    r - annotated resistance
Net         Fanout  Fanin  Load Resistance Pins Attributes
----------------------------------------------------------
. . .
cell57/n16     1      1    1.00    100.00     2    r
cell57/n17     3      1    3.17    100.00     4    c, r
cell57/n18     2      1    5.36    100.00     3    c, r
. . .
----------------------------------------------------------
Total 13 nets 25     13   31.53   1300.00    38
Maximum        4      1    8.53    100.00     5
Average     1.92   1.00    2.43    100.00  3.22
```

You can also display back-annotated values by using the `report_attribute -net` command. The `load` attribute is used for back-annotated capacitance. The `ba_net_resistance` attribute is used for back-annotated resistance.

Reporting Annotated Delay Values

The `report_annotated_delay` command displays back-annotated data for cells or nets.

The delay values reported are the resulting cell and net delays used in `report_timing` and might not be your annotated values if delays were annotated with the `-load_delay net` option of the `read_timing` command or the `set_annotated_delay` command.

The syntax is

```
report_annotated_delay [-cell] [-net] [-min] [-nosplit]
```

`-cell`

> Reports back-annotated pin-to-pin delay on cells. Use the `set_annotated_delay` -cell command or the `read_timing` command to annotate cell delays.

`-net`

> Reports back-annotated pin-to-pin delays, net capacitances, and net resistances on nets. Net back-annotation is the net capacitance (annotated with the `set_load` command), net resistance (annotated with the `set_resistance` command), and connect delays (annotated with the `set_annotated_delay -net` or `read_timing` command). If you omit both `-cell` and `-net`, both cell and net back-annotation are reported.

`-min`

> Reports annotated delay values used for minimum delay analysis. If you omit `-min`, Design Compiler reports values used for maximum delay analysis.

`-nosplit`

Disables line splitting when information exceeds its column's width.

Example 1

This example shows that the counter design has only one cell delay annotated between pin CO/A and pin CO/Z.

```
dc_shell> report_annotated_delay -cell
****************************************
Report : annotated -cell
Design : counter
Version: v1997.01
Date : Tues Jan 14 1997
****************************************
Cell Name    From   To   Rise   Fall
-------------------------------------
  CO          A     Z    100.00 100.00
```

Example 2

This example shows that the counter design has one net with annotated values. Net h has a lumped capacitance (50.00) and a lumped resistance (200.00). A net delay of 200 is annotated between two of the pins on net h. Net h has no annotated timing between pin ffc/QN and pin r/B.

```
dc_shell> report_annotated_delay -net
****************************************
Report : annotated -net
Design : counter
Version: v1997.01
Date : Tues Jan 14 1997
****************************************

    Net Name    From    To    Rise    Fall    Load    Res.
-----------------------------------------------------------
   h    ffc/QN    w/A    200.00    200.00    50.00    200.00
```

```
    h    ffc/QN    m/B    200.00    200.00    50.00    200.00
     h    ffc/QN    r/B                50.00   200.00
```

Reporting Annotated Timing Checks

The `report_annotated_check` command displays back-annotated timing checks on the current design, including the data rise and fall and the clock edge.

To list annotated delays, use `report_annotated_delay` (described earlier in this chapter).

The syntax is

```
report_annotated_check [-nosplit]
```

`-nosplit`

Disables line splitting when information exceeds its column's width.

Example

```
dc_shell> report_annotated_check

****************************************
Report : annotated_check
Design : ff_tcheck
Version: v1997.01
Date : Tues. Jan 14 1997
****************************************


Cell Name  From   To     Rise    Fall      Timing Check
----------------------------------------------------------------
U1         c      cdn    0.16    0.00      recovery_clock_rising
U1         cn     cdn    0.33    0.00      removal_clock_rising
```

Post-layout optimization and in-place optimization remove the annotated timing checks when they become invalid (when a net connected to the cell with timing checks is modified).

Reporting Back-Annotated Values Command Summary

Table 9-2 summarizes the commands for displaying back-annotated values in the current design.

Table 9-2 Summary of Commands for Reporting Back-Annotated Values

To report this	**Use this**
Timing information for current instance	`report_timing`
Back-annotated data for cells or nets	`report_annotated_delay`
Annotated timing checks	`report_annotated_check`
Net load and resistance values for current instance or current design	`report_net`
Attributes and their values associated with a cell, net, pin, port, instance, or design	`report_attribute`

Removing Back-Annotated Values

You can remove back-annotated values from the current design. Design Compiler provides commands to

- Remove annotated delay values
- Remove annotated timing checks between specified pins
- Remove annotated resistance and capacitance values
- Remove back-annotated values command summary

Removing Annotated Delay Values

The `remove_annotated_delay` command removes annotated delay values between two pins or all annotated delay values in the same cell.

Both rise and fall annotated delays are removed. `remove_annotated_delay` can be used for pins at lower levels of the design hierarchy. These pins are specified as INSTANCE1/INSTANCE2/PIN_NAME.

If the current_design is hierarchical, link the design by using `link -all` before using `remove_annotated_delay`.

Use the `-from` and `-to` options in the same manner you used them to set annotated values. For example,

To Remove This	Use this
An annotated delay set with set_annotated_delay -from	`remove_annotated_delay -from`
An annotated delay set with set_annotated_delay -to	`remove_annotated_delay -to`
An annotated delay set with set_annotated_delay -from -to	`remove_annotated_delay -from -to`

A timing arc must exist between the pins in *from_list* and the pins in *to_list*, or Design Compiler will not remove any annotated delay and a warning will appear:

```
dc_shell> remove_annotated_delay -from ffd/Q -to m/B
Warning: There is no timing arc between pin ‘ffd/Q’ and pin ‘m/B’. (OPT-834)
```

When the tool removes delays, informational messages appear:

```
Information: Removing delays annotated to pin ‘w/A’. (OPT-831)
Information: Removing annotated delays from pin ‘ffc/QN’ to pin ‘w/A’. (OPT-830)
```

If no annotated delay values exist on a net or cell specified in the `remove_annotated_delay` command, no warning or informational message appears.

The syntax is

```
remove_annotated_delay -all | -from pin_list | -to pin_list
```

`-all`

Removes back-annotated net and cell delay values from the current design. If you use `-all`, you cannot use `-from` and `-to`.

`-from` *`pin_list`*

Lists leaf cell pins or top-level input ports that specify the start point of the timing arcs from which to remove annotated delays. If you use `-from`, you cannot use `-all`.

`-to` *`pin_list`*

Lists leaf cell pins or top-level output ports that specify the endpoint of the timing arcs from which to remove annotated delays. If you use `-to`, you cannot use `-all`.

Examples

To remove a delay value from a net annotated with the command

```
dc_shell> set_annotated_delay -net delay_value -from ffd/CP -to m/B
```

enter

```
dc_shell> remove_annotated_delay delay_value -from ffd/CP -to m/B
```

To remove all annotated delays from the current design, enter

```
dc_shell> remove_annotated_delay -all
Information: Removing all annotated delays from design ‘counter’. (OPT-804)
1
```

Removing Annotated Timing Checks Between Specific Pins

The `remove_annotated_check` command removes annotated timing check information between specific pins.

Removes annotated timing checks between the specified pins. Data rise and fall and clock rise and fall annotated checks are removed by default. The `remove_annotated_check` command can be used for pins at lower levels of the design hierarchy. These pins are specified as
INSTANCE1/INSTANCE2/PIN_NAME.

If the design is not already linked, `remove_annotated_check` links it automatically.

The `remove_annotated_check` command removes annotated timing checks set by `set_annotated_check` and removes setup, hold, recovery, or removal annotated timing checks set by `read_timing`.

The syntax is

```
remove_annotated_check
     -all | -from from_pins | -to to_pins
     [-rise |-fall] [-clock rise | fall]
     [-setup] [-hold] [-recovery] [-removal]
```

`-all`

Removes all annotated checks in the current design. You cannot use `-all` with any other option.

`-from` *`from_pins`*

Lists leaf cell clock pins (not hierarchical boundary pins) from which to remove annotated checks. If you omit this option, the checks annotated to *to_pins* are removed. You cannot use this option with `-all`.

`-to` *`to_pins`*

Lists leaf cell data or asynchronous pins (not hierarchical boundary pins) from which to remove annotated checks. If you omit this option, the checks annotated to *from_pins* are removed. You cannot use this option with `-all`.

`-rise | -fall`

Removes checks for data rise or fall transitions. If you omit both options, both values are removed.

`-clock rise | fall`

Removes the check for clock rising or falling. If you omit this option, checks for both clock rise and fall are removed. You cannot use this option with `-all`.

`-setup`

Removes only setup timing information.

`-hold`

Removes only hold timing information.

`-recovery`

Removes only recovery timing information.

`-removal`

Removes only removal timing check information.

Example

To remove an annotated setup check between pins u1/u2/CP and u1/u2/D, enter

```
dc_shell> remove_annotated_check -setup -from u1/u2/CP -to u1/u2/D
```

To specify a rising or falling clock edge, enter,

```
dc_shell> remove_annotated_check -clock rise \
        -from find(pin,"U1/cn") -to find(pin,"U1/sdn")
dc_shell> remove_annotated_check -clock fall \
        -from find(pin,"U1/cn") -to find(pin,"U1/sdn")
```

You can use the `reset_design` command to remove back-annotated values in the design, but `reset_design` removes all attributes and constraints from the design.

Removing Annotated Resistance or Capacitance Values

You can also use the `remove_attribute` command to remove resistance or capacitance back-annotation from specified nets in the design.

- To remove annotated capacitance, remove the load attribute.

- To remove annotated resistance, remove the attribute `ba_net_resistance`.

Examples

To remove the back-annotated resistance on net U1/U2/Net3, enter

```
dc_shell> remove_attribute find(net,U1/U2/Net3)
          ba_net_resistance
```

To remove all annotated capacitances on all nets, enter

```
dc_shell> remove_attribute find(net,"*") load
```

To remove the annotated capacitance on net foo, enter

```
dc_shell> remove_attribute find(net,"foo") load
```

Removing Back-Annotated Values Command Summary

Table 9-3 summarizes the commands for removing back-annotated values.

Table 9-3 Summary of Commands for Removing Back-Annotated Values

To Do This	Use this
Remove annotated values between two pins or all annotated values from the current design	`remove_annotated_delay`
Remove annotated timing checks between specific pins	`remove_annotated_check`
Remove annotated resistance	`remove_attribute load`
Remove annotated capacitance	`remove_attribute ba_net_resistance`

Table 9-3 Summary of Commands for Removing Back-Annotated Values

To Do This	Use this
Remove all user-specified objects and attributes, except those defined using set_attribute.	`reset_design`

Determining Post-Placement Optimization Strategies

After layout, the final timing verification might uncover some timing or design rule violations that must be fixed. Because layout is a time-consuming and difficult process to predict, making small modifications to the netlist and patching the changes into the existing layout is preferable to creating a new layout.

Design Compiler provides optimization features for fixing timing and design rule violations after layout with a minimal number of changes. Design Compiler also contains features to help the layout tool make the corresponding incremental changes.

The features are

- Incremental compilation (`compile -incremental`), described in Chapter 3, “Optimizing Designs.”
- Reoptimization (`reoptimize_design`), described in the *Floorplan Manager* User Guide; requires an DC Ultra license or Floorplan Manager license.

10

Optimizing Finite State Machines

Design Compiler supports unique optimizations for finite state machines (FSMs). An FSM is a special representation of a sequential design. Design Compiler provides capabilities for representing, extracting, and optimizing FSM designs. This chapter contains the following sections:

- Sequential Circuits
- Synthesizing Finite State Machines
- Extracting a Finite State Machine
- Compiling Finite State Machines
- Optimizing a Finite State Machine
- Verifying a Finite State Machine
- Creating Finite State Machine Reports

Sequential Circuits

Circuits are classified as combinational or sequential.

Combinational circuits are time-independent; the output function of a design depends only on the current input values.

Sequential circuits are time-dependent. These circuits have outputs that depend not only on the current input values but also on the sequence of previous input values. Each sequence of previous input values causes the circuit to assume a specific state. Because practical designs have a finite number of such internal states, these types of circuits are called FSMs.

The current state of an FSM is stored as a predefined value for each state (the state encoding), and one or more memory elements are used. Typically, these memory elements are flip-flops that are either reset (output pin Q at logic 0) or set (output Q at logic 1). Each internal state is represented as a unique pattern of logic values of 0 and 1 stored by the state flip-flops. These patterns of logic 0 and 1 are the encoding for a state.

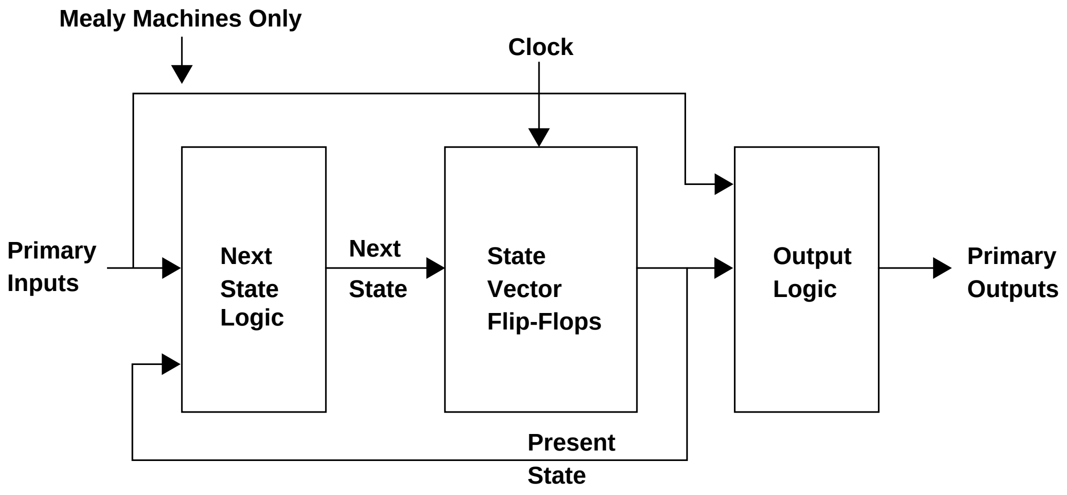
A circuit with N states requires at least log2(N) flip-flops. These flip-flops are collectively referred to as the state vector. A state vector with a length of M has 2M possible patterns (state encodings). If the number of valid states (N) in a circuit is less than the maximum number of encodable states (2M), there are some state encodings that do not represent a valid state. In a correctly defined circuit, unused encodings never appear, so they can be considered don’t care conditions to simplify the logic for the FSM.

Finite State Machine Architecture

Design Compiler optimizes the state encoding of each FSM, then converts them to Boolean equations and technology-independent flip-flops.

Figure 10-1 shows the general structure of an FSM.

Figure 10-1 Architecture of a Finite State Machine



The general architecture shown in Figure 10-1 is called a Mealy machine. In some FSMs, the output logic block depends only on the present state (not the primary inputs). This type of FSM is called a Moore machine.

The architecture for an FSM consists of a set of flip-flops, for holding the state vector, and two combinational logic networks: the next state logic and the output logic.

The FSM primary inputs and the present-state vector feed the next-state logic, which generates the next-state vector for the data pins of the state vector flip-flops. The output pins of the state vector flip-flops hold the value of the present state and are fed to the output logic and back to the next-state logic. The primary outputs are generated by the output logic from the primary inputs and the present-state vector.

FSMs can be described in terms of a state table, as described in the following section, or as a set of state vector flip-flops and associated combinational logic (netlist), described under “Extracting a Finite State Machine” in this chapter.

Finite State Machine Representation

The behavior of an FSM is typically described in terms of state transitions. At any instant, the FSM is in its current state. When a synchronizing event occurs (usually the rising or falling edge of a periodic clock signal) the FSM transitions to the next state. The next state is a function of the current input values and the previous (formerly the current) state.

State Table

A state table is a description of the behavior of an FSM. It consists of rows that each describe a particular transition in the machine. The state table rows have four columns: input values, current state conditions for that row’s transition, next state, and the transition’s output values.

Example

This example shows a simple state table with one input, two states (RESET and STATE1), and two outputs.

```
0  RESET   RESET   00
1  RESET   STATE1  01
0  STATE1  RESET   10
1  STATE1  STATE1  11
```

The state table input format is described in Appendix D, “State Table Design Format.”

Typically, the items in the present- and next-state columns are symbolic state names rather than sequences of logical bit values (state encodings). The correspondence between state names and their unique bit patterns (encodings) can be specified after the state table or by use of the Design Compiler commands `set_fsm_encoding` (sets encoding values) and `set_fsm_state_vector` (names the state vector flip-flops and sets the length of the state vector).

The example shows that all combinations of input values and present states are specified, as are all output possibilities. You can describe a state table by specifying only the set of state transition rows you want, leaving the unspecified transitions (if any) as don’t care conditions, as described in the following section and in “Encoding the States Manually and Specifying a State Assignment Style” on page 10-33.

Completely and Incompletely Specified Finite State Machines

FSMs have the following sets of don’t care conditions:

- Input conditions under which the next state of the machine is unspecified. These input conditions are the next state don’t care set and are obtained from the state table description.
- Output conditions under which the value for certain outputs is unspecified. These output conditions are the output don’t care set and are obtained from the state table description.
- State codes not used in the particular encoding of the FSM. These state codes are the encoding don’t care set and are automatically derived by Design Compiler after the encodings for all states are known.

The output and encoding don’t care sets are combinational don’t cares that simplify the combinational logic of the machine.

The next state don’t care set is a sequential don’t care— that is, the machine’s behavior is undefined for these conditions and can transition to any state in the machine, including invalid states.

A machine whose transition behavior is specified for all possible input conditions (has no next state don’t care set) is a completely specified FSM. A completely specified FSM can still have output and encoding don’t care sets.

A machine whose transition behavior is not specified for all possible input conditions (has a next state don’t care set) is an incompletely specified FSM.

If an incompletely specified FSM description is read into Design Compiler, the following message is displayed during compilation:

```
Warning: In design name, the next state is unspecified for
some transitions. (FSM-104)
```

The difference between completely and incompletely specified FSMs is important in FSMs extraction.

Synthesizing Finite State Machines

This section describes how to use Design Compiler commands with FSM designs.

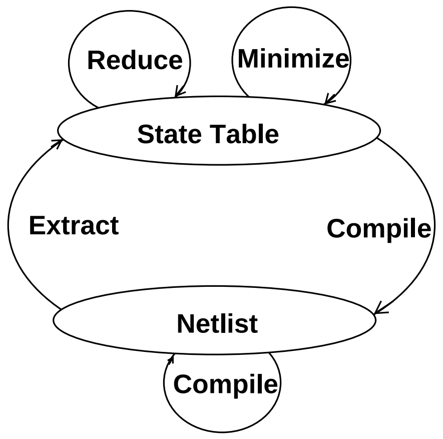
Model for Synthesizing a Finite State Machine

To use Design Compiler to synthesize an FSM,

1. Read the design into Design Compiler.
2. If the design is not in state table format, extract the FSM from the circuit, as follows:
   a. Group the state vector flip-flops (`group -fsm` command) from the set of flip-flops in the circuit, if necessary.
   b. Extract the FSM portion of the design (`extract` command).
3. Optionally, specify FSM attributes (state names and encodings, encoding generation style, and state vector flip-flops).
4. Specify circuit-level constraints and attributes for the design.
5. Compile the design.

Figure 10-2 shows the two levels of FSM design representation, state table and netlist, and diagrams the types of commands associated with each.

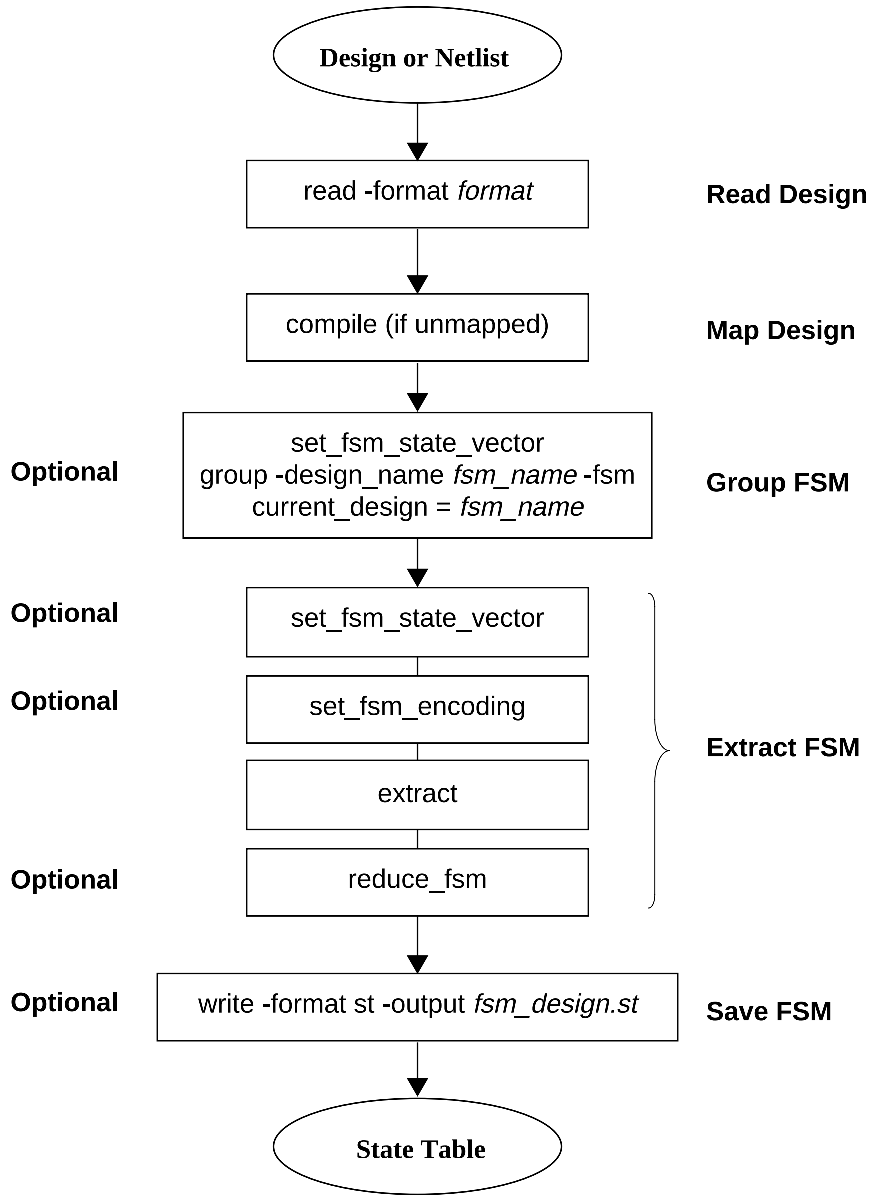
Figure 10-2 Finite State Machine Command Diagram



Extracting a Finite State Machine From a Sequential Design

Figure 10-3 shows the flow from a design or netlist to an FSM, including optional commands.

Figure 10-3   Extracting a Finite State Machine From a Design



Reading in the Design

Read the design into Design Compiler, using the appropriate format option. For example, for a design in EDIF format, enter

```
dc_shell> read -format edif my_design.edif
```

Mapping the Design

If the design is not mapped, run `compile` to map the design to gates.

Grouping the Finite State Machine

If the entire design is an FSM, grouping the FSM is optional.

Grouping the FSM part of a circuit produces a new level of hierarchy containing only the FSM state vector flip-flops and their associated logic. The new FSM design is still in netlist format.

1. If not all flip-flops in the design are part of the FSM, use the `set_fsm_state_vector` command to name the FSM state vector flip-flops. For example, enter

   ```
   dc_shell> set_fsm_state_vector {ff0, ff1, ...}
   ```

2. Use the `group -fsm` command to group the subset of the design that includes only the FSM flip-flops and their associated logic into a new level of design. For example, enter

   ```
   dc_shell> group -fsm -design_name my_fsm
   ```

3. Move down into the new FSM part of the design, by setting the `current_design` variable to the new FSM name. For example, enter

   ```
   dc_shell> current_design = my_fsm
   ```

Extracting the Finite State Machine

Extracting an FSM from a circuit changes the representation from netlist format to FSM (state table) format. These FSM-specific optimizations can be applied:

1. Set the order of the state vector flip-flops, using the `set_fsm_state_vector` command. The order of the flip-flops must agree with the order of the state vector bits; each bit in the state vector is represented by one flip-flop, in order. For example, enter

   ```
   dc_shell> set_fsm_state_vector {ff0, ff1, ...}
   ```

2. Optionally set the state encoding with the `set_fsm_encoding` command. State encoding gives Design Compiler the names and values of each state. For example, enter

   ```
   dc_shell> set_fsm_encoding {"STATE0=0", "STATE1=1", ...}
   ```

3. Extract the FSM (convert the circuit into an FSM). Enter

   ```
   dc_shell> extract
   ```

Optionally, you can use the `reduce_fsm` command to minimize the state transition logic. For example, enter

```
dc_shell> reduce_fsm
```

Saving the Finite State Machine

After the design is in FSM format, you can optionally write it to a file in state table format. For example, enter

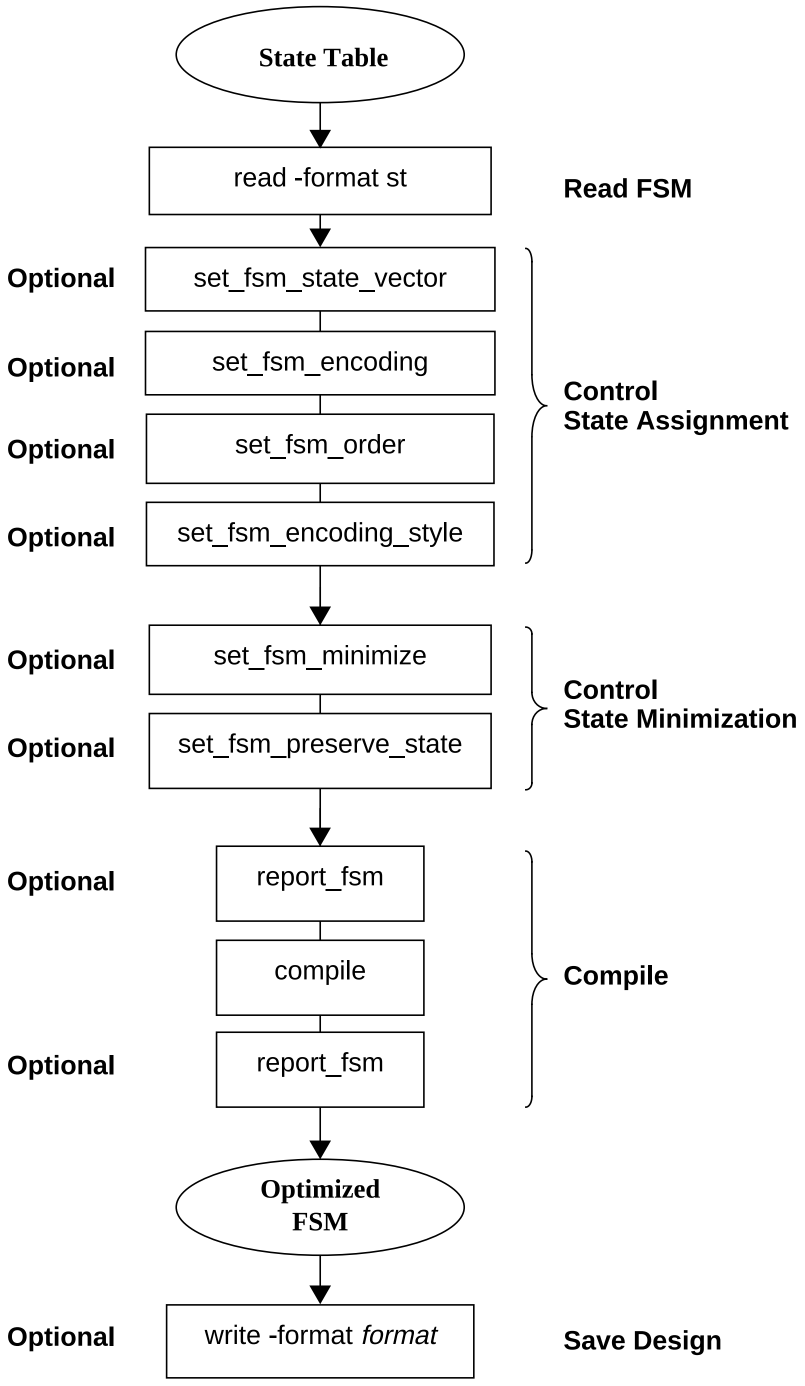
```
dc_shell> write -format st -output my_fsm.st
```

Now that the design is in FSM format, FSM-specific optimizations can be applied, as described in the following section.

Compiling a Finite State Machine

Figure 10-4 shows the flow from a state table description to an optimized FSM, including optional commands.

Figure 10-4 Compiling an FSM From a State Table Description



Reading in the Design

If the design is not already in memory (from a previous extraction), read in an FSM from a state table description. For example, enter

```
dc_shell> read -format st my_fsm.st
```

Controlling State Assignment

Optionally, you can set the encoding, ordering, and generation of state vector values. You can use a combination of manual and automatic state assignment described in “Encoding the States Manually and Specifying a State Assignment Style” on page 10-33.

1. Use the `set_fsm_state_vector` command to name the FSM state vector flip-flops in order. For example, enter

   ```
   dc_shell> set_fsm_state_vector {ff0, ff1, ...}
   ```

2. To manually name some states and their values, enter

   ```
   dc_shell> set_fsm_encoding {"STATE0=0", "STATE1=1", ...}
   ```

3. To manually define the ordering of states and have Design Compiler automatically number the state vectors in binary or gray encoding, enter

   ```
   dc_shell> set_fsm_order {STATE0, STATE1, STATE2, STATE3,
   ...}
   dc_shell> set_fsm_encoding_style binary
   dc_shell> set_fsm_encoding_style gray
   ```

4. To direct Design Compiler to automatically generate the state vectors for a fast FSM (one-hot) where only one bit per state is used, enter

   ```
   dc_shell> set_fsm_encoding_style one_hot
   ```

5. To direct Design Compiler to automatically generate any unassigned state vector encodings (except the one-hot design), enter

   ```
   dc_shell> set_fsm_encoding_style auto
   ```

6. To direct Design Compiler to automatically generate all state encodings, enter

   ```
   dc_shell> set_fsm_encoding { }
   dc_shell> set_fsm_encoding_style auto
   ```

Controlling State Minimization

1. Optionally, you can tell Design Compiler to remove redundant states from the FSM during optimization. For example, enter

   ```
   dc_shell> set_fsm_minimize true
   ```

2. Optionally, you can name the states to be preserved during optimization:

   ```
   dc_shell> set_fsm_preserve_state {STATE0, STATE7...}
   ```

Compiling the Finite State Machine

1. Before compiling an FSM, you can get a precompilation FSM report that shows the existing state table information (see “Creating Finite State Machine Reports” at the end of this chapter). Enter

   ```
   dc_shell> report_fsm
   ```

2. Compile an FSM into an optimized circuit. Enter

   ```
   dc_shell> compile
   ```

3. After compiling an FSM, you can get a postcompilation FSM report that shows the optimized state table information and lists redundant (removed) states. Enter

   ```
   dc_shell> report_fsm
   ```

Saving the Design

Save the FSM as a circuit in any supported output format. For example, to save the design in EDIF format, enter

```
dc_shell> write -format edif -output my_fsm.edif
```

Finite State Machine Synthesis Example

This example shows the complete state table description of a simple vending control unit for a soft drink machine. The commands for generating the design and the schematic generated for the design follow the example.

```
# Soft drink machine -- Price is 15 cents

# Assumptions:Only one coin is deposited at a time
# and only one coin is processed per clock cycle

.design      soft_drink_machine

# Inputs:  clock and reset signals;
#          nickel, dime, and quarter input signals
.inputnames  clk reset nickel_in dime_in quarter_in

# Outputs:  nickel change, dime change, dispense drink
.outputnames nickel_out dime_out dispense

# Clock signal name and type
.clock clk rising_edge

# Asynchronous reset signal, type, and reset state
.asynchronous_reset reset rising IDLE

# State table
```

```
100 IDLE       FIVE      000
010 IDLE       TEN       000
001 IDLE       IDLE      011
100 FIVE       TEN       000
010 FIVE       IDLE      001
001 FIVE       IDLE      111
100 TEN        IDLE      001
010 TEN        IDLE      101
001 TEN        OWE_DIME  011
%%% OWE_DIME   IDLE      010

# Wait in current state until money is deposited
000 IDLE       IDLE      000
000 FIVE       FIVE      000
000 TEN        TEN       000
```

Commands for Generating the Vending Control Unit

1. Read the design in state table format from a file named vending.st. Enter

   ```
   dc_shell> read -format st vending.st
   ```

2. Perform state minimization during optimization. Enter

   ```
   dc_shell> set_fsm_minimize true
   ```

3. Define a circuit constraint (minimum circuit area). Enter

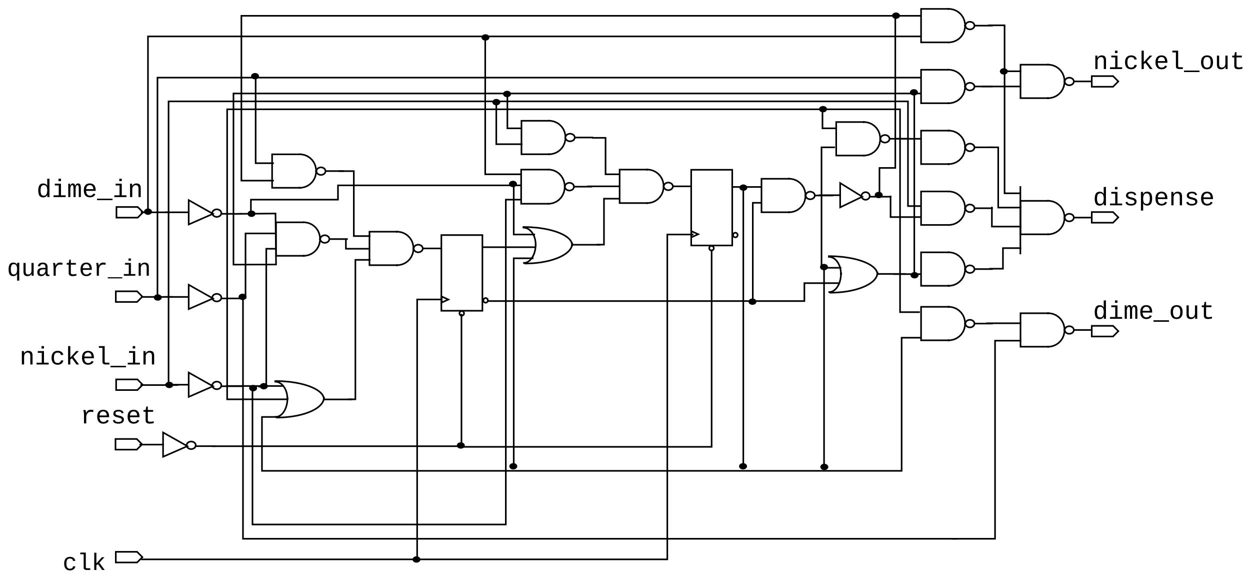
   ```
   dc_shell> set_max_area 0
   ```

4. Compile the FSM into an optimized circuit. Design Compiler performs state minimization, state assignment, and logic optimization. Enter

   ```
   dc_shell> compile
   ```

Figure 10-5 shows the schematic generated for the vending control unit.

Figure 10-5 Schematic Generated for the Vending Control Unit



Extracting a Finite State Machine

If your design is already in state table format, you can skip this section.

FSM extraction is the process of creating a technology-independent representation of an FSM’s behavior (a state table). The initial FSM design is typically a netlist or an HDL description.

A state table describes the next-state transitions and output values as a function of input conditions and the present state of an FSM. States in the machine are represented in terms of symbolic names rather than specific bit values. After a state table is extracted for a design, you can write it to a file in a state table format to serve as technology-independent documentation for the design.

A design represented as a state table enables Design Compiler's FSM optimization in addition to area and speed optimizations. An example is the encodings that represent each state; the number of bits in these encodings can be changed. Redundant states can be removed.

Some types of designs are represented more efficiently in their original form than as extracted FSMs. For example, counters and designs that have registered outputs typically cannot be improved by state minimization or state assignment.

During extraction, the design must be flattened before being represented as a state table. For some designs, flattening can be difficult or impossible. Some large designs can have enough existing structure that optimization (compilation) with a low flatten effort produces better results than extraction followed by optimization.

State Machine Extraction Example

This example demonstrates FSM extraction with a small EDIF netlist implementation of an FSM. The process described here is independent of how the design is represented.

This FSM detects whether its serial input signal (serial) includes a 1 0 1 sequence. If this sequence is detected, the FSM output signal (found_seq) is set to 1.

Figure 10-6 shows the state transition diagram.

Figure 10-6 State Transition Diagram

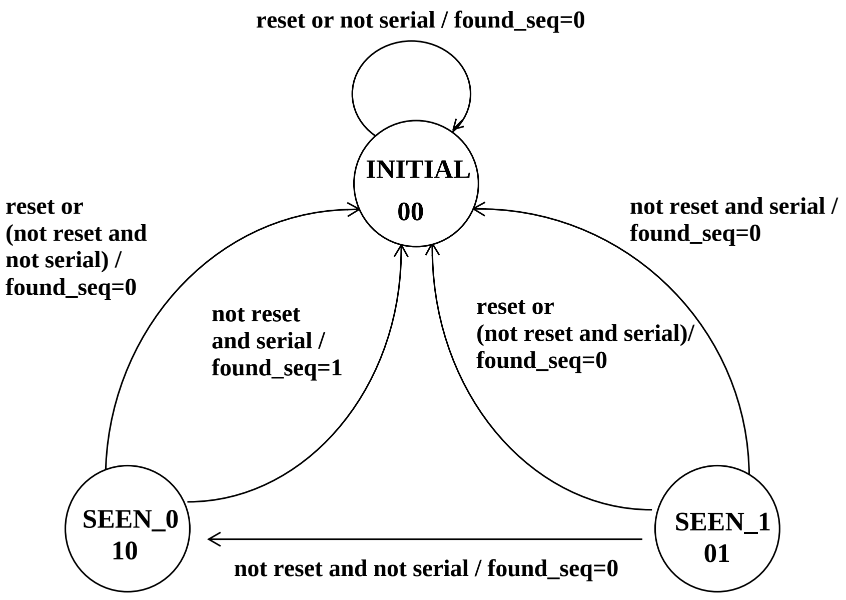
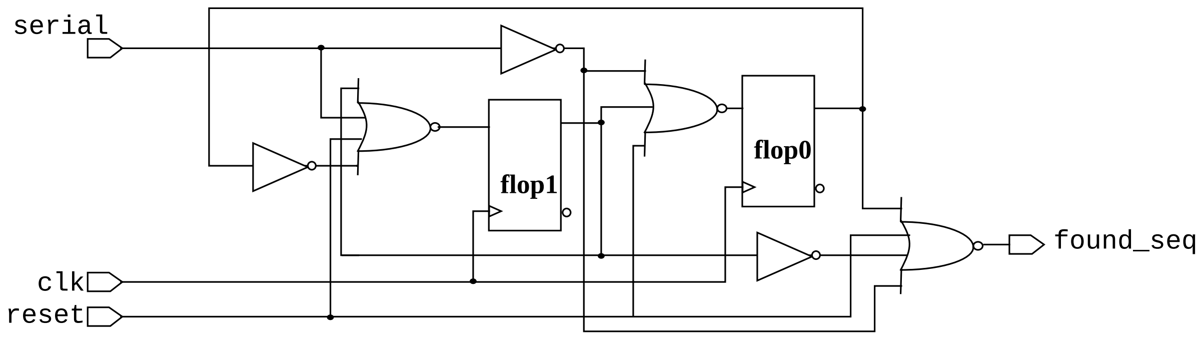


Figure 10-7 shows the corresponding schematic version.

Figure 10-7 Schematic



The extract Command

The simplest way to extract an FSM is to read in the design, list the state vector flip-flops in order, then use the `extract` command:

```
dc_shell> read -format verilog state_machine.v
dc_shell> extract
```

By default, Design Compiler assumes that all noncombinational components (usually flip-flops) in the design are state vector flip-flops. Combinational components connected to the flip-flops are considered part of the FSM.

After extraction, the design is represented as a state table. Design Compiler automatically generates the state names in the resulting state table.

After extraction, you can write out the design in state table format, using the `write` command. For example,

```
dc_shell> write -format st -output state_machine.st
```

Extracting the Finite State Machine Subset of a Circuit

To extract a subset of flip-flops for an FSM, name the state vector flip-flops, then group them with their attached logic as an FSM. For example,

```
dc_shell> read -format edif \
          circuit_with_state_machine.edif
dc_shell> set_fsm_state_vector {flop0, flop1}
dc_shell> group -fsm -design FSM_group
dc_shell> current_design = FSM_group
dc_shell> extract
```

The set_fsm_state_vector Command

The `set_fsm_state_vector` command explicitly names each flip-flop used to store the state vector in the FSM. This command requires a list of instance (component) names. The number of instances in the list determines the length of the state encodings used to represent each state.

The list of state vector names maps directly to the encodings for all states. The first element in the instance list stores the farthest-left (most significant) bit of each state encoding, the second element in the list stores the next bit of each state encoding, and so on.

For example, to encode four states, using 2 bits with flip-flops named ff1 and ff0, enter

```
dc_shell> set_fsm_state_vector { ff1 ff0 }
```

The set of all state encodings is

00

01

10

11

Instance ff1 stores the left column for the state encodings, and ff0 stores the right column.

Use the `set_fsm_state_vector` command to name the flip-flops used for an FSM in a larger design. After the state vector flip-flops are specified, use the `group -fsm` command to isolate them and their associated logic. For the `extract` command, `set_fsm_state_vector` determines the ordering and association of flip-flops to state vector encoding bits.

The group -fsm Command

The `group -fsm` command examines flip-flops in the current design and groups the associated combinational logic. Use the `set_fsm_state_vector` command to specify the state vector subset of a larger design's set of flip-flops.

The result of using `group -fsm` is a new level of design hierarchy containing only the FSM portion of the larger circuit's netlist.

Extracted State Tables

The following state table was extracted (without use of `set_fsm_encoding`) from the FSM shown previously.

```
.design sequence_detector

.inputnames clk serial reset
.outputnames found_seq
.clock clk rising_edge

-1 S0 S0 0
0- S0 S0 0
10 S0 S1 0
-1 S1 S0 0
10 S1 S1 0
00 S1 S2 0
-- S2 S0 ~
-1 S2 S0 0
0- S2 S0 0
10 S2 S0 1
-- S3 S0 0
```

```
.encoding
S0 2#00
S1 2#01
S2 2#10
S3 2#11
```

This state table matches the functionality of the state transition diagram, with two differences:

- The state names do not correspond to those in the transition diagram.
- The state table has four states (S0 to S3), although the state transition diagram has only three states (INITIAL, SEEN_0, SEEN_1).

  Because this FSM has two flip-flops storing the state of the machine, this machine has four possible states. That is, the number of possible states (encodings) is 2n, where n is the number of flip-flops in the state vector. The example machine does not use all possible states. Without information from the designer, Design Compiler assumes that the machine contains the maximum number of states, resulting in the four states in the state table, as opposed to the original three.

Format of Extracted State Tables

Specify extracted state tables by listing all combinations of input conditions and valid states. Abbreviations for combinations of input conditions (%, *, &, and +) are not used. (See Appendix D, “State Table Design Format,” for an explanation of these characters in state tables.)

Reducing Extracted State Tables

Extracted state tables usually are not very compact. To make a state table easier to read, use the `reduce_fsm` command.

The `reduce_fsm` command reduces the logic for each state transition on the current design represented as a state table.

The syntax is

```
reduce_fsm
```

The `reduce_fsm` command has no arguments. After the logic reduction, the design is still represented as a state table.

Typically, the `reduce_fsm` command is useful only following a state table extraction.

Extracting Finite State Machines Efficiently

The extraction example shows that all possible states in the machine are extracted by default. Quite often, only a subset of all possible states are actually used in the FSM. If only the valid states are specified, a smaller state table is extracted. To specify only these valid states, use the `set_fsm_encoding` and `set_fsm_state_vector` commands.

If no state encodings are set, the maximum supported state vector length is 20 bits.

The set_fsm_encoding Command

The `set_fsm_encoding` command enumerates all valid states and their binary encodings.

The syntax is

```
set_fsm_encoding encoding_list
```

The encoding list is a list of the states in the current design with the assigned bit encoding. Enclose each state name and the encoding in double quotation marks (“ ”). If the encoding list contains more than one state, enclose the list in braces ({ }).

Example

To assign code to two states in the machine, enter

```
dc_shell> set_fsm_encoding {"IDLE=2#00" "FIVE=2#01" }
```

The `set_fsm_encoding` command accepts a list of pairs: symbolic state names and their binary encodings. When you specify these state encodings, you must provide the names of the flip-flops and the ordering of the state vector flip-flops, using the `set_fsm_state_vector` command (see “Minimal Finite State Machine Extraction Example” on page 10-28” for more information).

After setting the encoding of one or more states, you can use the `extract -reachable` command to extract the FSM containing all states reachable from the specified (encoded) states from a netlist-format design. For more information, see “Interpreting State Codes When Only Some States Are Known” later in this chapter.

Defining Symbolic State Names

Enclose symbolic state names and their encoding in double quotation marks:

```
"STATE_X=17"
```

Use one of the following two formats to specify encoding values:

- Use a base specifier number (2, 8, 10, or 16), followed by a pound sign (#), followed by a string of digits in the given base. You can separate the string of digits with underscores (_) to make the encodings easier to read.

  Using this format, you can enter the number 15 in the following ways:

  ```
   2#1111
   8#17
  10#15
     15
  16#f
  ```

  If you omit the base specification number and the # character, the string of digits is interpreted as a decimal value.

- Use the caret (^) character, followed by a base specification character (binary, octal, decimal, or hexadecimal—in uppercase or lowercase), followed by a string of digits in the given base.

  Using this format, you can enter the number 15 in the following ways:

  ```
  ^B1111
  ^O1_7
  ^D15
  ^Hf
  ```

All specified state encodings must have the same length.

Example

Consider a simple machine with four states: IDLE, FIVE, TEN, and OWE_DIME. To manually assign 2-bit codes to these four states, enter

```
dc_shell> set_fsm_encoding { "IDLE=0", "FIVE=1",
          "TEN=2",  "OWE_DIME=3" }
```

Minimal Finite State Machine Extraction Example

Following is a sequence of commands to extract the previous state table example.

```
dc_shell> read -f edif state_machine.edif
dc_shell> set_fsm_encoding { "INITIAL=2#00",
         "SEEN_1=2#01", "SEEN_0=2#10" }
dc_shell> set_fsm_state_vector { flop1, flop0 }
dc_shell> extract
dc_shell> reduce_fsm
```

1. Read in the design.
2. Specify that the FSM has states INITIAL, SEEN_1, and SEEN_0 with encodings 2#00, 2#01, and 2#10.
3. Specify the instance names of the flip-flops storing the state vector in the machine. For this example, cell instance flop1 stores the farthest left bit of the state encoding and flop0 stores the farthest right bit.

The extracted state table (using set_fsm_encoding) follows:

```
.design sequence_detector
.inputnames clk serial reset
.outputnames found_seq
.clock clk rising_edge
```

```
-1 INITIAL INITIAL 0
0- INITIAL INITIAL 0
10 INITIAL SEEN_1  0
-- SEEN_0  INITIAL ~
-1 SEEN_0  INITIAL 0
0- SEEN_0  INITIAL 0
10 SEEN_0  INITIAL 1
-1 SEEN_1  INITIAL 0
00 SEEN_1  SEEN_0  0
10 SEEN_1  SEEN_1  0

.encoding
INITIAL 2#00
SEEN_1  2#01
SEEN_0  2#10
```

Interpreting State Codes When Only Some States Are Known

In the previous extraction examples, all valid states are specified before extraction. This assumes that all valid state codes are known.

If only some of the valid states are known, use the `extract` command `-reachable` option to interpret any specified state codes set by `set_fsm_encoding` as an initial set of valid states rather than as the set of all valid states.

Any state that can be reached from a valid state is considered a valid state and is extracted. In some cases, such as with incompletely specified designs, invalid states can be extracted. In incompletely specified designs, some state transitions are unspecified for some input condition and can be considered don’t care transitions. If the FSM is optimized with this don’t care information, the resulting extracted state table can have transitions to invalid states and transitions that did not exist in the original design.

At least one initial valid state must be specified before use of `extract -reachable`. Some designs can have more than one chain of valid states, each starting from a different initial state. Other designs have multiple initial states, where some initial states are not reached from any other state. In such cases, each initial state must be specified before use of `extract -reachable`.

Design Compiler automatically generates names for reachable states that are not initial states.

The -reachable Option

With the `extract` command `-reachable` option, the command sequence for extracting the previous example is

```
dc_shell> read -f edif state_machine.edif
dc_shell> set_fsm_encoding { "INITIAL=2#00" }
dc_shell> set_fsm_state_vector { flop1, flop0 }
dc_shell> extract -reachable
dc_shell> reduce_fsm
```

The resulting state table follows. Substituting state names SEEN_1 for ST_2 and SEEN_0 for state ST_1 gives exactly the same state table as in the earlier state table example.

```
.design state_machine
.inputnames clk serial reset
.outputnames found_seq
.clock clk rising_edge

-1 INITIAL INITIAL 0
0- INITIAL INITIAL 0
10 INITIAL ST_1    0
-- ST_2    INITIAL ~
-1 ST_2    INITIAL 0
0- ST_2    INITIAL 0
10 ST_2    INITIAL 1
-1 ST_1    INITIAL 0
```

```
10 ST_1    ST_1    0
00 ST_1    ST_2    0

.encoding
INITIAL 2#00
ST_1    2#01
ST_2    2#10
```

Finite State Machine Extraction Assumptions

FSM extraction makes some fundamental assumptions that restrict the types of designs that can be extracted.

Initial Design

All ports of the initial design must be either input ports or output ports. Inout ports cannot be represented in a state table.

Because the state table format is a two-level representation of the FSM, it must be possible to flatten the initial design.

The design must consist only of combinational logic components and the flip-flops that hold the state vector. Other noncombinational components are not allowed. Combinational logic that does not depend on the state vector is accurately represented, although combinational feedback loops are not supported. Use the `group -fsm` command to get the current design into this form. The state vector must be specified by `set_fsm_state_vector` before use of `group -fsm`. All combinational logic in the transitive fanin and fanout of the state vector elements, not including ports and noncombinational components, is grouped into a new design. Extraction can then be performed on this new design, which is a subdesign of the initial design.

State Machine Hardware Model

The architectural model for FSMs supported by Design Compiler includes a single clock and an optional synchronous or asynchronous reset signal. For extraction, these signals must connect *only* to every state vector element and must have the same rising or falling sense. The clock or reset signals cannot be gated signals, but they can be driven by buffers or inverters.

The reset state of the machine is automatically determined during the extraction process by noting how the asynchronous reset signal is connected to the set or reset pins of each state vector flip-flop. The extracted encoding of the asynchronous reset state must be a valid state of the machine.

Synchronous reset signals are incorporated into the state table.

Compiling Finite State Machines

Design Compiler optimizes designs by using logic-level and gate-level optimization techniques. You use constraints to control the type of design produced and the area and speed tradeoff of the design. For designs represented as FSMs, Design Compiler uses two additional optimization techniques:

- State minimization
- State assignment

You can vary other aspects of the FSM architecture, such as the number of bits used to encode the states and the choice of sequential elements used to store the current state of the machine.

Reading In Finite State Machine Designs

Use the `read` command `-format st` option to read in a state table description file. When you read in a state table description, the resulting design is tagged as an FSM design rather than as the usual internal netlist-format designs.

You can specify FSM behavior by using the Design Compiler state table input format. The syntax of this format is described in detail in Appendix D, "State Table Design Format."

The state table format specifies the state transition behavior for an FSM. You describe the input and output ports for the design, the clock signal and its sense, optional asynchronous or synchronous reset signals or states, a table describing the state transitions, and an optional assignment of codes for symbolic states in the machine.

Encoding the States Manually and Specifying a State Assignment Style

State encodings used internally to represent each state affect the logic required to implement the design and, consequently, the area and speed. The choice of state encodings is a critical step in the design process. You can assign the state encodings arbitrarily or based on the transition behavior of the machine. The latter approach can be a tedious way to determine a good set of encodings. It is difficult to know how good the encoding selection is before the logic is implemented and optimized.

- Use the `set_fsm_encoding` command to manually encode some or all states.

- Use the `set_fsm_encoding_style` command to specify a state assignment method.

  Design Compiler can determine encodings for any states that do not have encodings specified. The supported encoding styles are auto, one_hot, binary, and gray.

The type of encoding style to use depends on the characteristics of the FSM and the area or speed constraints for the design. During state assignment, the circuit constraints specified for the design are not used. Thus, a state assignment always yields the same set of encodings for a given FSM, independent of whether you are optimizing for area or delay.

The set_fsm_encoding Command

To manually encode one or more states and set the ordering of state names, use the `set_fsm_encoding` command. Manually assigned codes are not reassigned by any automatic state assignment.

The set_fsm_encoding_style Command

Design Compiler ensures that all states have been assigned encodings during the generation of the logic for the FSM. If you manually assigned encodings for any states in the design, these codes are not reassigned by auto state assignment.

If some states have not been assigned an encoding, Design Compiler chooses an encoding for each of them. This process is called state assignment. There are different ways to perform state assignment. The `set_fsm_encoding_style` command specifies which of these methods to use.

The length of the state code is based on one of the following criteria:

- Bit length of any manually assigned encodings
- Number of instance names specified in the `set_fsm_state_vector` command

If the length of the encodings cannot be determined by these criteria, the bit length is a function of the encoding style specified.

The choices of encoding styles are auto, one_hot, binary, and gray.

auto State Encoding

The auto encoding style uses a proprietary algorithm in which the primary objective is to determine a set of encodings that best reduces the complexity of the combinational logic while using the minimum number of encoding bits.

This encoding style is targeted toward optimizations in which you want minimum area. The effects of speed as a result of the encoding are not addressed by this algorithm. If you decrease the amount of combinational logic, some circuits become faster.

The maximum supported state vector length for auto encoding is 30 bits.

Example

To assign encodings to all states in the machine, use the automatic state assignment algorithm. Enter

```
dc_shell> set_fsm_encoding { }
dc_shell> set_fsm_encoding_style auto
```

Example

To manually encode some states, have the automatic state assignment algorithm determine the best encodings for all remaining states. Enter

```
dc_shell> set_fsm_encoding { "RESET=2#111", ...}
dc_shell> set_fsm_encoding_style auto
```

If any state encodings remain undetermined, the auto encoding style chooses an encoding length that minimizes the FSM.

one_hot State Encoding

The one_hot encoding style uses 1 unique bit in the state vector corresponding to each state in the machine. The state codes are all 0s except for the state's particular bit position, which is set to 1.

Encoding states in this manner reduces the combinational logic. For example, to determine whether the machine is in a given state, the full state encoding does not need to be decoded—only a single bit in the state vector needs to be tested. If the machine is in the given state, the bit is 1; if it is not in that state, the bit is 0.

No other portions of the state code need to be examined. In addition, you need to change only 2 bits in the state vector to transition from one state to another: Set the bit representing the previous state to 0, and set the bit representing the next state to 1. The remainder of the state code remains at 0.

Although the combinational logic is simplified, the area of the sequential elements that store the state of the machine is increased, because there now is one flip-flop per state. Consequently, one_hot encoded machines are usually larger than machines encoded with auto, although they tend to have smaller propagation delays between states. For machines in which speed is critical, you can optimize the design for speed by using both the auto and the one_hot encoding styles and keep the best design.

Manually assigned codes (from `set_fsm_encoding`) are ignored in the one_hot encoding style.

If the number of elements in the state vector does not equal the number of states, a warning appears and the length of the code is reset to the number of states in the machine.

Example

To encode a machine by using the one_hot encoding style, enter

```
dc_shell> set_fsm_encoding_style one_hot
```

The binary and gray State Encoding Styles

The binary and gray encoding styles are convenient for encoding states by using known encodings without having to manually specify each state and its encoding.

Manually assigned codes (from `set_fsm_encoding`) are ignored in the binary or gray encoding style.

These encoding styles sequentially assign codes to all states using a binary numbering or gray numbering sequence.

- The binary numbering sequence consists of using binary values to encode the sequence of counting integers. For example, four states encoded with 2 bits are numbered 0, 1, 2, and 3, represented by the encodings 00, 01, 10, and 11. The code length is the binary logarithm (log2) of the number of states in the machine.

- The gray numbering sequence assigns codes to states so that successive codes differ by only 1 bit. In this case, the same four states encoded in the gray encoding style are numbered 0, 1, 3, and 2, represented by the encodings 00, 01, 11, and 10. In the sequence of gray encodings, only one bit position changes value in going from one to the next (for example, 01 to 11).

To assign numbers to states with either binary or gray numbering sequences, the states must be ordered. States are ordered by the way they are entered in the state table or can be manually assigned with the `set_fsm_order` command, as described in the following section.

Defining an Ordering of States

The `set_fsm_order` command defines an ordering of states in the current design.

Ordering the states is important only when you use the binary or gray encoding styles. Using `set_fsm_order` with an encoding style of binary and gray is more convenient than specifying the same information by using the `set_fsm_encoding` command.

- The previously assigned order is removed and replaced by the new order specified in the state list for this command.
- States not included in the state list are given an arbitrary order at the end of the specified order.
- An empty state list removes the ordering of all states in the design.

Example

Given the states S0, S1, S2, S3, S4, and S5, the following commands are equivalent:

```
set_fsm_encoding_style binary
set_fsm_order {S0 S1 S2 S3 S4 S5}
```

is equivalent to

```
set_fsm_encoding {"S0=0", "S1=1", "S2=2", "S3=3",
"S4=4","S5=5"}
```

Example

Given the states S0, S1, S2, S3, S4, and S5, the following commands are equivalent:

```
set_fsm_encoding_style gray
set_fsm_order {S0 S1 S2 S3 S4 S5}
```

is equivalent to

```
set_fsm_encoding {"S0=2#000", "S1=2#001", "S2=2#011", \
     "S3=2#010", "S4=2#110", "S5=2#111"}
```

Understanding State Encoding Length

The one-hot encoding style can be used to simplify the complexity of the combinational logic portion of an FSM, but it increases the sequential element count and overall circuit area. The one-hot encoding style is an extreme example of increasing the length of the state encoding.

In some instances, increasing the state encoding length can lead to a decrease in overall circuit area. The area decrease can sometimes be realized for FSMs where the number of states in the machine is close to the maximum number of states that can be represented with the current state encoding length. For example, an FSM with 15 states can be encoded with the minimum code length of 4 bits, with one code being unused. This results in a set of encodings whose truth table contains only one don't care condition. In this case, the algorithm required to implement this encoding scheme cannot be reduced and can lead to a more complex design.

Example

If the encoding length is increased to 5 bits, the maximum number of unique codes is 32, which leaves 17 unused codes. The extra flexibility provided by 17 don't care conditions can result in a set of encodings with a decreased combinational logic requirement that compensates for the increased area attributable to the additional sequential element.

To specify an increased state encoding length size for the previous example, use the following commands:

```
dc_shell> set_fsm_state_vector { ff4 ff3 ff2 ff1 ff0 }
dc_shell> set_fsm_encoding { }
dc_shell> set_fsm_encoding_style auto
```

The first command specifies five state vector, flip-flop instance names indicating a state encoding length of 5 bits. Next, all existing encodings are removed so Design Compiler can find the most-efficient encodings for all states in the machine. Finally, Design Compiler is instructed to use the proprietary auto state assignment algorithm.

Design Compiler can use fewer than five encoding bits; the five state vector flip-flops indicate a maximum of 5 bits.

Reducing the Number of States (State Minimization)

State minimization reduces the number of states in an FSM without changing the external behavior of the design. Minimization involves achieved by finding sets of equivalent states and replacing each equivalent set by a new state.

Performing State Minimization

State minimization can be performed in one of two ways:

- On a state table design during compilation when the `set_fsm_minimize` command is used before `compile`:

  ```
  dc_shell> compile
  ```

- If the `minimize_fsm` command takes the current FSM design, performs state minimization, and returns the design still as a state table design:

  ```
  dc_shell> minimize_fsm
  ```

During the state minimization process, certain states can be preserved by use of the `set_fsm_preserve_state` command with those state names. For example, to keep state S0 in the design, enter

```
dc_shell> set_fsm_preserve_state { S0 }
dc_shell> minimize_fsm
```

State S0 remains in the design. If any other states are equivalent to S0, those states are removed and state S0 is preserved.

Preserving States during Minimization

The set_fsm_preserve_state command specifies that certain states must not be removed from the design during state minimization.

Example

Assume that machine states S0, S1, and S2 are equivalent. Although you can remove two states from the design by merging these three states into one state, you want to keep these states in the machine. To preserve states S0, S1, and S2 in the design, enter

```
dc_shell> set_fsm_preserve_state { S0 S1 S2 }
```

If you prefer to keep state S1 and do not care whether states S0 and S2 are merged into state S1, enter

```
dc_shell> set_fsm_preserve_state { S1 }
```

Optimizing a Finite State Machine

After an FSM has been read in or extracted, it can be optimized (compiled) to a target technology. Before compiling, you can use `reduce_fsm` to shrink the state table and `minimize_fsm` to remove redundant states. The `compile` command can minimize the state table if `set_fsm_minimize` is run before compilation.

Setting Finite State Machine Attributes

After a design is in state table format, its synthesis attributes can be specified as state encoding style, state ordering, and preferred state vector components.

Specifying Sequential Elements

Specify the sequential elements that store the state vector, using the `set_register_type` (default flip-flop type in a design) and `set_register_type` (specific flip-flop type used for a given cell) commands.

Compiling the Design

After reading an FSM description and specifying attributes for the design, you can perform state assignment and logic generation on the design. These processing steps are incorporated into the `compile` command.

The set_fsm_minimize Command

The `set_fsm_minimize` command specifies whether state minimization is performed on the state table design during compilation. State minimization identifies and removes equivalent states in the FSM without changing the input or output behavior of the machine.

By default, state minimization is not performed on a design during compilation.

To specify state minimization during compilation, use the command

```
dc_shell> set_fsm_minimize true
```

To turn off state minimization during compilation, use the command

```
dc_shell> set_fsm_minimize false
```

The compile Command

If the current design is an FSM, the `compile` command performs the following actions:

state minimization

> Identifies equivalent states and replaces them with one arbitrarily chosen state. State minimization is enabled only if the `set_fsm_minimize` command is set to true.

state assignment

> Checks the encodings for each state (from `set_fsm_encoding`). If all states have been assigned codes, state assignment is not necessary. Otherwise, the type of state

assignment is based on the encoding style specified by the `set_fsm_encoding_style` command. If no encoding style has been specified, the default state assignment style is auto.

logic minimization

Minimizes the combinational logic portion of the design. The type of minimization is set with the `set_flatten` command `-minimize` option. During the minimization, the output, next state, and encoding don't care sets help minimize the amount of combinational logic required to implement the FSM. When minimization is being performed because a state table representation is already flat, a flattening step later during compilation would be redundant and is automatically disabled.

logic generation

Describes the combinational logic portion of the design as Boolean equations. State vector storage elements are instantiated as technology-independent generic flip-flops.

If the current design is hierarchical, the special processing for FSMs is performed on each FSM subdesign. However, for hierarchical designs, flattening is not automatically disabled. After all FSMs are processed, the `compile` command proceeds as usual, performing the logic-level and gate-level optimizations governed by the `compile` options you specified. (See Chapter 3, "Optimizing Designs.")

Verifying a Finite State Machine

The `compare_fsm` command compares two netlist-format designs to determine whether the sequential behavior of the two designs is equivalent. Verification is possible for all netlists produced by the FSM compiler, as well as for other netlists. The designs must be completely specified but need not have the same state vector length or encoding.

The syntax of the `compare_fsm` command is

```
compare_fsm fsm_design1 fsm_design2
```

The arguments `fsm_design1` and `fsm_design2` are the names of the two netlist-format designs you want compared.

Before running `compare_fsm`, define each design's initial state and its state vector flip-flops. For example, enter

```
dc_shell> read FSM_design1
dc_shell> set_fsm_encoding { "START = 2#11" }
dc_shell> set_fsm_state_vector {ff1 ff0}
dc_shell> read FSM_design2
dc_shell> set_fsm_encoding { "FIRST_STATE = 2#010" }
dc_shell> set_fsm_state_vector {XF1 XF2 XF3}
dc_shell> compare_fsm FSM_design1 FSM_design2
```

Note:

Asynchronous reset conditions are not verified.

Creating Finite State Machine Reports

The `report_fsm` command creates an FSM report which includes the following information:

- Name of the clock signal and its sense
- Name of the optional asynchronous reset signal, its sense, and its state name
- Encoding bit length and encoding style
- State vector flip-flop names and ordering
- A list of state names and encodings in state order
- Preserved states and merged (equivalent) states

A typical FSM report follows.

```
dc_shell> report_fsm
****************************************
Report : fsm
Design : BUS_ARBITRATOR
Version: v1997.01
Date   : Tues Jan 14 1997
****************************************
Clock             : CLK         Sense: rising_edge
Asynchronous Reset: Unspecified

Encoding Bit Length: 3
Encoding style     : auto
```

```
State Vector: { FF2 FF1 FF0 }

State Encodings and Order:

Grant_A    : 001
Wait_A     : 011
Timeout_A1 : 111
Grant_B    : 010
Wait_B     : 110
Timeout_B1 : 101

Preserved States: Grant_A

Merged States: None
```

11

Analyzing Timing

After optimizing a design, generate reports and schematics to analyze your timing, area, and component selection results.

When you study the reports, note the constraint violations in the design. You might have to apply special compile techniques to overcome violations.

Report commands display information about the current design. Most reports can also be run on the current instance.

This chapter contains the following sections:

- Timing-Analysis Basic Concepts
- Analyzing Paths
- Generating Reports Command Summary
- Displaying Net Information

- Reporting Fanout Logic
- Reporting Fanin Logic
- Tracing Cones of Logic
- Identifying Cells and Startpoints or Endpoints of Critical Paths
- Updating Timing
- Reporting Timing
- Determining Net Delays From the Timing Report
- Reporting False Paths Attributable to Resource Sharing
- Reporting a Carry-Bypass Adder as the Longest Path
- Tracing Paths Through inout Pins and Ports
- Doing Timing Checks With the report_timing Command
- Timing Report With Options to the report_timing Command

Timing-Analysis Basic Concepts

Timing analysis is primarily for ensuring that the setup and hold requirements of a design’s sequential elements are met and that specific path requirements in a design are satisfied.

Static timing analysis checks the timing of all possible paths in a design against the design requirements. This technique provides speed of operation and critical path identification. The disadvantages are that false paths are often identified as being in violation and that the analysis typically is restricted to the synchronous portions of a design.

The Design Compiler static timing analyzer traces and analyzes all signal paths in a design. Unlike with dynamic timing analysis, no input test vectors are required in order to validate design timing requirements. The timing analyzer determines the actual path delays of the design and compares them with the required path delays. The delay calculations the DesignTime tool from Synopsys performs are the same as the delay calculations you perform on the design, but they are calculated much faster.

The timing analyzer computes each gate and interconnect delay, then traces critical paths, calculating minimum and maximum arrival times to points of interest. The timing analyzer uses the critical path values to evaluate design constraints and create timing reports.

Analyzing Paths

After the timing analyzer computes all the local gate delays, it applies a critical path tracing algorithm to the network. This algorithm computes the minimum and maximum path delay for every path in the design.

Path delays are computed by adding delay values through chains of gates and interconnects. The timing analyzer considers the unateness of each gate's output pin when combining delay values, (see Appendix A, "Static Timing Analysis"). For example, when delay values are being added through an inverter, the incoming rise delay is added to the fall delay of the inverter and the incoming fall delay is combined with the rise delay of the inverter.

Paths starting from power or ground connections are not timed, because these connections cannot propagate rising or falling state transitions.

Generating Reports Command Summary

Table 11-1 summarizes the Design Compiler commands for generating reports for the current design, current instance, and postcompilation.

Table 11-1 Summary of Commands to Generate Reports

Command	**Current design**	**Current instance**	**Postcompile**
`all_connected`			X
`all_critical_cells`			X

Table 11-1 Summary of Commands to Generate Reports (continued)

Command	Current design	Current instance	Postcompile
all_critical_paths			X
all_fanin			X
all_fanout			X
all_inputs			X
all_outputs			X
all_registers			X
check_design			X
check_timing			X
derive_clocks			X
report_annotated_delay	X	X	
report_area	X	X	X
report_attribute	X	X	X
report_bus	X	X	
report_cache	X	X	
report_cell	X	X	X
report_clock	X		X
report_compile_options	X	X	
report_constraint	X		X
report_delay_calculation	X	X	
report_design	X	X	
report_design_lib	X		
report_hierarchy	X	X	

Table 11-1   Summary of Commands to Generate Reports (continued)

Command	**Current design**	**Current instance**	**Postcompile**
`report_name_rules`	X	X	
`report_names`	X	X	
`report_net`	X	X	X
`report_port`	X	X	X
`report_reference`	X	X	
`report_timing`			X
`report_timing_requirements`	X	X	X
`report_resources`	X		
`report_transitive_fanin`	X	X	X
`report_transitive_fanout`	X	X	X
`report_wire_load`	X		
`report_xref`	X	X	

Displaying Net Information

The `report_net`, `report_internal_loads`, and `report_annotated_delay -net` commands display net information.

Listing Nets in the Current Design or Instance

The `report_net` command lists all the nets in the current design or the current instance.

The syntax is

```
report_net [-noflat] [-nosplit] [-transition_times]
    [-cell_degradation] [-connections]
    [-verbose] net_list
```

For detailed information, see *Design Compiler User Guide* and the man page.

Listing Only Nets With Annotated Loads

The `report_internal_loads` command lists the nets that have annotated loads.

For detailed information, see *Design Compiler User Guide* and the man page.

Listing Back-Annotated Net Loads

The `report_net` and `report_annotated_delay -net` commands list nets with annotated loads in the current design or current instance.

Example

```
dc_shell> report_annotated_delay -net
****************************************
Report : annotated -net
Design : counter
Version: v1997.08
Date : Tues Apr 23 1997
****************************************
Net Name    From    To    Rise    Fall    Load    Res.
----------------------------------------------------------
h            ffc/QN  w/A  200.00   200.00  50.00  200.00
h            ffc/QN  m/B  200.00   200.00  50.00  200.00
h            ffc/QN  r/B   50.00   200.00
```

Reporting Fanout Logic

The `report_transitive_fanout` command displays fanout logic within the boundaries set by the `current_instance` variable.

If `current_instance` is defined, the reports show the hierarchical boundary pins on the current instance.

The syntax is

```
report_transitive_fanout [-clock_tree]
        [-from list] [-nosplit]
```

`-clock_tree`

Lists the fanout logic for clocks in the current design by levels of logic.

`-from` *`list`*

Lists the fanout logic for the pins, nets, and ports.

`-nosplit`

Disables line splitting in the report output when information exceeds its column’s width.

Example

```
dc_shell> report_transitive_fanout -clock_tree

****************************************
Report : transitive_fanout
        -clock_tree
Design : inverted
Version: v1997.01
Date   : Tues Jan 14 1997
****************************************

The fanout network of source clk is as follows:

Driver Pin       Load Pin          Type        Sense
```

```
----------------------------------------------------------
clk             clk_bar/A         (net arc)   same
clk             ff1/CP            (net arc)   same
clk             ff3/CP            (net arc)   same


Load Pin        Driver Pin        Type        Sense
----------------------------------------------------------
clk_bar/A       clk_bar/Z         INV         opposite


Driver Pin      Load Pin          Type        Sense
----------------------------------------------------------
clk_bar/Z       ff2/CP            (net arc)   opposite
clk_bar/Z       ff4/CP            (net arc)   opposite
```

Reporting Fanin Logic

The `report_transitive_fanin` command displays the logic fanning in to a given net, pin, or port.

The `report_transitive_fanin` command reports information within the boundaries set by the `current_instance` variable. If `current_instance` is defined, the reports show the hierarchical boundary pins on the current instance.

The syntax is

`report_transitive_fanin [-to` *`list`*`] [-nosplit]`

`-to` *list*

Lists the fanin logic for the pins, nets, and ports.

`-nosplit`

Disables line splitting in the report output when information exceeds its column’s width.

Example

```
dc_shell> current_instance i2
Current instance is ’top/i2’.
{"i2"}
dc_shell> report_transitive_fanin -to ./out_pin
Performing report_transitive_fanin on pin ’i2/out_pin’.
****************************************
Report : transitive_fanin
Design : top/i2
Version: v3.4
****************************************
The fanin network of sink i2/out_pin is as follows:
Driver Pin    Load Pin   Type       Sense
-----------------------------------------------------------
i2/c/a/Z      i2/out_pin      (net arc)  same
Load Pin      Driver Pin            Type       Sense
-----------------------------------------------------------
i2/c/a/A i2/c/a/Z IVA opposite
Driver Pin Load Pin Type Sense
-----------------------------------------------------------
i2/b/Z   i2/c/a/A (net arc) opposite
Load Pin Driver Pin Type Sense
-----------------------------------------------------------
i2/b/A   i2/b/Z IVA same
Load Pin Driver Pin Type Sense
-----------------------------------------------------------
i2/b/A   i2/in_pin (net arc)same
```

Tracing Cones of Logic

The `all_fanin` and `all_fanout` commands trace cones of logic. Use these commands to identify and isolate portions of a design for special processing or analysis.

- `all_fanin` traces the cone of logic leading to a specified point.
- `all_fanout` traces the cone of logic beginning at a specified point.

For detailed information about the `all_fanin` and `all_fanout` commands, see *Design Compiler User Guide*.

Identifying Cells and Startpoints or Endpoints of Critical Paths

The `all_critical_cells` and `all_critical_pins` commands identify the cells in critical paths (paths with the largest negative slack) and their startpoints or endpoints. In their default use,

- `all_critical_cells` returns the set of cells on the critical path.
- `all_critical_pins` returns the startpoint or endpoint of the critical path.

Both commands are similar to the `all_fanin` and `all_fanout` commands but are more specific in their scope and allow you to identify certain portions of your design for other processing or analysis.

The `all_critical_cells` and `all_critical_pins` commands are described in detail in *Design Compiler User Guide*.

Updating Timing

The `update_timing` command runs the timing analyzer on the current design. Although timing analysis is run by various other commands such as `compile` and `report_timing`, manually retiming the current design can be helpful. For example, if you use `remove_attribute` to modify port drive, use `update_timing` to update the affected timing path values.

Reporting Timing

The `report_timing` command creates timing reports for the current design or the current instance. It lists the full path of the longest maximum delay timing path for each path group.

Note:

Before you run `report_timing`, link your design.

The syntax is

```
report_timing [-to to_pin_list] [-from from_pin_list]
     [-through through_pin_list]
     [-path short | full | only | end]
     [-delay min | min_rise|min_fall|max|max_rise| max_fall]
     [-nworst paths_per_endpoint]
     [-max_paths max_path_count] [-input_pins] [-nets]
     [-transition_time] [-lesser_path max_path_delay]
     [-greater_path min_path_delay] [-loops]
     [-true [-true_threshold path_delay]]
     [-justify] [-enable_preset_clear_arcs]
     [-significant_digits digits]
     [-nosplit]
```

With no options, the `report_timing` command reports the most critical maximum path (the path with the worst slack). The timing analyzer reports only constrained paths based on worst slack computations; unconstrained paths do not appear in the default timing reports. The cell delay and net delay associated with the net connecting the previous cell with the current cell are bundled and displayed next to the output pin of the cell. For example, in the following report excerpt, 0.67 includes the delay of cell u1/o1 and the delay of the net connecting cell u1/a1 and cell u1/o1.

```
Point            Incr    Path
-----------------------------------------
...
u1/a1/Z (AN2)    0.67    1.82
u1/o1/Z (AN2)    0.67    2.49
...
```

If the design has some back-annotated data when `report_timing` is run, Design Compiler runs the `update_timing` command to calculate the delays for nets that are not fully annotated. For example, if some nets have no back-annotated SDF timing or RC loads, their delay is calculated with the current wire load model for that block. This results in every object having an annotated delay (some from back-annotation and some from calculation).

If a design is changed— for example, by a change in process parameters or the creation of custom wire load models— Design Compiler reruns `update_timing` to recalculate the timing.

The report_timing Command Options Summary

Table 11-2 summarizes `report_timing` command options. For detailed information about `report_timing` and its options, see the man page.

Note:

> The `-justify`, `-true`, and `-true_threshold` `path_delay` options to the `report_timing` command require a DC Ultra license.

Table 11-2   report_timing Command Option Summary

To Report This	**Use this option**
Paths to specified pins, ports, or clocks.	-to *pin_list*
Paths from specified pins, ports, or clocks.	-from *pin_list*
Paths through specified pin.	-through *pin_list*
Path startpoint and endpoint.	-path short
Full path.	-path full
Path only, with no required time or slack calculation.	-path only
Endpoints, with path total, required time, and clock.	-path end
Path type at the endpoint (default: max).	-delay min
Specified number of worst paths per endpoint (default: 1).	-nworst *path_count*
Specified maximum number of paths per path group (default: 1).	-max_paths *number*
Paths having input as well as output pins.	-input_pins
Nets in the path and the fanout for those nets. (-nets does not report the delay; it displays 0.00 for each net.)	-nets
Net transition time for each driving pin.	-transition_time
Transition times at each cell on the path being reported.	-transition
Paths with a delay less than specified. (Use with -greater_path to select paths inside or outside a given delay range.)	-lesser_path *max_path_delay*
Paths with a delay greater than specified. (Use with -lesser path to select paths inside or outside a given delay range.)	-greater_path *min_path_delay*
Only the timing. Do not use with `-justify`.	-loops

Table 11-2 report_timing Command Option Summary (continued)

To Report This	Use this option
Longest (least slack) true paths.	-true. (Requires a DC Ultra license.)
First path greater than or equal to the specified path delay. (Use with `-true`.)	-true_threshold path_delay. (Requires a DC Ultra license.)
An input vector that sensitizes the reported paths or that reports that the path is false if no input vector is found. Do not use with `-loops`.	-justify. (Requires a DC Ultra license.)
Timing with arcs enabled.	-enable_preset_clear_arcs
A specified number of digits (0 to 13) to the right of the decimal point (default: 2).	-significant_digits *digits*
Disable line splitting when information exceeds its column's width.	-no_split

Listing Paths to, From, or Through Specified Pins

To list only the paths to, from, or through specified pins, use these options to the `report_timing` command:

`-to` *pin_list*

`-from` *pin_list*

`-through` *pin_list*

Only paths starting at one of the `-from` points, going through all of the `-through` points, and ending at one of the `-to` points are considered. The pins specified by `-through` need not be in topological order. For example,

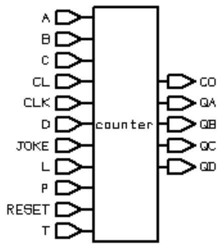
```
report_timing -from {A B} -through {L M} -to {Y Z}
```

must contain points (A or B), (L and M), and (Y or Z).

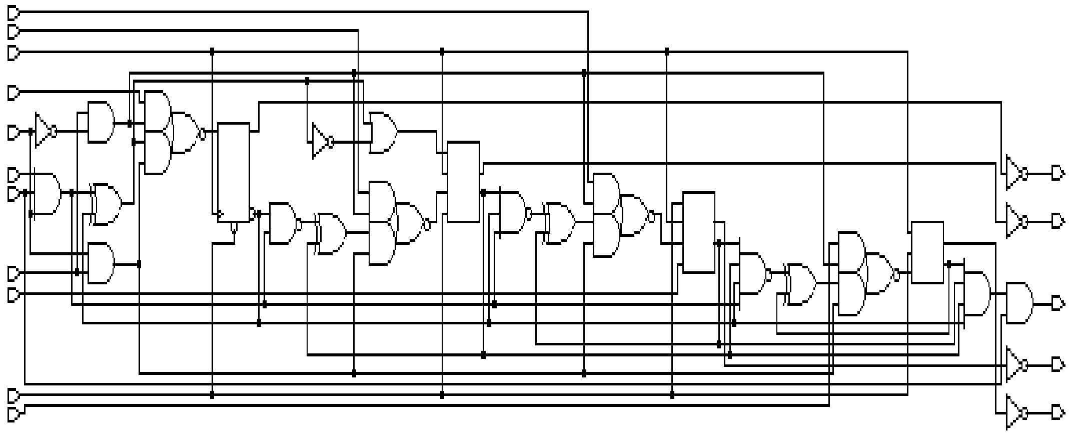
Default Timing Report Example

Figure 11-1 shows a counter design. The `report_timing` command with no options generates the default timing report that follows the figure. Options (in this case, the defaults) are listed at the top of the report.

Figure 11-1 Counter Design



Symbol View



Schematic View

```
dc_shell> report_timing

****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : counter
Version: v1997.08
Date   : Tues Apr 23 1997
****************************************

Operating Conditions:
Wire Loading Model Mode: top

Startpoint: ffb (rising edge-triggered flip-flop clocked by CLK)
  Endpoint: ffd (rising edge-triggered flip-flop clocked by CLK)
  Path Group: CLK
  Path Type: max

 Point                                     Incr       Path
 -----------------------------------------------------------
  clock CLK (rise edge)                     0.00       0.00
  clock network delay (ideal)               0.00       0.00
  ffb/CP (DFF)                              0.00       0.00 r
  ffb/QN (DFF)                              2.42       2.42 r
  w/Z (NAND4)                               0.59       3.01 f
  q/Z (EO)                                  1.13       4.14 f
  j/Z (AO2)                                 1.08       5.22 r
  ffd/D (DFF)                               0.00       5.22 r
  data arrival time                                    5.22

  clock CLK (rise edge)                    10.00      10.00
  clock network delay (ideal)               0.00      10.00
  ffd/CP (DFF)                              0.00      10.00 r
  library setup time                       -0.90       9.10
  data required time                                   9.10
  -----------------------------------------------------------
  data required time                                   9.10
  data arrival time                                   -5.22
  -----------------------------------------------------------
  slack (MET)                                          3.88
```

Determining Net Delays From the Timing Report

Determine the net delays by using one of the following commands:

- `report_delay_calculation`
- `report_timing -input_pins`

The report_delay_calculation Command

The `report_delay_calculation` command reports detailed timing calculation information about a specified cell or net timing arc.

For more information, see the `report_delay_calculation` man page.

The report_timing -input_pins Command

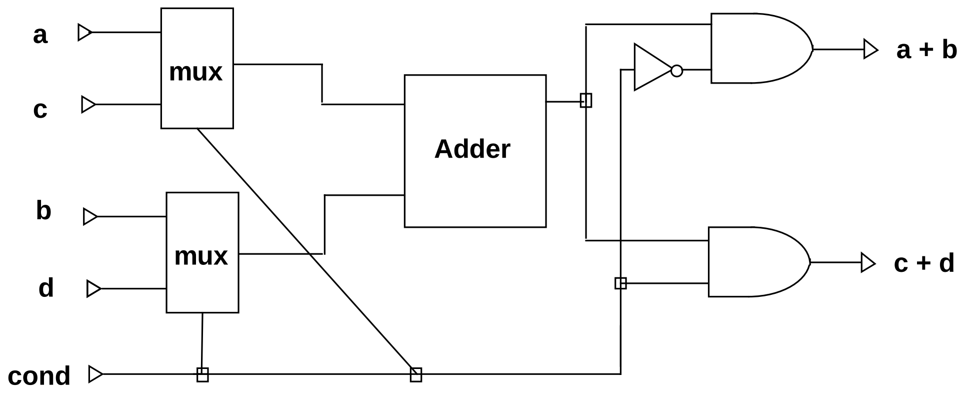
The `report_timing -input_pins` command generates a timing report that includes the input pins in the path. The delays of the nets feeding into the input pins are associated with the input pins.

In order to use the `report_delay_calculation` command on a design, the target library must have the `library_features (report_delay_calculation);` variable set. For additional information, refer to the Library Compiler documentation.

For detailed information, see the `report_timing` man page.

Reporting False Paths Attributable to Resource Sharing

In this example, the adder is shared between the multiplexed inputs a + b and c + d. The paths from a to c + d are false. If a arrives late and c + d is required early, `report_timing` reports a violation attributable to the false path.



```
dc_shell> set_input_delay 10 {a}
1
dc_shell> set_output_delay 10 {c_d}
1
dc_shell> max_delay 15,all_outputs()
1
dc_shell> report_timing
```

The `report_timing` command generates the following report:

```
****************************************
Report : timing
Design : FP_SHR
Version: v1997.01
Date   : Tues Jan 14 1997
****************************************
Operating Conditions:
Wire Loading Model Mode: top
```

```
  Startpoint: a (input port)
  Endpoint: c_d (output port)
  Path Group: default
  Path Type: max

  Point                                      Incr       Path

-----------------------------------------------------------
  input external delay                     10.00      10.00 r
  a (in)                                    0.00      10.00 r
  m1/Z (MUX21H)                             1.00      11.00 r
  u1/S (FA1)                                1.00      12.00 r
  c_d/Z (AN2)                               1.00      13.00 r
  c_d (out)                                 0.00      13.00 r
  data arrival time                                   13.00

  max_delay                                  15.00     15.00
  output external delay                     -10.00      5.00
  data required time                                    5.00

-----------------------------------------------------------
  data required time                                    5.00
  data arrival time                                   -13.00

-----------------------------------------------------------
  slack (VIOLATED)                                     -8.00
```

Reporting a Carry-Bypass Adder as the Longest Path

In a carry-bypass adder, adding logic turns the long path of the carry chain into a false path to provide a shorter path to the output (in the paths where the carry would propagate down the entire chain). In a carry-bypass adder design, `report_timing` reports the carry chain as the longest path.

```
dc_shell> report_timing
****************************************
Report : timing
Design : false12cbp2
Version: 31997.01
Date   : Wed Jan 14 1997
****************************************
  Startpoint: s (input port)
  Endpoint: x[11] (output port)
  Path Group: default
  Path Type: max
  Point                                    Incr       Path
  -----------------------------------------------------------
  input external delay                     0.00       0.00 r
  s (in)                                     0.00       0.00 r
  U27/z1937                                 1.00       1.00 f
  U257/z2167                                1.00       2.00 f
  U233/z2143                                1.00       3.00 f
  U234/z2144                                1.00       4.00 r
  U239/z2149                                1.00       5.00 r
  U108/z2018                                1.00       6.00 r
  ...
  U199/z2109                                1.00      37.00 r
  U200/z2110                                1.00      38.00 r
  U167/z2077                               1.00      39.00 r
  U1/z1911                                1.00      40.00 r
  x[11] (out)                              0.00      40.00 r
  data arrival time                                  40.00
  max_delay                            0.00       0.00
  output external delay                0.00       0.00
  data required time                              0.00
  -----------------------------------------------------------
  data required time                                  0.00
  data arrival time                                 -40.00
  -----------------------------------------------------------
  slack (VIOLATED)                                  -40.00
```

Tracing Paths Through inout Pins and Ports

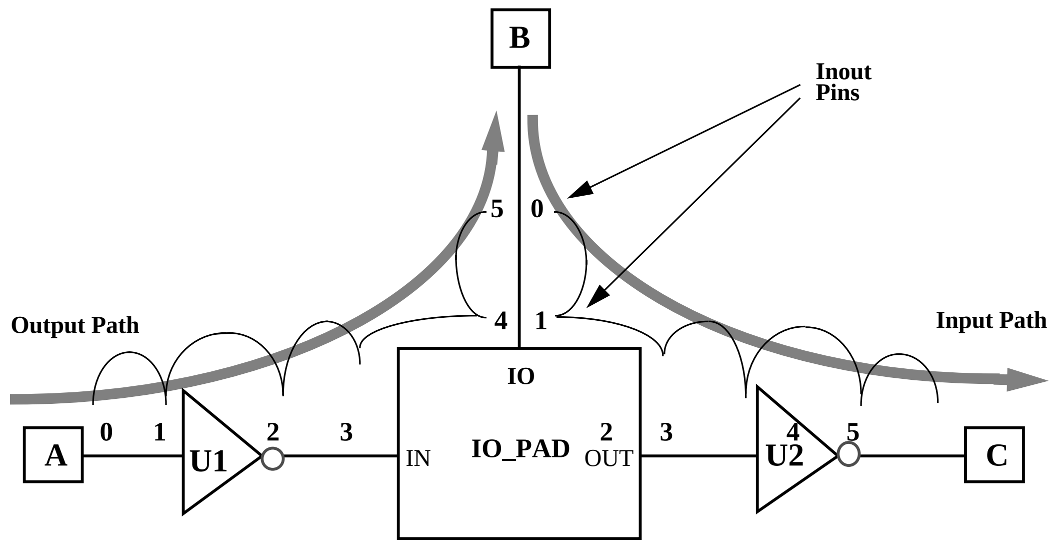
Most cells in a library have either input or output pins. There is only one path to trace through such a pin. Some complex cells have inout pins. The timing analyzer treats inout pins as two pins: one input and one output.

To differentiate between paths through an inout pin, the timing analyzer uses the following rules:

- When it approaches an inout pin from a net, the timing analyzer treats that pin as a load (input).
- When it approaches an inout pin from a cell, the timing analyzer treats that pin as a driver (output).

Figure 11-2 shows an example using inout ports and pins.

Figure 11-2 Timing Paths of inout Pins and Timing Report



In Figure 11-2, B is an inout port and IO is an inout pin of cell IO_PAD. Each of these is on one of two distinct timing paths. Assuming a 1-ns delay for each gate and net, path delay totals are shown next to each pin.

Doing Timing Checks With the report_timing Command

The `report_timing` command provides the following timing checks:

- Recovery and removal timing
- Nonsequential timing
- No-change timing
- Clock gating timing

Recovery and Removal Timing Check

The `report_timing` command can report recovery and removal timing. To do so,

- The library you are using must support this functionality. Look in the library file for information and the syntax description.
- The `enable_recovery_removal_arcs` variable must be set to true.

The following report shows recovery and removal timing.

```
Startpoint: r1 (input port)
  Endpoint: count_reg[2] (rising-edge recovery check against clock clock)
  Path Group: clock
  Path Type: max
  Point                                    Incr       Path
  -----------------------------------------------------------
  input external delay                     0.00       0.00 f
  r1 (in)                                  0.00       0.00 f
  U78/A (IVP)                              0.00       0.00 f
  U78/Z (IVP)                              0.16       0.16 r
  U79/B (MUX21L)                           0.00       0.16 r
  U79/Z (MUX21L)                           0.31       0.47 f
  U56/B (ND2P)                             0.00       0.47 f
  U56/Z (ND2P)                             0.42       0.89 r
  count_reg[2]/CD (FJK3S)                  0.00       0.89 r
  data arrival time                                   0.89

  max_delay                                1.50       1.50
  clock network delay (ideal)              0.50       2.00
  clock uncertainty                       -0.50       1.50
  library recovery time                   -0.75       0.75
  data required time                                  0.75
  -----------------------------------------------------------
  data required time                                  0.75
  data arrival time                                  -0.89
  -----------------------------------------------------------
  slack (VIOLATED)                                   -0.14
```

```
Startpoint: r1 (input port)
Endpoint: count_reg[2] (rising-edge removal check against clock clk1)
Path Group: clk1
Path Type: min

Point                                   Incr       Path
-----------------------------------------------------------
clock (input port clock) (rise edge)     0.00       0.00
clock network delay (ideal)              0.00       0.00
input external delay                     0.00       0.00 f
r1 (in)                                  0.00       0.00 f
U83/A (EN)                               0.00       0.00 f
U83/Z (EN)                               0.64       0.64 f
U61/B (ND2)                              0.00       0.64 f
U61/Z (ND2)                              0.73       1.37 r
count_reg[2]/CD (FJK3S)                  0.00       1.37 r
data arrival time                                   1.37

clock clk1 (rise edge)                   0.00       0.00
clock network delay (ideal)              0.50       0.50
clock uncertainty                        0.30       0.80
count_reg[2]/CP (FJK3S)                  0.30       0.80 r
library removal time                     0.52       1.32
data required time                                  1.32
-----------------------------------------------------------
data required time                                  1.32
data arrival time                                  -1.37
-----------------------------------------------------------
slack (MET)                                         0.05
```

Nonsequential Timing Check

The `report_timing` command provides a nonsequential timing check.

```
 Startpoint: d4/out_reg[1] (rising edge-triggered flip-flop clocked by clk)
Endpoint: m42/U25/U1 (non-sequential rising-edge timing check clocked by clk’)
 Path Group: clk
 Path Type: max

 Point                                    Incr       Path
 -----------------------------------------------------------
 clock clk (rise edge)                    0.00       0.00
 clock network delay (ideal)              0.50       0.50
 d4/out_reg[1]/CP (FD1QA)                 0.00       0.50 r
 d4/out_reg[1]/Q (FD1QA)                  0.75       1.25 f
 d4/out[1] (dff4)                         0.00       1.25 f
 U26/C (AND3A)                            0.00       1.25 f
 U26/Z (AND3A)                            0.45       1.70 f
 m42/din[3] (mux4to1_1)                   0.00       1.70 f
 m42/U25/D3_0 (_MUX_OP_4_2.1)             0.00       1.70 f
 m42/U25/U2/B (MUX21HB)                   0.00       1.70 f
 m42/U25/U2/Z (MUX21HB)                   0.56       2.26 f
 m42/U25/U1/D2 (MUX31HC)                  0.00       2.26 f
 data arrival time                                   2.26

 clock clk’ (rise edge)                   4.00       4.00
 clock network delay (ideal)              0.50       4.50
 clock uncertainty                       -0.50       4.00
 m42/U25/U1/A (MUX31HC)                   0.00       4.00 r
 library setup time                      -0.45       3.55
 data required time                                  3.55
 -----------------------------------------------------------
 data required time                                  3.55
 data arrival time                                  -2.26
 -----------------------------------------------------------
 slack (MET)                                         1.29

 sequential falling-edge propagation clocked by clk’)
```

```
Endpoint: d11/out_reg (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max

Point                                    Incr       Path
-----------------------------------------------------------
clock clk’ (fall edge)                   0.00       0.00
clock network delay (ideal)              0.50       0.50
m41/U25/U1/B (MUX31HC)                   0.00       0.50 f
m41/U25/U1/Z (MUX31HC)                   0.50       1.00 f
m41/U25/Z_0 (_MUX_OP_4_2.1_1)            0.00       1.00 f
m41/dout (mux4to1_0)                     0.00       1.00 f
d11/d (dff_1)                            0.00       1.00 f
d11/out_reg/D (FD1QA)                    0.00       1.00 f
data arrival time                                   1.00

clock clk (rise edge)                    8.00       8.00
clock network delay (ideal)              0.50       8.50
clock uncertainty                       -0.50       8.00
d11/out_reg/CP (FD1QA)                   0.00       8.00 r
library setup time                      -0.25       7.75
data required time                                  7.75
-----------------------------------------------------------
data required time                                  7.75
data arrival time                                  -1.00
-----------------------------------------------------------
slack (MET)                                         6.75
```

No Change Timing Check

The `report_timing` command provides a no-change timing check.

```
Startpoint: EN1 (level-sensitive input port clocked by CLK)
Endpoint: UCKLENH (positive nochange timing check clocked by CLK’)
Path Group: CLK
Path Type: max

Point                       Fanout      Incr        Path
-----------------------------------------------------------
clock CLK (rise edge)                   0.00        0.00
clock network delay (ideal)             0.00        0.00
input external delay                    1.00        1.00 r
EN1 (in)                                0.00        1.00 r
EN1 (net)                     1         0.00        1.00 r
U24/A (N1A)                             0.00        1.00 r
U24/Z (N1A)                             0.07        1.07 f
E112 (net)                    1         0.00        1.07 f
U25/A (N1B)                             0.00        1.07 f
U25/Z (N1B)                             0.08        1.15 r
E113 (net)                    1         0.00        1.15 r
U26/A (N1C)                             0.00        1.15 r
U26/Z (N1C)                             0.07        1.21 f
E114 (net)                    1         0.00        1.21 f
U27/A (N1D)                             0.00        1.21 f
U27/Z (N1D)                             0.04        1.25 r
E115 (net)                    2         0.00        1.25 r
UCKLENH/EN (LD1QC_HH)                   0.00        1.25 r
data arrival time                                   1.25

clock CLK’ (rise edge)                  2.00        2.00
clock network delay (ideal)             0.00        2.00
UCKLENH/G (LD1QC_HH)                    0.00        2.00 r
library nochange setup time            -0.32        1.68
data required time                                  1.68
-----------------------------------------------------------
data required time                                  1.68
data arrival time                                  -1.25
-----------------------------------------------------------
slack (MET)                                         0.43
```

```
Startpoint: EN1 (level-sensitive input port clocked by CLK)
Endpoint: UCKLENH (positive nochange timing check clocked by CLK’)
Path Group: CLK
Path Type: min

Point                        Fanout      Incr       Path
-----------------------------------------------------------
clock CLK (rise edge)                    0.00       0.00
clock network delay (ideal)              0.00       0.00
input external delay                     1.00       1.00 f
EN1 (in)                                 0.00       1.00 f
EN1 (net)                      1         0.00       1.00 f
U24/A (N1A)                              0.00       1.00 f
U24/Z (N1A)                              0.08       1.08 r
E112 (net)                     1         0.00       1.08 r
U25/A (N1B)                              0.00       1.08 r
U25/Z (N1B)                              0.08       1.16 f
E113 (net)                     1         0.00       1.16 f
U26/A (N1C)                              0.00       1.16 f
U26/Z (N1C)                              0.07       1.23 r
E114 (net)                     1         0.00       1.23 r
U27/A (N1D)                              0.00       1.23 r
U27/Z (N1D)                              0.05       1.28 f
E115 (net)                     2         0.00       1.28 f
UCKLENH/EN (LD1QC_HH)                    0.00       1.28 f
data arrival time                                   1.28

clock CLK’ (fall edge)                   0.00       0.00
clock network delay (ideal)              0.00       0.00
UCKLENH/G (LD1QC_HH)                     0.00       0.00 f
library nochange hold time               0.32       0.32
data required time                                  0.32
-----------------------------------------------------------
data required time                                  0.32
data arrival time                                  -1.28
-----------------------------------------------------------
slack (MET)                                         0.96
```

Clock Gating Timing Check

The `report_timing` command provides a clock gating timing check. This example assumes that your design contains library cells modeled with no_change/setup/hold timing checks in the input pin section, targeted as gated clock cell NR3_GAT.

```
Startpoint: g4 (input port clocked by clock)
Endpoint: GL_NR3 (negative nochange timing check clocked by clock)
Path Group: clock
Path Type: max

Point                                    Incr       Path
-----------------------------------------------------------
clock clock (rise edge)                  0.00       0.00
clock network delay (propagated)         0.00       0.00
input external delay                     0.50       0.50 f
g4 (in)                                  0.00       0.50 f
GL_NR3/B (NR3_GAT)                       0.00       0.50 f
data arrival time                                   0.50

clock clock (fall edge)                  2.50       2.50
clock network delay (propagated)         0.37       2.87
clock uncertainty                       -0.70       2.17
GL_NR3/CLK (NR3_GAT)                     0.00       2.17 f
library nochange setup time             -1.78       0.39
data required time                                  0.39
-----------------------------------------------------------
data required time                                  0.39
data arrival time                                  -0.50
-----------------------------------------------------------
slack (VIOLATED)                                   -0.11
```

Timing Report With Options to the report_timing Command

This section provides example timing reports using various options to the `report_timing` command, including

- Longest path without required time and slack calculation
- Endpoint path delay, required time, and slack for each path
- Startpoint and endpoint delay
- Input pins and default values
- Input pins and nets but no required time and slack calculation

Longest Path Without Required Time and Slack Calculation

Example 11-1 reports the longest path to Z1, without required time and slack calculation. This example is considered an unconstrained path for optmization. If this occurs you need reconsider the constraints you set for optimization.

Example 11-1 Report on Longest Path Without Timing Constraint Setting

```
dc_shell> report_timing -to z1 -nworst 2 -path only
****************************************
Report : timing
        -path only
        -delay max
        -nworst 2
        -max_paths 2
Design : led
Version: v3.1a
Date   : Tue Apr  7 16:52:43 1992
****************************************

Operating Conditions:
Wire Loading Model Mode: top


  Startpoint: c (input port)
  Endpoint: z1 (output port)
  Path Group: default
  Path Type: max

  Point                                     Incr       Path
  -----------------------------------------------------------
  input external delay                      0.00       0.00 f
  c (in)                                    0.00       0.00 f
  u1/Z (IVA)                                0.60       0.60 r
  u17/Z (AO7)                               0.53       1.13 f
  u18/Z (OR3)                               1.24       2.37 f
  z1 (out)                                  0.00       2.37 f
  data arrival time                                    2.37
  -----------------------------------------------------------

  Startpoint: d (input port)
  Endpoint: z1 (output port)
  Path Group: default
  Path Type: max

  Point                                     Incr       Path
  -----------------------------------------------------------
  input external delay                      0.00       0.00 f
  d (in)                                    0.00       0.00 f
  u20/Z (IVA)                               0.53       0.53 r
  u17/Z (AO7)                               0.53       1.06 f
  u18/Z (OR3)                               1.24       2.30 f
  z1 (out)                                  0.00       2.30 f
data arrival time                                    2.30
  -----------------------------------------------------------
```

Endpoint Path Delay, Required Time, and Slack

This example shows a report of the endpoint path delay, required time, and slack for each path.

```
dc_shell> report_timing -path end
****************************************
Report : timing
        -path end
        -delay max
Design : led
Version: v3.1a
Date   : Tue Apr  7 16:28:07 1992
****************************************

Operating Conditions:
Wire Loading Model Mode: top


Endpoint                         Path Delay     Path Required     Slack
-----------------------------------------------------------------
z2                                  3.41 f           0.00        -3.41
z3                                  3.03 f           0.00        -3.03
z4                                  2.77 f           0.00        -2.77
z6                                  2.69 r           0.00        -2.69
z0                                  2.59 f           0.00        -2.59
z1                                  2.37 f           0.00        -2.37
z5                                  2.26 f           0.00        -2.26
```

Startpoint and Endpoint Delay

This example reports the startpoint and endpoint of the path from a to z2.

```
dc_shell> report_timing -from a -to z2 -path short
****************************************
Report : timing
        -path short
        -delay max
        -max_paths 1
Design : led
Version: v3.1a
Date   : Tue Apr  7 16:29:40 1992
****************************************

Operating Conditions:
Wire Loading Model Mode: top


  Startpoint: a (input port)
  Endpoint: z2 (output port)
  Path Group: default
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  input external delay                     0.00       0.00 f
  a (in)                                   0.00       0.00 f
  ...
  z2 (out)                                 1.24       1.24 f
  data arrival time                                   1.24

  max_delay                                0.00       0.00
  output external delay                    0.00       0.00
  data required time                                  0.00
  -----------------------------------------------------------
  data required time                                  0.00
  data arrival time                                  -1.24
  -----------------------------------------------------------
  slack (VIOLATED)                                   -1.24
```

Input Pins and Default Values

This example reports the input pins and the default values.

```
dc_shell> report_timing -input_pins
****************************************
Report : timing
        -path full
        -delay max
        -input_pins
        -max_paths 1
Design : led
Version: v3.1a
Date   : Tue Apr  7 16:32:28 1992
****************************************
Operating Conditions:
Wire Loading Model Mode: top

  Startpoint: c (input port)
  Endpoint: z2 (output port)
  Path Group: default
  Path Type: max

  Point                                    Incr       Path
  ----------------------------------------------------------
  input external delay                     0.00       0.00 r
  c (in)                                   0.00       0.00 r
  u1/A (IVA)                               0.00       0.00 r
  u1/Z (IVA)                               0.54       0.54 f
  u0/A (NR2)                               0.00       0.54 f
  u0/Z (NR2)                               1.20       1.74 r
  u8/A (IVA)                               0.00       1.74 r
  u8/Z (IVA)                               0.43       2.17 f
  u7/B (OR3)                               0.00       2.17 f
  u7/Z (OR3)                               1.24       3.41 f
  z2 (out)                                 0.00       3.41 f
  data arrival time                                   3.41

  max_delay                                0.00       0.00
  output external delay                    0.00       0.00
  data required time                                  0.00
  ----------------------------------------------------------
  data required time                                  0.00
  data arrival time                                  -3.41
  ----------------------------------------------------------
  slack (VIOLATED)                                   -3.41
```

Input Pins and Nets but No Other Calculations

This example reports the input pins and nets but does not show the required time and slack calculation.

```
dc_shell> report_timing -input_pins -nets -path only
****************************************
Report : timing
        -path only
        -delay max
        -input_pins
        -nets
        -max_paths 1
Design : led
Version: v3.1a
Date   : Tue Apr  7 16:34:20 1992
****************************************

Operating Conditions:
Wire Loading Model Mode: top

  Startpoint: c (input port)
  Endpoint: z2 (output port)
  Path Group: default
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  input external delay                     0.00       0.00 r
  c (in)                                   0.00       0.00 r
  c (net)                                  0.00       0.00 r
  u1/A (IVA)                               0.00       0.00 r
  u1/Z (IVA)                               0.54       0.54 f
  cell24/n22 (net)                         0.00       0.54 f
  u0/A (NR2)                               0.00       0.54 f
  u0/Z (NR2)                               1.20       1.74 r
  cell24/n21 (net)                         0.00       1.74 r
  u8/A (IVA)                               0.00       1.74 r
  u8/Z (IVA)                               0.43       2.17 f
  cell24/n19 (net)                         0.00       2.17 f
  u7/B (OR3)                               0.00       2.17 f
  u7/Z (OR3)                               1.24       3.41 f
  z2 (net)                                 0.00       3.41 f
  z2 (out)                                 0.00       3.41 f
  data arrival time                                   3.41
  -----------------------------------------------------------
```

Creating Collections and Querying Objects

Design Compiler's Tcl-mode supports a set commands that you can use to create a collection. You can use this collection to pass other commands to analyze your design.

Design Compiler builds an internal database of the netlist and the attributes it applied to the database. This database consists of several classes of objects, such as designs, libraries, ports, cells, nets, pins, clocks, and so on. Most Design Compiler commands operate on these objects.

A collection is a group of objects referred to by a string identifier known as a collection handle. You can create collections of objects, then apply a set of commands to interact with those collections. Collections can be homogeneous (contain objects of one type) or heterogeneous (contain objects of many types).

The collection commands are divided into three categories:

- Commands that create collections of objects for use by another command
- Commands that manipulate collections
- Commands that query objects for you to view

You can use wildcards and filtering criteria to narrow the focus of a collection. You can store collections in variables for use in setting attributes or clocks, or for performing custom reporting.

In Design Compiler, you pass collections to commands by using an identifier called a collection handle. This handle is a string that uniquely identifies the collection. A collection handle is the result of a command that creates collections.

Table 11-3 Tcl-mode Commands for Creating Collections

Commands	Description
`get_cells`	Create a collection of cells
`get_clocks`	Create a collection of clocks
`get_clusters`	Create a collection of clusters
`get_designs`	Create a collection of designs
`get_lib_cells`	Create a collection of lib cells
`get_lib_pins`	Create a collection of lib pins
`get_libs`	Create a collection of libs
`get_multibits`	Create a collection of multibits
`get_nets`	Create a collection of nets
`get_path_groups`	Create a collection of path groups
`get_pins`	Create a collection of pins
`get_ports`	Create a collection of ports
`get_references`	Create a collection of references
`get_timing_paths`	Creates a collection of timing paths for custom reporting and other processing.

Using Paths to Generate Custom Reports

You can use the `get_timing_paths` and the `get_path_groups` commands to create a collection of paths for custom reporting and other processing. You can assign these timing paths to a variable or pass them into another command.

Use the `foreach_in_collection` command to iterate among the paths in the collection. The collection commands `index_collection`, `copy_collection`, `add_to_collection`, and `remove_from_collection` are not applicable to timing path collections.You can use the `get_attribute` command to obtain information about the paths.

Table 11-4 Support Attributes on Timing Paths

Attribute	Type
`clock_uncertainty`	float
`endpoint`	string
`endpoint_clock`	string
`endpoint_clock_close_edge_type`	string
`endpoint_clock_close_edge_value`	float
`endpoint_clock_is_inverted`	boolean
`endpoint_clock_is_propagated`	boolean
`endpoint_clock_open_edge_type`	string
`endpoint_clock_open_edge_value`	float
`endpoint_clock_pin`	string
`endpoint_hold_time_value`	float
`endpoint_is_level_sensitive`	boolean

Table 11-4 Support Attributes on Timing Paths (continued)

Attribute	**Type**
`endpoint_output_delay_value`	float
`endpoint_recovery_time_value`	float
`endpoint_removal_time_value`	float
`endpoint_setup_time_value`	float
`object_class`	string
`path_group`	string
`path_type`	string
`points`	string
`slack`	string
`startpoint`	string
`startpoint_clock`	string
`startpoint_clock_is_inverted`	boolean
`startpoint_clock_is_propagated`	boolean
`startpoint_clock_latency`	float
`startpoint_clock_open_edge_type`	float
`startpoint_clock_open_edge_value`	float
`startpoint_input_delays_value`	float
`startpoint_is_level_sensitive`	boolean
`time_borrowed_from_endpoint`	
`time_lent_to_startpoint`	

One attribute of a timing path is the points collection. A point corresponds to a pin or port along the path. Iterate through these points using the `foreach_in_collection` command and get the attributes on them using the `get_attribute` command.

Table 11-5 Supported Attributes for Points of a Timing Path

Attribute	Type
arrival	string
object	string
object_class	string
rise_fall	string
slack	string

For more information, see the man pages for collections and `foreach_in_collection`.

The syntax for `get_timing_paths` is

```
get_timing_paths [-to to_list] [-from from_list]
         [-through through_list] [-delay_type delay_type]
         [-nworst paths_per_endpoint]
         [-max_paths max_path_count]
         [-enable_preset_clear_arcs]
         [-group group_name] [-true]
         [-true_threshold path_delay]
         [-greater greater_limit]
         [-lesser lesser_limit]
         [-slack_greater_than greater_slack_limit]
         [-slack_lesser_than lesser_slack_limit]
```

For more information, see the `get_timing_paths` man page.

```
get_path_groups [-quiet] [-regexp] [-nocase]
              [-filter expression] patterns
```

For more information, see the `get_paths_clocks` man page.

Selecting Clock Objects

The `get_clocks` command selects clocks for a command to use, for example, to ensure that a command works on the CLK clock and not on the CLK port.

The syntax is

```
get_clocks    [-quiet] [-regexp] [-nocase]
              [-filter expression] patterns
```

For more information, see the `get_clocks` man page.

12

Analyzing Scan Conformance

DC Expert *Plus* provides capabilities described in the following sections of this chapter, to analyze the scan conformance of your design:

- Checking Test-Design Rules
- Generating Fault Coverage Results
- Reporting Test Information

Checking Test-Design Rules

Use the `check_test` command to check the current design for test-design rule violations. If the design has violations, the `check_test` command issues warning or error messages. A warning message indicates a testability problem that lowers the fault coverage of the design. An error message indicates a serious problem that prevents further processing of the design in DC Expert *Plus* until you resolve the problem. See the *Scan Synthesis User Guide* for information about understanding and correcting violations reported during test-design rule checking.

test-design rule checking supports designs in the following scan states:

Nonscan

The design contains nonscan sequential cells.

Unrouted scan

The design contains unrouted scan cells, which can result from test-ready compile or from the scan replacement phase of constraint-optimized scan insertion.

Scan

The design contains routed scan chains, which can be generated by the scan assembly phase of constraint-optimized scan insertion or can be existing scan chains (indicated by the `-existing_scan` option of the `set_scan_configuration` command).

On nonscan and unrouted scan designs, test-design rule checking uses pre-scan checks (including the scan equivalence checks). On scan designs, test-design rule checking uses post-scan checks.

The test-design rule checks include

- Identifying barriers to scan replacement (nonscan designs only)
- Identifying structures that can reduce the fault coverage results
- Validating proper scan operation

See the *Scan Synthesis User Guide* for details about these test-design rule checks.

The syntax for the `check_test` command is

```
check_test [-verbose]
```

`-verbose`

The `-verbose` option causes DC Expert *Plus* to generate warnings for all similar cells or pins involved in the same design rule violation. If you do not designate `-verbose`, DC Expert *Plus* lists warnings only for the first cell or pin that violates a rule, followed by the number of additional violations.

Preparing for Test-Design Rule Checking

Before running the `check_test` command, you must

- Specify the test configuration
- Specify the test timing

Specifying the Test Configuration

Test configuration information includes

- Test modes

- Initialization sequences
- Scan port annotations (for existing scan designs)

Defining a Test Mode

If your design requires a test configuration to satisfy design rules or to enable specific circuit paths, you must define a test mode. A test mode is a set of static logic values (logic 1 or logic 0) that you apply to input ports. The test mode is in effect during the entire scan-test sequence: from scan in, through normal operation, to scan out.

Use the `set_test_hold` command to define the test mode for the current design. The syntax for the `set_test_hold` command is

```
set_test_hold value port_list
```

`value`

Specifies the desired logic value. Valid values are 1 or 0.

`port_list`

Specifies the design ports that are to have the assigned value. To specify more than one port, enclose the list in curly braces ({}) or use the `find` command.

The `set_test_hold` command places the `test_hold` attribute on the specified ports. When you run test-design rule checking, DC Expert *Plus* propagates the values defined at ports with `test_hold` attributes through the circuit and evaluates their effect on the test-design rules.

The conditions you set with the `set_test_hold` command also apply when you run Test Compiler ATPG.

See the *Scan Synthesis User Guide* for more information about using test modes.

Specifying an Initialization Sequence

If your design requires an initialization sequence to configure it for scan testing, you must provide the initialization vectors through an initialization protocol. For more information about initialization protocols, see the *Scan Synthesis User Guide*.

Annotating Scan Ports

If your design is an existing scan design, you must annotate the scan ports to enable DC Expert *Plus* to recognize the scan chains. Your design is an existing scan design if

- DC Expert *Plus* (or the Test Compiler tool from Synopsys) did not create the scan chains.

  or

- DC Expert *Plus* (or Test Compiler) created the scan chains in a previous session and you read the design with an ASCII format.

Use the `set_signal_type` command to annotate the scan ports in an existing scan design. The syntax of the `set_signal_type` command is

```
set_signal_type attribute port_name
```

`attribute`

Describes the type of scan signal. See Table 12-1 for valid values.

`port_name`

Identifies the design port driving the specified scan signal.

Note:

The `set_signal_type` command has other options that are not related to test. See the man pages for a complete list of options.

Table 12-1 Valid Values for the signal_type Attribute Values

Scan signal	**signal_type attribute**
scan input	`test_scan_in`
scan output	`test_scan_out`
scan enable	`test_scan_enable` or`test_scan_enable_inverted`
test scan clock	`test_scan_clock`
a scan clock	`test_scan_clock_a`
b scan clock	`test_scan_clock_b`
test clock	`test_clock`
asynchronous port control	`test_asynch` or`test_asynch_inverted`
bidirectional enable	`test_bidir_control` or`test_bidir_control_inverted`

Specifying the Test Timing

The test timing information includes

Test period

The test period is the duration of a tester cycle.

Input delay time

The input delay time is the time, relative to the start of the tester cycle, at which data is applied to all nonclock inputs.

Bidirectional delay time

The bidirectional delay time is the time, relative to the start of the tester cycle, at which data is applied to all bidirectional ports in input mode and is released from all bidirectional ports in output mode.

Strobe time

The strobe time is the time, relative to the start of the tester cycle, at which the output strobe occurs.

Strobe width

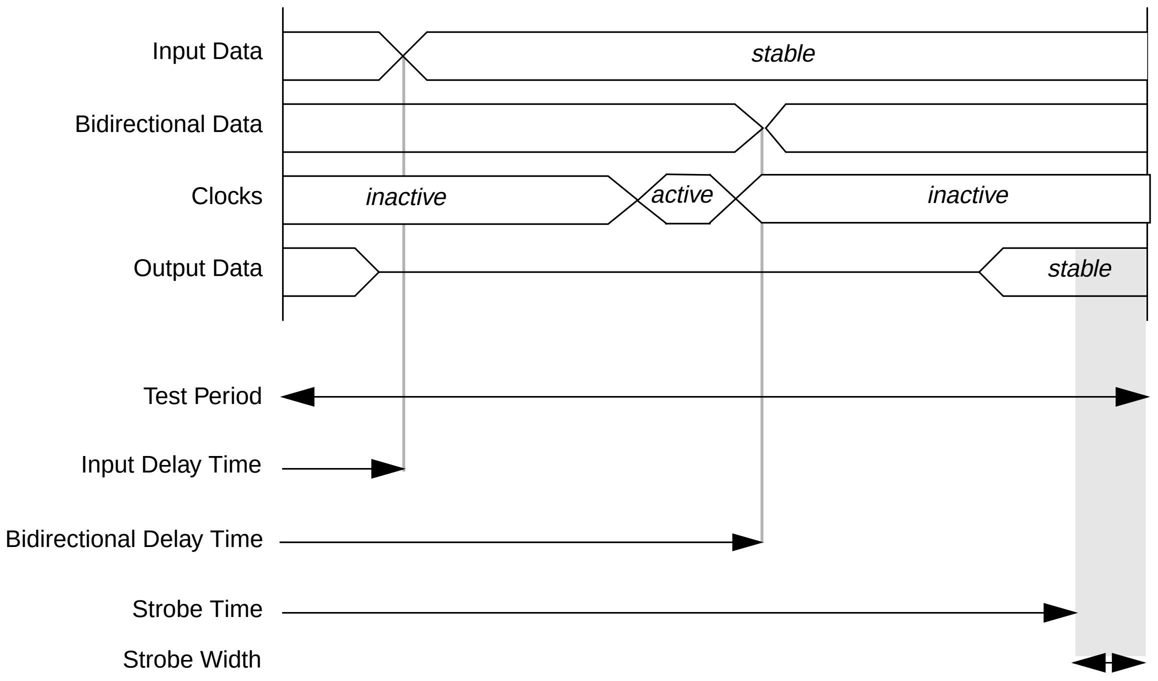
The strobe width is the width of the output strobe. A width of zero indicates that the strobe ends at the end of the period or at the first input event following the strobe time, whichever occurs first.

Clock waveform

The clock waveform defines the time, relative to the start of the tester cycle, at which the clock edges occur.

The timing diagram in Figure 12-1 shows the test timing information elements in the context of a tester cycle.

Figure 12-1 Timing Information



The test period, input delay time, bidirectional delay time, strobe time, and strobe width are the test timing parameters. Table 12-2 shows the default values for these timing parameters. The default test clock waveform depends on the type of clock (see Table 12-3).

If your test timing requirements differ from the default test timing, specify your test timing requirements by using the `test_default` variables and the `create_test_clock` command.

Table 12-2 Default Values for Test Timing Parameters

Parameter	**Default value (in ns)**
test period	100.0
input delay time	5.0
bidirectional delay time	55.0
strobe time	95.0

Table 12-2 Default Values for Test Timing Parameters (continued)

Parameter	**Default value (in ns)**
strobe width	0.0

Table 12-3 Default Test Clock Waveforms

Clock type	**First edge (in ns)**	**Second edge (in ns)**
Edge-triggered	45.0	55.0
Master clock	30.0	40.0
Slave clock	60.0	70.0
Edge-triggered*	45.0	60.0
Master clock*	50.0	60.0
Slave clock*	40.0	70.0

*Auxiliary-clock LSSD scan style only. In this scan style, the system clock is not used, the edge-triggered test clock is used for capture, and the master (a) scan clock and slave (b) scan clock are used for scan shift.

Note:

The polarity (rise or fall) of the first edge is determined from the technology library timing description for the sequential cells. DC Expert *Plus* selects the polarity of the first edge so that the majority of the cells are triggered off the first edge. The polarity of the second edge is determined by the polarity of the first edge. For example, if the first edge is rising, the second edge is falling.

Specifying the Test Period

Use the `test_default_period` variable to specify the test period (in nanoseconds). The period value must be a positive real number.

```
dc_shell> test_default_period = period
```

Specifying the Input Delay Time

Use the `test_default_delay` variable to specify the input delay time (in nanoseconds). The input delay value must be a positive real number less than the strobe value.

```
dc_shell> test_default_delay = delay
```

Specifying the Bidirectional Delay Value

Use the `test_default_bidir_delay` variable to specify the bidirectional delay time (in nanoseconds). The bidirectional delay value must be a positive real number less than the strobe value and greater than or equal to the input delay value.

```
dc_shell> test_default_bidir_delay = delay
```

Specifying the Strobe Time

Use the `test_default_strobe` variable to specify the strobe time (in nanoseconds). The strobe value must be a positive real number less than the period value and greater than the delay values.

```
dc_shell> test_default_strobe = strobe
```

Specifying the Strobe Width

Use the `test_default_strobe_width` variable to specify the strobe width (in nanoseconds). The strobe width value must be a positive real number. The strobe value plus the strobe width value must be less than or equal to the period value. A strobe width of zero indicates that the strobe width is equal to the difference between the strobe time and the first of the following events: the end of the period or the first input event after the strobe time.

```
dc_shell> test_default_strobe_width = width
```

Specifying the Test Clock Waveform

Use the `create_test_clock` command to specify the required test clock waveforms. The syntax of the `create_test_clock` command is

```
create_test_clock clock_port_list
     -waveform {rise, fall}
     [-period period_value]
```

clock_port_list

Specifies the list of clock ports that have the specified timing characteristics. If the list contains more than one port, the list must be enclosed in curly braces ({}).

Note:

DC Expert *Plus* does not support the use of bidirectional ports as clock pins.

`-waveform` {*rise, fall*}

Specifies the clock waveform. The rise argument specifies the time, relative to the start of the tester cycle, of the rising edge of the clock. The fall argument specifies the time, relative to the start of the tester cycle, of the falling edge of the clock. Both the rise and fall arguments must be positive real numbers. If you specify a value for either argument that is greater than the period value, DC Expert *Plus* calculates the value as (argument_value mod period_value).

To define a positive pulse (return-to-zero) clock, specify a larger value for the fall argument than the rise argument. To define a negative pulse (return-to-one) clock, specify a larger value for the rise argument than the fall argument.

`-period` *period_value*

Specifies the test clock period. The *`period_value`* argument must be a positive real number. The *`period_value`* argument must have the same value as the `test_default_period` variable.

The `create_test_clock` command sets the following timing attributes on the clock ports you specify:

- `test_clock_period`
- `test_clock_rise_time`
- `test_clock_fall_time`

To verify the values of these timing attributes for all clock ports, use the `report_test -port` or `report_test -timing` command.

For more information about specifying the test timing, see the *Scan Synthesis User Guide*.

Understanding the Test-Design Rule Checking Process

The `check_test` command performs test-design rule checking in distinct phases:

- Modeling checks
- Topological checks
- Protocol inference
- Protocol simulation

As `check_test` checks test-design rules, it displays information about where it is in the process and generates warnings about design rule violations. Cells corresponding to design rule violations are immediately marked as violated and modeled internally as black box cells. Black boxes inject unknown values into the simulation. Unknown values can trigger other violations. Two or more warnings issued in apparently different contexts can indicate multiple manifestations of a unique underlying problem. In general, a good strategy is to correlate various warnings, paying particular attention to the earlier ones.

At the end of design rule checking, `check_test` displays summary information about your design, including

- Net tracing results (optional)
- test-design rule violations summary
- Sequential cells summary

The *Scan Synthesis User Guide* provides information about understanding and correcting violations reported during test-design rule checking.

In addition to checking the test-design rules, the `check_test` command performs the following preprocessing functions:

- Identification of scannable cells for scan insertion
- Generation of the model used for Test Compiler automatic test-pattern generation

In most cases, scan insertion (the `insert_scan` command) and Test Compiler ATPG (the `create_test_patterns` command) use the information generated by the last `check_test` run. If you have modified the design or the test protocol since the last `check_test` run, you must rerun test-design rule checking to update the information. If you do not explicitly rerun test-design rule checking, the `insert_scan` or `create_test_patterns` command reruns the test-design rule checks before processing your design.

Modeling Checks

Modeling checks identify barriers to scan replacement and identify structures that can reduce the fault coverage results. Specifically, modeling checks identify the following modeling issues in your design:

- Black box cells
- Unsupported sequential cells
- Generic cells
- Latches using the transparent latch model
- Sequential cells with `dont_touch` attributes
- Sequential cells with `scan_element` false attributes

In addition, the modeling checks verify that the target library contains scan equivalent cells for all sequential cells in your design.

Topological Checks

Topological checks identify structures that can reduce the fault coverage results. Specifically, topological checks identify the following structures in your design:

- Invalid wired nets
- Combinational feedback loops

Protocol Inference

DC Expert *Plus* uses the test configuration and test timing you specify to infer the test protocol for the current design. In the process of inferring the test protocol, DC Expert *Plus* validates

- The clock networks
- The asynchronous pin networks

Protocol Simulation

Protocol simulation validates the proper scan operation of the current design by checking

- The ability to load (and unload) scan data during scan shift
- The ability to capture data during the parallel capture cycle

Protocol simulation validates scan operation by focusing on the behavior of the scan circuitry rather than on possibly restrictive topological rules.

The focus on circuit behavior results in the following benefits:

- Generalization of the scan element concept to include any sequential element that can be controlled and observed through a test protocol
- Ability to use the functional modes of a design to load and unload test data from scan registers

Debugging the Test Protocol

Some test-design rule violations can result from an incorrect or incomplete test protocol. DC Expert *Plus* provides an ability to analyze and debug the test protocol. This section describes the commands for debugging the test protocol. See Appendix F, “Test Protocol File Syntax,” for information about test protocol file syntax. See the *Scan Synthesis User Guide* for information about debugging techniques.

Viewing the Test Protocol

After running test-design rule checking, you can use the `write_test_protocol` command to generate an ASCII version of the inferred test protocol. You can view and edit the ASCII test protocol file.

The syntax of the `write_test_protocol` command is

```
write_test_protocol [-out test_protocol_file_name]
```

`-out` *test_protocol_file_name*

The `test_protocol_file_name` argument is the name of the ASCII output file. The default file name is design_name.tpf, where design_name is the current design and the.tpf extension identifies the file type as a test protocol file.

Tracing Nets During Protocol Simulation

DC Expert *Plus* provides a net tracing capability to enable you to see the effects of protocol simulation during test-design rule checking. Use the `trace_nets` command to specify nets for tracing during protocol simulation.

The syntax of the `trace_nets` command is

```
trace_nets {hierarchical_net_list}
```

{hierarchical_net_list}

Identifies a list of hierarchical net names you want to trace. Use a comma or space to separate net names.

You can use asterisks (*) as wildcards within a net name in the hierarchical net list. If the command parser encounters an asterisk within the list, DC Expert *Plus* traces all nets containing the specified partial net name. For example, to trace all the nets in a given level of hierarchy, specify level_of_hierarchy*. To trace all nets containing the name fragment foo, specify *foo*.

Note:

Use wildcards carefully. For example, the following command directs Test Compiler to trace every net in the current level of hierarchy, which creates a difficult debugging situation.

```
dc_shell> trace_nets{*}
```

To disable tracing on nets previously selected with the `trace_nets` command, use the `untrace_nets` command.

The syntax of the `untrace_nets` command is

```
untrace_nets {hierarchical_net_list} | -all
```

{hierarchical_net_list}

Specifies the hierarchical nets you want to untrace. Use commas or spaces to separate net names. Use asterisks for a wildcard net search, as described in the `trace_nets` command syntax.

`-all`

Disables net tracing on all nets in the current design that have net tracing enabled by the `trace_nets` command.

Generating Fault Coverage Results

The sequential cell summary provides an indicator of the testability of the design. Use fault coverage information to

- Get a quantitative assessment of the achievable fault coverage in a module.
- Evaluate the fault coverage impact of structures, such as transparent latches and internal three-state nets.

Use the `create_test_patterns -dft` command to run Test Compiler ATPG. The `create_test_patterns -dft` command generates fault coverage information but does not save the test patterns.

The syntax for the `create_test_patterns -dft` command is

```
create_test_patterns -dft
    [-backtrack_effort low | medium | high]
    [-check_contention true | false]
    [-check_float true | false]
    [-max_cpu_per_fault CPU_time_per_fault]
    [-max_total_cpu total_CPU_time]
    [-max_random_patterns pattern_maximum]
    [-random_pattern_failure_limit failure_limit]
    [-sample percent] [-background run_name
    [-host machine_name]
    [-arch architecture] [-xterm]
```

`-backtrack_effort low | medium | high`

Controls the amount of effort expended in the ATPG process. Backtracking occurs when Test Compiler ATPG attempts to generate a test pattern, determines that the pattern is unsuitable, then generates another pattern. For example, if a required condition at one internal node conflicts with a required condition on another node, Test Compiler ATPG cannot generate a pattern that satisfies both conditions. Consequently, Test Compiler ATPG backtracks to determine an alternate pattern to cover the fault. This option applies to the deterministic phase of pattern generation. Refer to the “Deterministic-Pattern Generation” on page 12-23, for more information.

Valid values for the `backtrack_effort` argument are low, medium, or high. By default, the backtrack effort level is low.

`-check_contention true | false`

Controls three-state bus contention checking. When set to true, Test Compiler ATPG checks for three-state bus contention. The default is true.

`-check_float true | false`

Controls three-state bus float checking. When set to true, Test Compiler ATPG checks for three-state bus float. The default is true.

`-max_cpu_per_fault` *`CPU_time_per_fault`*

Controls the maximum amount of CPU time Test Compiler ATPG takes to generate a pattern to detect a single fault. Specify the time in seconds (there is no maximum time limit). If Test Compiler ATPG cannot generate a pattern for a particular fault within the time you specify, that fault is classified as abandoned. This option applies to the deterministic phase of pattern generation. Refer to the “Deterministic-Pattern Generation” on page 12-23, for more information.

`-max_total_cpu` *total_CPU_time*

Controls the maximum total amount of CPU time Test Compiler ATPG uses when generating test patterns. Specify the time in seconds (there is no maximum time limit). The amount of time you specify varies with the type of computer you use. Faults that are not processed within the time you define are classified as abandoned. If the specified time limit is exceeded, the `create_test_patterns -dft` command issues the following message:

```
Test Vector Generation Interrupted Due to Exceeding
Maximum CPU.
```

`-max_random_patterns` *pattern_maximum*

Controls the maximum number of random patterns generated during the random-pattern generation phase. Test Compiler ATPG generates random patterns in parallel sets of 32, so choose a multiple of 32 for the argument. By default, there is no limit. Refer to “Random-Pattern Generation” on page 12-23 for more information.

`-random_pattern_failure_limit` *failure_limit*

Controls the point where Test Compiler ATPG switches from random-pattern generation to deterministic-pattern generation. A set of 32 random patterns fails if it does not increase the fault coverage for a design by at least 0.5 percent. Because the default limit is 64, Test Compiler ATPG switches to deterministic-pattern generation when two consecutive sets of 32 random patterns fail to increase the fault coverage by at least 0.5 percent. Refer to “Understanding the ATPG Process” on page 12-23, for more information.

`-sample` *`percent`*

Controls the percentage of faults targeted in the Test Compiler ATPG run. You can speed up the Test Compiler ATPG run by estimating the fault coverage for a design using a random sampling of the fault list. Use the percent argument to specify the percentage of faults to be randomly selected for Test Compiler ATPG.

`-background` *`run_name`*

Moves the Test Compiler ATPG job to the background. You must specify the absolute or relative path name to the directory where the ATPG results will be stored. This directory cannot be an existing directory; it is created by the `-background` option. The following three options (`-host`, `-arch`, and `-xterm`) work with the `-background` option.

`-host` *`machine_name`*

> Specifies the machine where DC Expert *Plus* runs the background job. The `machine_name` argument must be a valid network machine name, and DC Expert *Plus* must be licensed to run on it. The default is the local host. This option is valid only with the `-background` option.

`-arch` *`architecture`*

> Specifies the architecture of the machine (such as sparc, solaris, rs6000, or hp700) where the background job is run. This option is valid only with the `-background` option.

`-xterm`

> Enables the results of the background job to be monitored in an xterm. This option is valid only with the `-background` option in the X11 environment.

The fault coverage of a module might decrease when you run ATPG at the chip level, because the controllability and observability of the module ports change after the module is embedded within the design hierarchy.

When you run Test Compiler ATPG on a nonscan or unrouted scan module, the tool assumes that the scan path includes all sequential cells reported in the sequential cell summary as

- Being valid scan cells
- Having scan shift violations
- Having parallel capture violations

Understanding the ATPG Process

The Test Compiler ATPG process consists of the following phases:

- Random generation of patterns to efficiently cover easy-to-detect faults
- Deterministic generation of patterns to cover particular stuck-at faults that are otherwise hard to detect

The following sections describe each of these phases:

Random-Pattern Generation

During the first phase, Test Compiler ATPG creates random test patterns and monitors the overall fault coverage achieved with each new pattern. Test Compiler fault-simulates each random pattern to determine the faults it detects.

In most designs, Test Compiler ATPG can easily detect many faults by using this method, which is faster than deterministic-pattern generation. When Test Compiler ATPG reaches the point where additional random patterns do not significantly increase the fault coverage, Test Compiler ATPG continues to the second phase.

Deterministic-Pattern Generation

In the second phase, Test Compiler ATPG analyzes the remaining undetected faults one at a time and tries to generate patterns to detect each of those faults. Whenever Test Compiler ATPG generates a deterministic test pattern, the tool fault simulates the pattern, to determine if the pattern detects other previously undetected faults. If the fault simulation identifies additional detected faults, Test Compiler ATPG classifies these faults as detected.

Understanding the Fault Coverage Information

After processing the design, Test Compiler ATPG generates a summary report of the fault coverage results. Example 12-1 shows the summary report.

Example 12-1 Output of the create_test_patterns -dft Command

```
...
                                Non-collapsed     Collapsed
    No. of detected faults      388               245
    No. of abandoned faults     0                 0
    No. of tied faults          0                 0
    No. of redundant faults     0                 0
    No. of untested faults      106               68
    Total no. of faults         494               313
    Fault coverage              78.54             78.27
...
```

The summary report provides the fault coverage for the design and the number of faults in each fault class. Test Compiler ATPG uses the following fault classes:

detected

Test Compiler ATPG generated a pattern that detects this fault.

abandoned

Test Compiler ATPG reached the CPU limit that you set before the tool could classify this fault. Even with additional computer resources, Test Compiler ATPG might not succeed in classifying this fault.

tied (high or low)

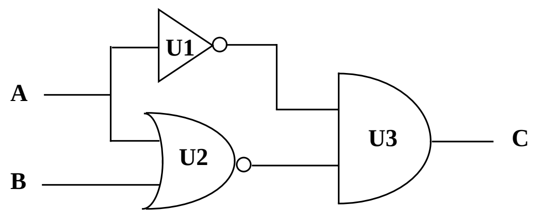
Test Compiler ATPG cannot generate a test for this fault, because the node is connected to logic 1 (tied high) or logic 0 (tied low). Test Compiler ATPG also classifies nodes controlled by the tied node as tied faults.

redundant

Test Compiler ATPG cannot generate a test for this fault, because the overall static behavior of the circuit is independent of the values at this node.

The design in Figure 12-2 contains redundant logic. The value of the output port C is independent of the value of the output of cell U1.

Figure 12-2 Redundant Logic Example



Because of the redundant logic, Test Compiler ATPG cannot generate a pattern to detect the stuck-at-1 fault at the output of cell U1. To generate a pattern to test for this fault, Test Compiler ATPG must

- Set the output of cell U1 to the opposite value (logic 0) by applying a logic 1 to input A.
- Set the output of U2 to a noncontrolling value of logic 1 by applying a logic 0 to both input A and input B.

Due to these conflicting requirements, Test Compiler ATPG cannot generate a pattern to detect the stuck-at-1 fault.

untested

Test Compiler ATPG cannot generate a test for this fault because the tool cannot control or observe the fault.

Test Compiler ATPG calculates the fault coverage as

$$\frac{\texttt{number of detected faults}}{\texttt{(total number of faults)} - \texttt{(number of undetectable faults)}}$$

The undetectable faults include abandoned, tied, redundant, and untested faults.

After running Test Compiler ATPG, you can use the `report_test -faults` or `report_test -coverage` command to obtain fault status information.

Reporting Test Information

Use the `report_test` command to display test-related information about the current design. To select the type of information to be printed, specify one or more `report_test` options. Some of the options have arguments that further control the report content. You can select as many different reports as desired, but you cannot select the same report more than once.

The syntax of the `report_test` command is

```
report_test
     [-assertions] [-atpg_conflicts]
     [-configuration]
     [-coverage [-incremental] [-inst instance_list]]
     [-faults [-class fault_class] [-inst instance_list]
     [-methodology] [-port]
     [-scan_path] [-trace_nets] [-nosplit]
```

`-assertions`

Selects the assertions report. This report contains a list of the ports in the current design that have `test_hold` attributes.

`-atpg_conflicts`

Selects the ATPG conflicts report. This report contains a list of the conflicts detected during ATPG. These conflicts can result from bus contention, float conditions, or conflicting logical conditions in the test configuration.

`-configuration`

Selects the scan configuration report. This report contains a list of the scan configuration settings for the current design, as specified by the `set_scan_configuration` command. The scan configuration for the design includes, among other things, information about the test methodology, the scan style for the design, the number of scan chains desired, and the type of clock mixing.

`-coverage`

Selects the fault coverage report. This report contains the fault coverage for the current design and all its subdesigns. In addition to the fault coverage percentage, the report gives the number of faults in each fault class. See the “Understanding the Fault Coverage Information” on page 12-24 for information about fault classes. You can restrict the fault coverage report to specific design instances, using the `-inst` option.

`-inst` *`instance_list`*

Specifies the design instances that DC Expert *Plus* includes in the fault report or the coverage report.

`-faults`

Selects the fault report. This report lists all pins and ports in the current design's hierarchy that have the fault class specified in the `-class` option. If you do not specify the `-class` option, the report lists all fault classes. You can restrict the fault report to specific design instances, using the `-inst` option.

`-class` *`fault_class`*

Specifies the fault classes to include in the fault report. Valid values for the `fault_class` argument are abandoned, detected, redundant, tied, and untested.

`-methodology`

Selects the scan methodology report. This report contains details of the scan style and the test methodology for the current design.

`-port`

Selects the test port report. This report contains details about the test ports for the current design and displays (if applicable) signal type, scan chain index, and clock attributes.

`-scan_path`

Selects the scan path report. This report contains a list of every scan cell on the scan path for the current design.

`-trace_nets`

Selects the trace nets report. This report contains a list of the nets in the current design that are enabled for tracing during test-design rule checking.

`-nosplit`

Prevents line splitting and facilitates writing software to extract information from the report output. DC Expert *Plus* lists most design information in fixed-width columns. If the information for a given field exceeds its column’s width, the next field begins on a new line, starting in the correct column.

13

Timing Analysis in Latches

A latch is a simple, 1-bit level-sensitive memory device. In simulation, a signal holds its value until the output is reassigned. In hardware, a latch implements this holding-of-state capability.

This chapter explains timing in latches and how time borrowing occurs and can be used to balance the slack times in latch-based designs to optimize near-critical paths and reduce delay costs. It includes an example of a latch-based design that utilizes time borrowing.

This chapter includes the following sections:

- Time Borrowing in Latch-Based Designs
- Constraining a Latch-Based Design
- Latch-Based Time-Borrowing Example: Linear Block Encoder and Decoder

Time Borrowing in Latch-Based Designs

Design Compiler applies special timing, called time borrowing, to designs containing level-sensitive latches. A latch is transparent for the duration of the active clock pulse, meaning that the data appearing on the input of the latch is propagated directly to the output of the latch. If the path leading to the data pin of a latch is too long, one cycle can borrow from the next cycle to extend the time during which the latch is enabled.

A Simple D Latch

To illustrate the processes involved in time borrowing, this section uses a simple D latch. This section includes example VHDL and Verilog code for the D latch and shows the inference report generated when either set of code is compiled.

When you infer a D latch, make sure you can control the gate and data signals from the top-level design ports or through combinational logic. Controllable gate and data signals ensure that simulation can initialize the design.

Example 13-1 gives the VHDL code for a D latch.

Example 13-1 VHDL Template for a D Latch

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

entity d_latch is
    port (GATE, DATA: in std_logic;
         Q : out std_logic );
end d_latch;

architecture rtl of d_latch is
begin
infer: process (GATE, DATA) begin
    if (GATE = ’1’) then
        Q <= DATA;
    end process infer;

end rtl;
```

Example 13-2 gives the Verilog code for a D latch.

Example 13-2 Verilog Template for a D Latch

```
module d_latch (GATE, DATA, Q);
    input GATE, DATA;
    output Q;
    reg Q;

always @(GATE or DATA)
    if (GATE)
        Q = DATA:

end module
```

An inference report contains the information the code compiler passes on to the Design Compiler about the inferred devices.

Table shows the verbose inference report generated for a D latch, using either the VHDL code in Example 13-1 or the Verilog code in Example 13-2. Although the coding styles differ, both inferences are the same, and thus they produce the same inference report.

Table 13-1   Inference Report for a D Latch

Register name	**Type**	**Width**	**Bus**	**MB**	**AR**	**AS**	**SR**	**SS**	**ST**
Q_reg	Latch	1	–	–	N	N	–	–	–

```
Q_reg
-----
    reset/set: none
```

For details to help you understand the inference report for Verilog code, see the chapter “Register, Mulitbit, Multiplexer, and Three-State Inference” in the *HDL Compiler for Verilog Reference Manual*.

For details to help you understand the inference report for VHDL code, see the chapter “Register, Multibit, Multiplexer, and Three-State Inference” in the *VHDL Compiler Reference Manual*.

About Time Borrowing

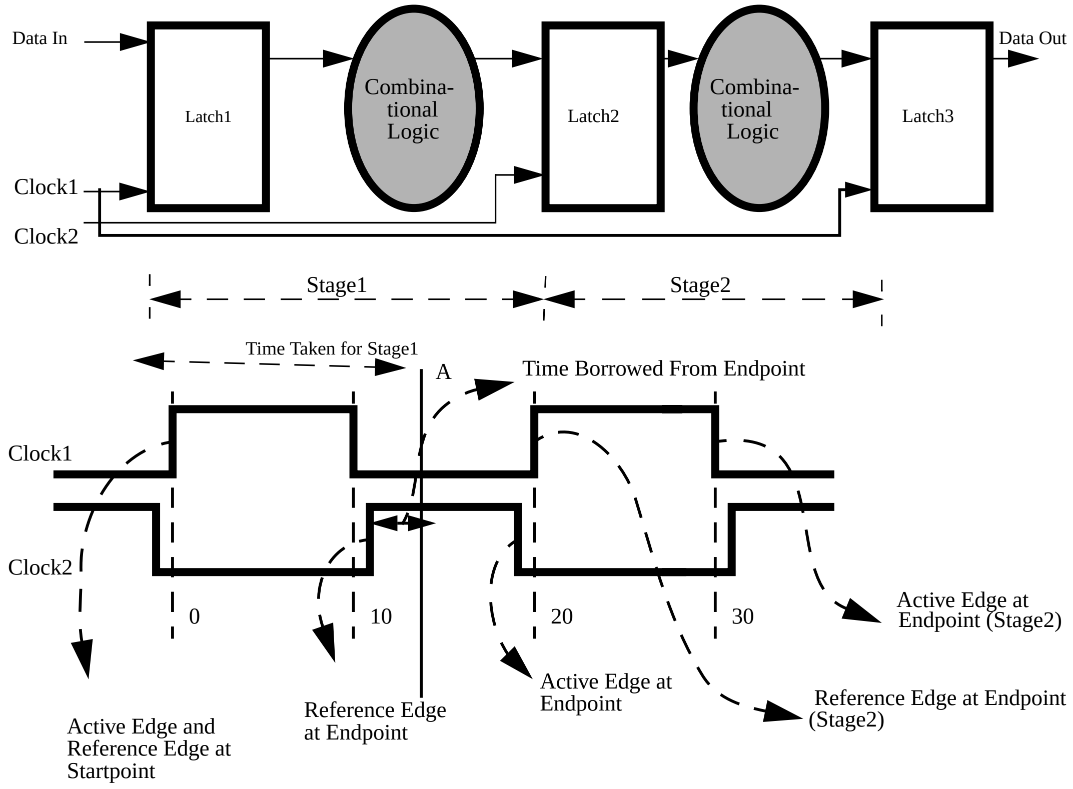
If the path leading to the data pin of a latch is too long, time can occasionally be borrowed from the next cycle to enable the latch for a certain duration. This section uses the two-stage latch-based design shown in Figure 13-1 to explain the concepts that underlie the time-borrowing process.

Each stage of the design consists of a block of combinational logic with a latch on either side. The startpoint of a stage is defined by the latch that precedes the combinational logic block; the endpoint of a stage is defined by the latch that follows the combinational logic block. Thus, as indicated in the bottom illustration of Figure 13-1, the endpoint of one stage is also the startpoint of the next stage.

In the bottom illustration of Figure 13-1, the dotted line labeled “Active Edge...at Startpoint” denotes the edge of the clock that renders the startpoint latch transparent—Latch1 in the top illustration of the same figure is the startpoint latch.

The dotted line labeled “Active Edge at Endpoint” denotes the edge of the clock before which the data should stabilize at the input of the endpoint latch—Latch2 in the top illustration of the figure is the endpoint latch.

Figure 13-1 General Structure of a Latch-Based Design


Data In
Latch1
Combina-tional Logic
Latch2
Combina-tional Logic
Latch3
Data Out
Clock1
Clock2
Stage1
Stage2
Time Taken for Stage1
A
Time Borrowed From Endpoint
Clock1
Clock2
0
10
20
30
Active Edge at Endpoint (Stage2)
Active Edge at Endpoint
Reference Edge at Endpoint (Stage2)
Reference Edge at Endpoint
Active Edge and Reference Edge at Startpoint

The reference edge at startpoint—marked by the same dotted line that shows the active edge at startpoint—is the edge of the clock that serves as the reference base for making timing calculations for the startpoint latch, Latch1. The reference edge at endpoint is the edge of the clock that serves as the reference base for making timing calculations for the endpoint latch, Latch2.

In this design, the combinational logic block between Latch1 and Latch2 has more delay than the delay between Latch2 and Latch3. To resolve this discrepancy, the first stage can borrow time from the second clock cycle. In this event, the second clock cycle is left with less time to accommodate the combinational logic block between Latch2 and Latch3.

Depending on the requirement, the combinational block between Latch1 and Latch2 can utilize the time period from the reference edge at startpoint (Latch1) to the line labeled AB; this is the borrowed time Stage1 gets. Initially, this is 50 units of the time used (whether milliseconds, nanoseconds, or another unit of time). Giving Stage1 the borrowed time results in the next combinational block—that is, Stage2— having the time period from AB onward. Thus, the starting point for the Stage2 combinational block becomes AB.

Balancing Relative Slacks

This section describes how to balance the relative slacks in two sides of a latch in order to cause near-critical paths to be optimized and reduce costs.

This section includes the following topics:

- Determining Relative Slack
- Optimizing Near-Critical Paths
- Reducing Delay Costs

Determining Relative Slack

The relative slack of a path is the absolute (negative) slack of the path group to which the path in question belongs, minus the (negative) slack of the critical path in the same path group. Unless the path in question is the critical path of the path group, the relative slack will always be a negative number. If the path in question is the path group's critical path, then the relative slack will be zero (0). The equation in Example 13-3 shows how relative slack is determined.

Example 13-3   Equation Used to Ascertain Relative Slack

```
relative_slack =
(absolute slack of the path group) – (slack of the critical path of the path group)
```

Optimizing Near-Critical Paths

This section explains the relative slacks of both stages of the example latch-based design, based on the initial amount of time borrowed by Stage1 from the Clock2 cycle—that is, before the relative slacks are balanced. Then it explains how to balance these relative slacks.

Relative Slacks of Both Stages Before Balancing

In considering the latch-based design example for whose two stages the relative slacks are shown in Figure 13-2 before balancing, assume the following values to be true:

- The time borrowed from the endpoint in Stage1 is 50.
- The critical path’s slack in the path group for Clock1 is –70.
- The critical path’s slack in the path group for Clock2 is –40.

In this scenario, neither Stage1 nor Stage2 is the critical path.

Given the equation in Example 13-3 that is used to determine relative slack and the values used in the example latch-based design, here is how the relative slacks resolve for both stages:

- Relative slack in Stage1 is

  ```
  0  –  –40 = 40
  ```

  where

  - 0 is the absolute slack

- –40 is the critical path’s slack
- 40 is the relative slack

- Relative slack in Stage2 is

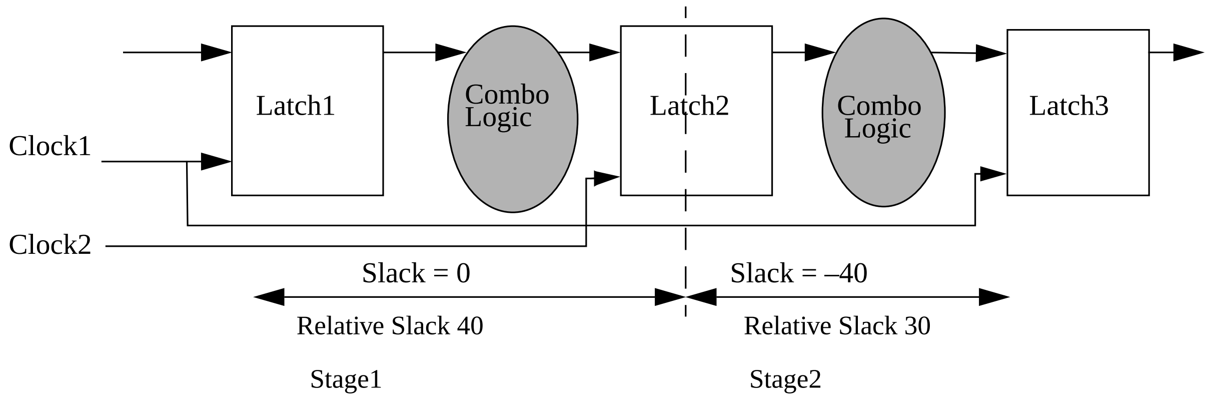
  ```
  –40  –  –70 = 30
  ```

  where

  - –40 is the absolute slack
  - –70 is the critical path’s slack
  - 30 is the relative slack

The Stage1 relative slack, which is 40, and the Stage2 relative slack, which is 30, are not balanced. To achieve an absolute slack of 0 for Stage1, the absolute slack for Stage2 must be increased—that is, worsened. Design Compiler will not attempt to optimize the Stage1 path even if a critical range is set. Figure 13-2 shows both stages before the relative slack is balanced.

Figure 13-2 Both Stages Before Relative Slack Balancing (for Optimizing Near-Critical Paths)



Relative Slacks After Balancing

To attempt to balance the relative slacks, the time borrowed from the Clock2 cycle and given to Stage1 is reduced to 45. This produces an absolute slack of –35 for Stage2 and an absolute slack of –5 for Stage1.

Given the equation shown in Example 13-3, used to ascertain relative slack, and the new absolute slack values, here is how the relative slacks now resolve for both stages:

- Relative slack in Stage1

  ```
  –5  –  –40 = 35
  ```

  where

  - 5 is the absolute slack
  - –40 is the critical path’s slack
  - 35 is the relative slack

- Relative slack in Stage2 is

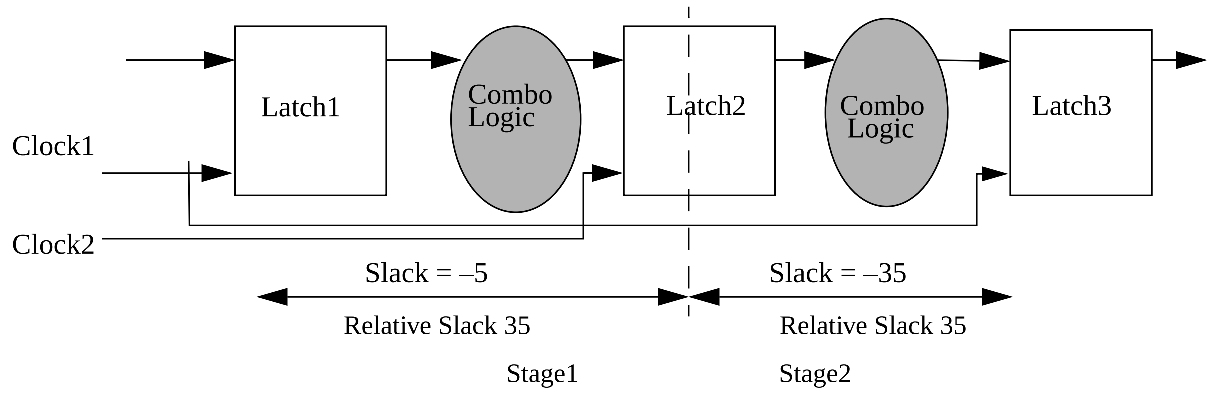
  ```
  –35  –  –70 = 35
  ```

  where

  - –35 is the absolute slack
  - –70 is the critical path’s slack
  - 35 is the relative slack

This modification of the amount of time borrowed results in Stage1 and Stage2 relative slacks that are balanced. Design Compiler performs this resolution automatically. Figure 13-3 depicts the result of balancing the relative slacks.

Figure 13-3 Both Stages After Balancing Relative Slacks (for Optimizing Near-Critical Paths)



Given that Stage1 now has a negative slack—that is, –5—if you set a critical range that is more than 5, Design Compiler will optimize the path. Therefore, balancing relative slacks helps in optimizing near-critical paths.

Reducing Delay Costs

This section explains how to reduce delay costs by balancing the relative slacks of both stages of a latch-based design.

Relative Slacks Before Balancing

In considering the latch-based design example for whose two stages the relative slacks are shown in Figure 13-2 before they are balanced, assume the following values to be true:

- The time borrowed from the endpoint in Stage1 is 60.

- The critical path in the path group for Clock1 has a slack of –70.
- The critical path in the path group for Clock2 has a slack of –40.

Assume, also, that the slacks in some of the near-critical paths in the Clock1 path group are –60, –50. and –40.

Here is how the relative slacks resolve for both stages:

- Relative slack in Stage1

  ```
  –0  –  –40 = 40
  ```

  where

  - 0 is the absolute slack in Stage1
  - –40 is the critical path’s slack in path group Clock2
  - 40 is the relative slack

- Relative slack in Stage2 is

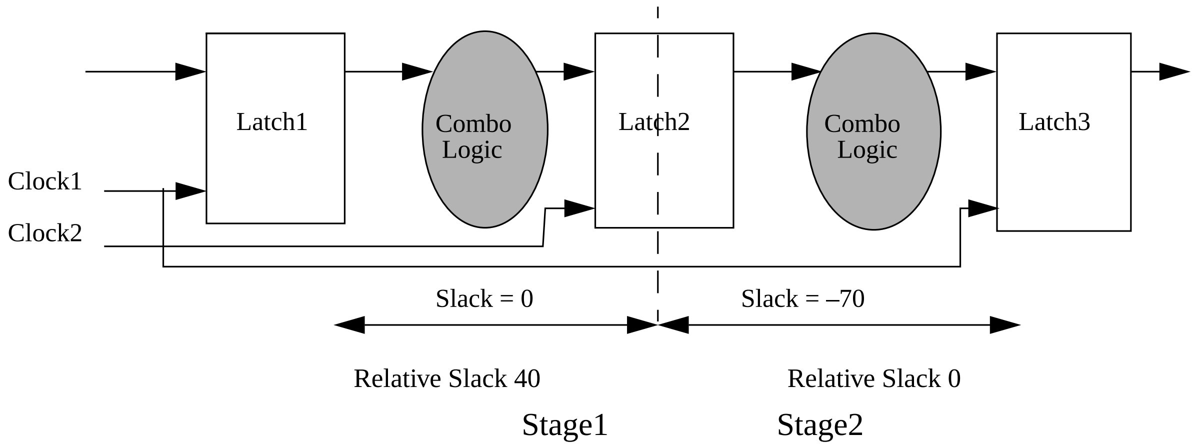
  ```
  –70  –  –70 = 0
  ```

  where

  - –70 is the absolute slack in Stage2
  - –70 is the critical path’s slack in path group Clock2
  - 0 is the relative slack

As shown in Figure 13-4, the relative stack in Stage1 is –40 and the relative slack in Stage2 is 0. This results in Stage2 being the critical path of its path group. The resulting delay cost is 110, given a critical path slack in path group Clock1 of 70 and a critical path slack in path group Clock2 of 40.

Figure 13-4 Both Stages Before Balancing Relative Slacks (for Reducing Delay Costs)



Relative Slacks After Balancing

In an attempt to balance the relative slacks of both stages, the time borrowed by Stage1 from the Clock2 cycle is reduced from 60 to 35. Ultimately, this reduces the delay costs from 110 to 100, as shown in Figure 13-5.

Reducing the time borrowed by Stage1 produces the following balanced relative slacks for both stages:

- Relative slack in Stage1

```
–25  –  –40 = 15
```

where

- 25 is the absolute slack in Stage1
- –40 is the critical path’s slack
- 15 is the relative slack

- Relative slack in Stage2 is

  ```
  –45  –  –70 = 0
  ```

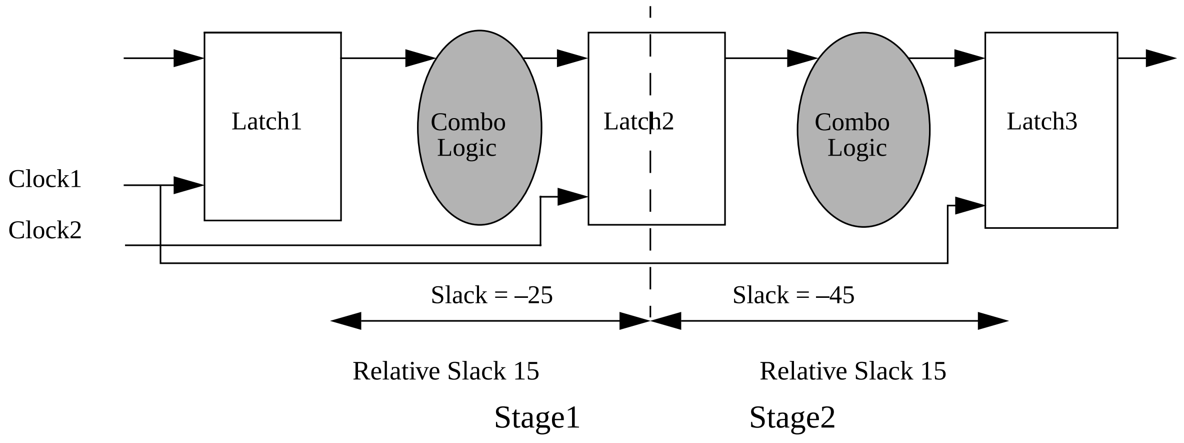
  where

  - –45 is the absolute slack in Stage2
  - –60 is the critical path’s slack
  - 15 is the relative slack

The critical slack in path group Clock1 is now 60, and the critical path’s slack in the path group of Clock2 is now 40.

The relative slack of both stages is now 15, as shown in Figure 13-5.

Figure 13-5   Result of Balancing Relative Slacks for Reducing Delay Costs



As is evident from Figure 13-5, if the relative slacks had not been balanced, the path in Stage2 would have been treated as the critical path and the near-critical path with a slack of –60 would not have been optimized. Balancing the relative slacks causes the previous near-critical path to become the critical path, subjecting it to optimization, which is desirable. Moreover, you can easily optimize the previous critical path by setting the proper critical range.

Constraining a Latch-Based Design

Design Compiler assumes that all external registers are positive edge flip-flops unless they are explicitly created as level-sensitive latches.

You use the following commands to infer external registers:

- set_input_delay *delay_value* -clock *clock_name input_port*
- set_output_delay delay –clock *clock_name output_port*

These commands set the input delay on input and output ports relative to a clock cycle. The *delay_value* argument specifies the path delay, expressed in units of time consistent with the technology library used during optimization. The *clock_name* argument specifies the clock the stipulated delay pertains to. The *input_port* argument specifies the input port in the current design to which *delay_value* is assigned. The *output_port* argument specifies the output port in the current design to which *delay_value* is assigned.

Specify the –level_sensitive switch for these commands to define external registers as active-high level-sensitive latches, as shown in the following example:

```
set_input_delay 4 –clock CLK2 –level_sensitive all_inputs()
```

Specifying the –level_sensitive switch allows the tool to derive the setup and hold relationships for paths from this port, with the presumption that it is a level-sensitive latch.

Specify both the –level_sensitive and –clock_fall switches to define external registers as active-low level-sensitive latches, as shown in the following example:

```
set_output_delay 3 –clock CLK2 –level_sensitive –clock_fall
all_outputs()
```

Specifying the –clock_fall switch stipulates that the delay is relative to the falling edge of the clock.

Creating Non-Overlapping Clocks

This section describes non-overlapping clocks, which are used by most two-phase designs. It also explains two-phase designs.

You use the `create_clock` command to create two non-overlapping clocks. Here are examples showing how to specify the `create_clock` command:

```
create_clock -name CLK1 -period 15 -waveform {9, 14} find(port, CLOCK1)

create_clock -name CLK2 -period 15 -waveform {2,7} find(port, CLOCK2)
```

What Are Two-Phase Designs?

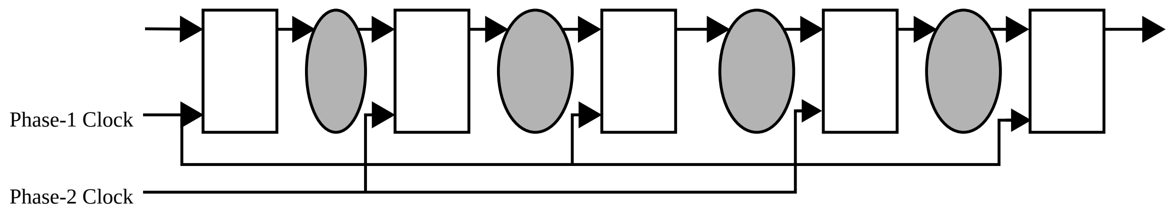
Most latch-based designs use two-phase non-overlapping clocks. These designs are referred to as two-phase designs.

The following requirements apply to two-phase designs:

- All paths originating at a phase-1 latch should terminate at a phase-2 latch. A phase-1 latch is a latch clocked by a phase-1 clock.
- All paths originating at a phase-2 latch should terminate at a phase-1 latch.

Consequently, successive stages of latches should be clocked by alternative clocks, as illustrated in Figure 13-6.

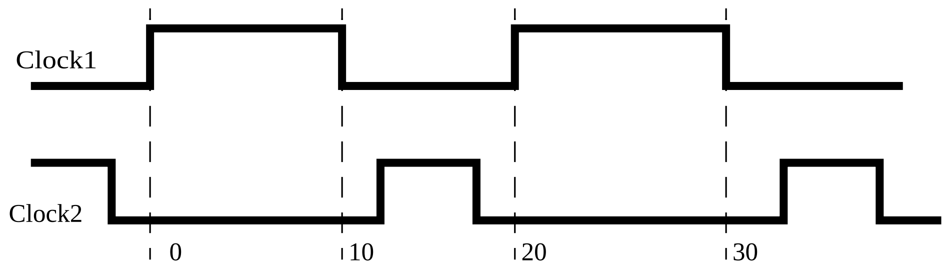
Figure 13-6 Two-Phase Design



What Are Non-Overlapping Clocks?

Non-overlapping clocks are clocks that don’t make a latch transparent simultaneously. Figure 13-7 shows two non-overlapping clocks, Clock1 and Clock2.

Figure 13-7 Non-Overlapping Clocks



Latch-Based Time-Borrowing Example: Linear Block Encoder and Decoder

This section provides an example design for a linear block encoder. The design is composed of the following parts:

- Linear Block Encoder
- Noisy Channel
- Linear Block Decoder for Single Bit Error

The linear block encoder and decoder design, presented in this section, is a latch-based design that uses time borrowing. This section describes the design before giving the code that implements it.

This section includes these topics:

- About the Linear Block Encoder and Decoder
- About the Noisy Channel
- About the Linear Block Decoder for Single-Bit Error
- Linear Block Encoder and Decoder Implementation
- Setting Constraints on the Linear Block Encoder

About the Linear Block Encoder and Decoder

Example 13-9 gives the VHDL code that implements the linear block encoder and decoder. Before you review the code, read this background information about the encoder and decoder implementation.

Here is how the linear block encoder and decoder design works: Each 4-bit message word, denoted as a row vector or 4-tuple

D = (d1,d2,d3,d4)

is transformed to a code word C, 7 bits in length

```
(C= (c1,c2,...c7))
```

To achieve this, the linear block encoder adds 3 parity bits to each 4-bit message.

The value of C is generated from the matrix multiplication equation in Example 13-4.

Example 13-4   Equation i: Matrix Multiplication

```
C = DG      ...(i)
```

where G is the generator matrix of the code.

Example 13-5 shows the generator matrix equation.

Example 13-5   Equation ii: Generator Matrix

```
G = [ I4 | P ]4x7    ...(ii)
```

where

- I4 is an identity matrix of order 4.
- P is an arbitrary matrix of order 4 by 3, known as a parity matrix.

The VHDL sample code includes these processes:

- P_MATRIX_IN

This process reads the P matrix and generates the G matrix and the HT matrix. The HT matrix is a transposition of the parity check matrix.

- DATA_READ

  This process reads in the 4-bit message word.

- LINEAR_BLOCK_CODE

  This process generates the Linear Block Code (C) according to equation i, the matrix multiplication equation in Example 13-4.

About the Noisy Channel

The noisy channel adds a 1-bit error to the code word C and generates the signal R. The signal is received by the decoder in the format represented by Equation iii in Example 13-6.

Example 13-6 Equation iii: Signal R

```
R = C + E        ...(iii)
```

where E is a 7-bit word with any one of the first 4 bits equal to 1 and all other bits equal to 0. The bit that is set to 1 is the erroneous bit.

About the Linear Block Decoder for Single-Bit Error

A parity check matrix H is associated with the generator matrix G. Example 13-7 shows the parity check matrix.

Example 13-7 Equation iv: Parity Check Matrix

```
H = [PT | I3]3x7   ...(iv)
```

In the equation given in Example 13-8, where S is a 3-bit word known as the error syndrome of R for nonzero E, S is one of the rows of the matrix HT, depending on the erroneous bit of R.

Example 13-8 Equation v: Error Syndrome

```
S = RHT       ...(v)
S = 0, if E = 0;
```

That is, if there is an error in the ith bit in R, then the syndrome is the ith row of the H^T matrix. Thus, by comparing the syndrome with the rows of H^T, the error can be detected and the data word D can be recovered.

The SYNDROME_GEN process in the VHDL code shown in Example 13-9 generates the syndrome according to equation v, the error syndrome equation.

The DECODER process compares the syndrome with the rows of HT and recovers D.

Linear Block Encoder and Decoder Implementation

This section includes these parts:

- VHDL Code
- Setting Constraints on the Linear Block Encoder

VHDL Code

Example 13-9 shows the VHDL code that implements the linear block encoder and decoder described in the preceding sections.

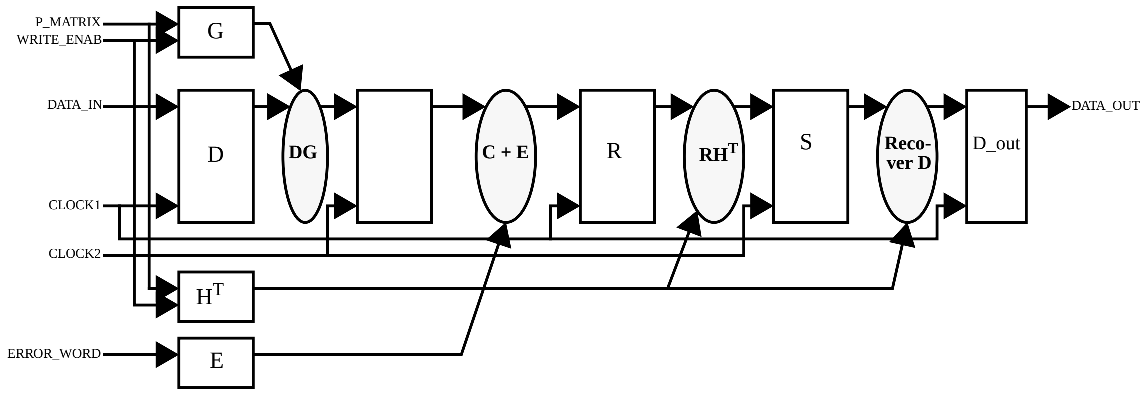
Example 13-9 VHDL Code for Linear Block Encoder and Decoder

```
library IEEE, SYNOPSYS, WORK;
use IEEE.std_logic_1164.all;
use SYNOPSYS.attributes.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_misc.all;
use WORK.MATRIX_RELATED.all;

entity LINEAR_DECODER is
  port ( DATA_IN : in DATA_WORD;
         ERROR_WORD : in CODE_WORD;
         P_MATRIX : in CHECK_MATRIX;
         CLOCK1, CLOCK2, WRITE_ENAB : in std_logic;
         DATA_OUT : out DATA_WORD );
end LINEAR_DECODER;

architecture BEHAVIORAL of LINEAR_DECODER is

signal D : DATA_WORD;
signal P : CHECK_MATRIX;
signal C, E, R, R1  : CODE_WORD;
signal S : SYNDROME_WORD;
signal G : GENERATOR_MATRIX;
signal H_tran : PARITY_CHECK_MATRIX_TRAN;

begin
  P_MATRIX_IN : process (P_MATRIX, WRITE_ENAB)
  variable G_temp : GENERATOR_MATRIX;
  variable H_tran_temp : PARITY_CHECK_MATRIX_TRAN;
  begin
    if(WRITE_ENAB = ’1’) then
      P <= P_MATRIX;
      E <= ERROR_WORD;
      for I in K downto 1 loop
        for J in K downto 1 loop
          if ( I = J ) then
            G_temp(I)(J) := ’1’;
          else G_temp(I)(J) := ’0’;
          end if;
        end loop;
        for J in N downto K+1 loop
          G_temp(I)(J) := P_MATRIX(I)(J-K);
        end loop;
      end loop;
      for I in K downto 1 loop
        for J in N-k downto 1 loop
           H_tran_temp(I)(J) := P_MATRIX(I)(J);
        end loop;
      end loop;
      for I in N downto K+1 loop
        for J in N-K downto 1 loop
          if ( I-K = J ) then
            H_tran_temp(I)(J) :=  ’1’;
          else H_tran_temp(I)(J) := ’0’;
          end if;
        end loop;
      end loop;
      G <= G_temp;
      H_tran <= H_tran_temp;
    end if;
  end process P_MATRIX_IN;

  DATA_READ : process ( DATA_IN, CLOCK1 )
  begin
    if ( CLOCK1 = ’1’ ) then
      D <= DATA_IN;
    end if;
  end process DATA_READ;

  LINEAR_BLOCK_CODE : process ( D, CLOCK2 )
  variable C_temp : CODE_WORD;
  begin
    if ( CLOCK2 = ’1’ ) then
      for I in C’range loop
        C_temp(I) := ’0’;
        for J in D’range loop
          C_temp(I) := C_temp(I) xor ( D(J) and G(J)(I) );
        end loop;
      end loop;
      C <= C_temp;
    end if;
  end process LINEAR_BLOCK_CODE;

  CHANNEL : process (C, CLOCK1)
  begin
    if ( CLOCK1 = ’1’ ) then
      for I in C’range loop
        R(I) <= C(I) xor E(I);
      end loop;
    end if;
  end process CHANNEL;

  SYNDROME_GEN : process ( R, CLOCK2 )
  variable S_temp : SYNDROME_WORD;
  begin
    if ( CLOCK2 = ’1’ ) then
      for I in S’range loop
        S_temp(I) := ’0’;
        for J in R’range loop
       S_temp(I) := S_temp(I) xor ( R(J) and H_Tran(J)(I) );
        end loop;
      end loop;
      S <= S_temp;
      R1 <= R;
    end if;
  end process SYNDROME_GEN;

  DECODER : process ( S, CLOCK1, R1 )
  variable temp,temp1 : CODE_WORD;
  variable J : integer;
  begin
   if ( CLOCK1 = ’1’ ) then
    J := 0;
    for I in H_Tran’range loop
      if (S = H_Tran(I)) then
        J := I;
      end if;
    end loop;
    for I in temp’range loop
```

```
      if ( I = J ) then
        temp(I) := ’1’;
      else
        temp(I) := ’0’;
      end if;
    end loop;
    for I in R1’range loop
      temp1(I) := temp(I) xor R1(I);
    end loop;
    for I in K downto 1 loop
      DATA_OUT(I) <= temp1(I);
    end loop;
   end if;
  end process DECODER;
end BEHAVIORAL;

library IEEE, SYNOPSYS;
use IEEE.std_logic_1164.all;
use SYNOPSYS.attributes.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_misc.all;

package MATRIX_RELATED is
  constant K : integer := 4;
  constant N : integer := 7;
  type DATA_WORD is array(K downto 1) of std_logic;
  type CODE_WORD is array(N downto 1) of std_logic;
  type CHECK_WORD is array(N-K downto 1 ) of std_logic;
  type SYNDROME_WORD is array(N-K downto 1) of std_logic;
   type GENERATOR_MATRIX is array(K downto 1) of CODE_WORD;
  type PARITY_CHECK_MATRIX_TRAN is array(N downto 1) of
SYNDROME_WORD;
  type CHECK_MATRIX is array(K downto 1) of CHECK_WORD;
end MATRIX_RELATED;
```

Figure 13-8 shows the LINEAR_DECODER synthesis produced from the code shown in Example 13-9.

Figure 13-8 LINEAR_DECODER Synthesis



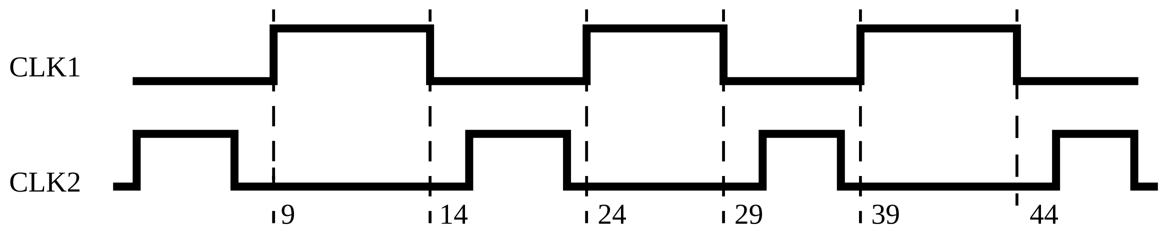
Setting Constraints on the Linear Block Encoder

You use the following commands to set constraints for the linear block encoder and decoder:

```
set_wire_load_min_block_size "05x05"
set_wire_load_model -library "lsi_10k"
set_operating_conditions -library "lsi_10k"  "WCCOM"
set_drive drive_of(lsi_10k/IV/Z) all_inputs()
set_load load_of(lsi_10k/IV/A) all_inputs()
set_load 0 find(port, CLOCK*)
create_clock -name CLK1 -period 15 -waveform {9, 14}
  find(port, CLOCK1)
create_clock -name CLK2 -period 15 -waveform {2,7}
  find(port, CLOCK2)
set_input_delay 4 -clock CLK2  -level_sensitive
   all_inputs()
set_output_delay 3 -clock CLK2 -level_sensitive
   all_outputs()
set_input_delay 0 find(port, CLOCK*)
```

Figure 13-9 shows the two-phase clocks for the linear block encoder and decoder produced by the preceding constraints.

Figure 13-9 Two-Phase Clocks for the LINEAR_DECO



report_timing Command Output

Example 13-10 shows a portion of the information produced by the following `report_timing` command:

```
report_timing -delay max -max_paths 20 command
```

Example 13-10 Report Timing Output Showing Relative Slack Balancing

```
Information: Updating design information... (UID-85)
Automatic time borrowing...

****************************************

Startpoint: R_reg[3] (positive level-sensitive latch clocked
by CLK1)
Endpoint: S_reg[2] (positive level-sensitive latch clocked
by CLK2)
Path Group: CLK2
Path Type: max

  Point                                  Incr       Path
-----------------------------------------------------
 clock CLK1 (rise edge)                  9.00      9.00
 clock network delay (ideal)             0.00      9.00
 R_reg[3]/G (LD1P)                       0.00      9.00 r
```

```
R_reg[3]/Q (LD1P)                    2.29     11.29 f
...
S_reg[2]/D (LD1P)                    7.16     18.45 r
data arrival time                             18.45

clock CLK2 (rise edge)              17.00     17.00
clock network delay (ideal)          0.00     17.00
S_reg[2]/G (LD1P)                    0.00     17.00 r
time borrowed from endpoint       0.09     17.09
data required time                            17.09
----------------------------------------------------
data required time                            17.09
data arrival time                            -18.45
------------------------------------------------------
slack (VIOLATED)                              -1.36

Time-Borrowing Information
-----------------------------------------------
CLK2 pulse width                               5.00
library setup time                            -0.60
 -----------------------------------------------
max time borrow                                4.40
actual time borrow                             0.09
------------------------------------------------
Startpoint: S_reg[2]
 (positive level-sensitive latch clocked by CLK2)
Endpoint: DATA_OUT_reg[3]
 (positive level-sensitive latch clocked by CLK1)
Path Group: CLK1
Path Type: max

 Point                                 Incr      Path
--------------------------------------------------------
 clock CLK2 (rise edge)                 2.00      2.00
 clock network delay (ideal)            0.00      2.00
 time given to startpoint            0.09      2.09
  S_reg[2]/D (LD1P)                     0.00      2.09 r
 S_reg[2]/Q (LD1P)                      2.89      4.98 r
 ...
 DATA_OUT_reg[3]/D (LD1)               10.32     15.30 r
data arrival time                                15.30
```

```
 clock CLK1 (rise edge)                  9.00       9.00
 clock network delay (ideal)             0.00       9.00
 DATA_OUT_reg[3]/G (LD1)                 0.00       9.00 r
 time borrowed from endpoint          4.25      13.25
 data required time                                13.25
-------------------------------------------------------
 data required time                                13.25
 data arrival time -15.30
------------------------------------------------------
 slack (VIOLATED)                                  -2.05


 Time-Borrowing Information
 ----------------------------------------------
 CLK1 pulse width                            5.00
 library setup time                         -0.40
 ----------------------------------------------
 max time borrow                             4.60
 actual time borrow                          4.25
 ----------------------------------------------
```

Figure 13-10 summarizes the `report_timing` output shown in Example 13-10. As Figure 13-10 shows, the cost is 3.77—2.41 + 1.36—and the relative slacks in paths II, III and IV are, respectively, 0, 0.36, and 0.11. These relative slacks are almost balanced. Note, however, that the absolute slacks in paths II, III, and IV are not balanced. The absolute slacks are, respectively, –1.36, –2.05, and –1.25.

Slacks in two paths that are in different path groups cannot be compared, but relative slack, which is the measure of the criticality of any path, can always be compared.

Figure 13-10   Timing Report Using Auto-Time Borrowing

Path Group	CLK1	CLK2	CLK1	CLK2
Critical Slack in Path Group	–2.41	–1.36	–2.41	–1.36
Slack in This path	2.55	–1.36	–2.05	–1.25
Relative Slack in This path	4.96	0	0.36	0.11

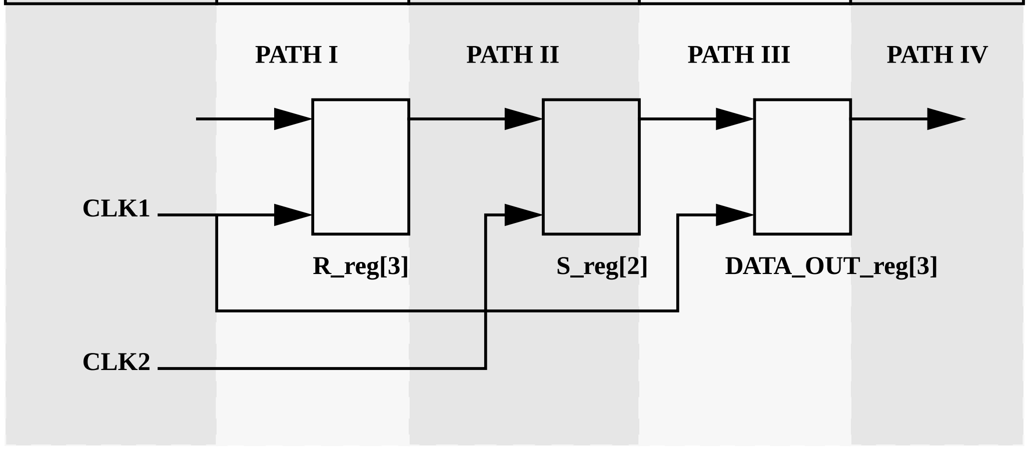


Figure 13-11 presents the timing report of the linear encoder and decoder design without the auto time-borrowing feature.

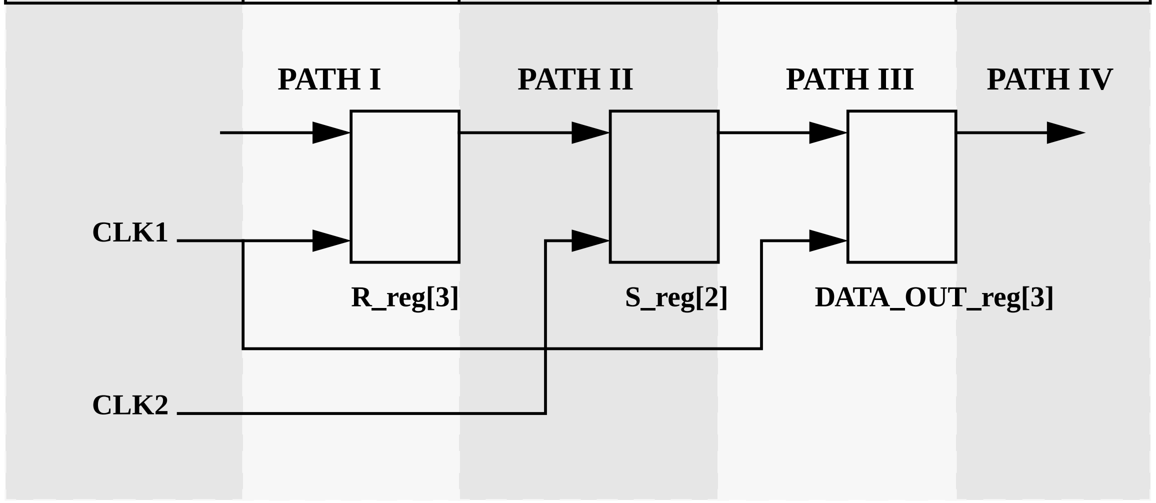
As shown in Figure 13-11, the cost is 5.13—3.77 + 1.36—and the relative slacks in paths II, III and IV are, respectively, 1.36, 0.39, and 0. These relative slacks are not balanced. Note that the absolute slacks in paths II, III, and IV are also not balanced. The absolute slacks are, respectively, –1.36, –2.05, and –1.25.

Note:

> In the 98.02 release, there is a hidden variable—tvc_auto_time_borrow—which is always set to 1.To set the timing report not to use auto-time borrowing, you must set this variable to 0 prior to generating the report.

Figure 13-11   Timing Report Without Using Auto-Time Borrowing

Path Group	CLK1	CLK2	CLK1	CLK2
Critical Slack in Path Group	–3.77	–1.36	–3.77	–1.36
Slack in This path	2.55	0	–3.38	–1.36
Relative Slack in This path	6.32	1.36	0.39	0



Results of Use of Auto-Time Borrowing

Use of auto-time borrowing, as shown in report timing for the example design, reduces the delay cost from 5.13 (3.77 + 1.36) to 3.77 (2.41 + 1.36).

Here are some comparisons of delay cost with and without use of auto-time borrowing:

Without use of auto-time borrowing,

- The path S-_reg[3] to DATA_OUT_reg[2] with a slack of –3.77 was the critical path in the path group CLK1.

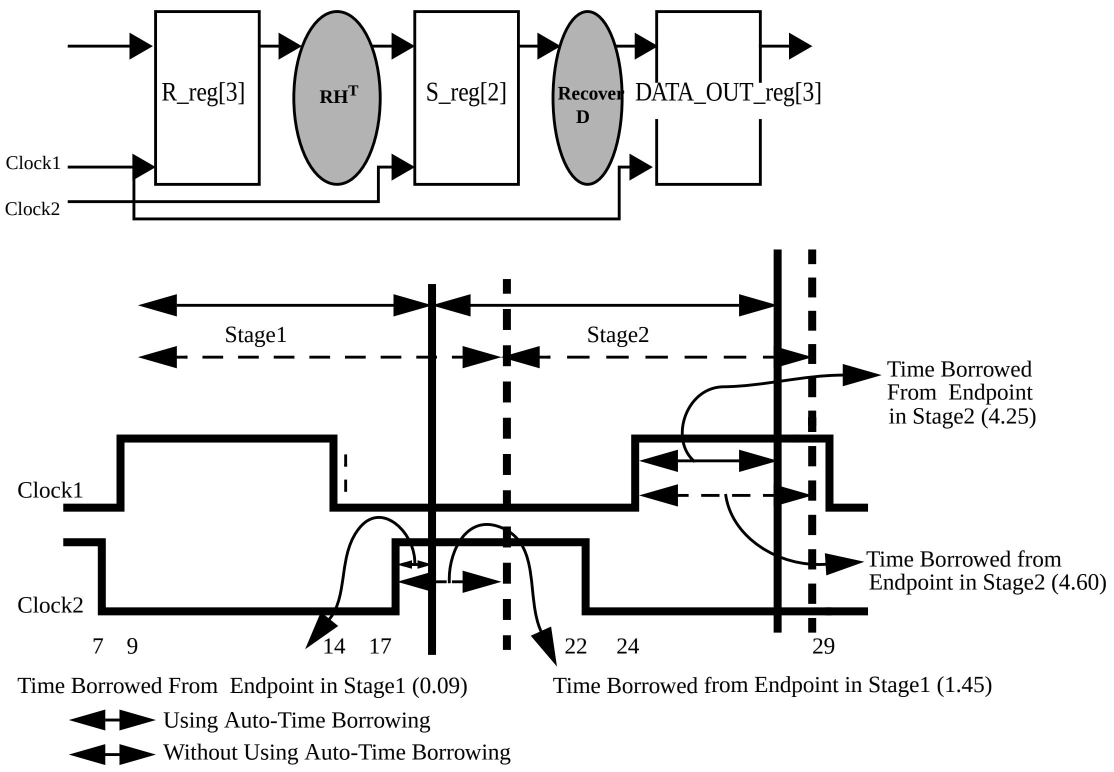
- The path DATA_OUT_reg[2] to DATA_OUT[1] with a slack of –1.36 was the critical path in the path group CLK2.
- The critical slack in group CLK1 is –3.77, and the critical slack in CLK2 is –1.36.

With use of auto-time borrowing,

- The slack in the path from S_reg[3] to DATA_OUT_reg[2] is reduced to –2.41, from –3.77.
- For the path group CLK2, the slack in the path from DATA_OUT_reg[2] to DATA_OUT[1] is reduced to –1.25 from –1.36. The path from R_reg[3] to S_reg[1] with a slack of –1.36 is now the critical path in group CLK2. (Without use of auto-time borrowing, this path had 0 slack.)
- The critical slack in CLK1 is –2.41, and the critical slack in CLK2 is –1.36.

Figure 13-12 shows how auto-time borrowing tried to balance the relative slacks in Paths II, III, and IV, which are shown in Figure 13-10 and Figure 13-11.

Figure 13-12 Balancing Relative Slacks by Reducing Borrowed Time


R_reg[3]
RHT
S_reg[2]
Recover D
DATA_OUT_reg[3]
Clock1
Clock2
Stage1
Stage2
Time Borrowed From Endpoint in Stage2 (4.25)
Clock1
Clock2
Time Borrowed from Endpoint in Stage2 (4.60)
7
9
14
17
22
24
29
Time Borrowed From Endpoint in Stage1 (0.09)
Time Borrowed from Endpoint in Stage1 (1.45)
Using Auto-Time Borrowing
Without Using Auto-Time Borrowing

A

Static Timing Analysis

Design Compiler’s timing analyzer provides accurate static timing information for all timing paths in the current design. This enables Design Compiler to optimize your design and identify critical paths. A delay model calculates the sources of delay, including propagation and interconnect delays (net transit time).

The timing analyzer computes each gate and interconnect delay, then traces critical paths, calculating minimum and maximum arrival times to points of interest. The timing analyzer uses the critical path values to evaluate design constraints and create timing reports.

The timing analyzer finds violations of design constraints and directs the optimization process to those parts of the design. Design Compiler uses a sophisticated incremental update algorithm to calculate circuit timing. This algorithm recalculates only parts of the network that changed during the last optimization step.

To understand the timing analyzer, you need to know about

- Path-Based Timing Optimization
- Delay Models
- Environmental Scaling

Path-Based Timing Optimization

The timing analyzer calculates minimum and maximum path delay costs during optimization and allows Design Compiler to make optimization decisions that improve delay cost. It provides fast timing updates as the design is changed during optimization and has advanced features to support

- Multiphase and multifrequency clocking
- Automatic time borrowing for latch-based design
- User-specified multicycle and false paths
- Specific point-to-point path delay requirements

You define port signal timing and clock waveforms, and the timing analyzer determines the required maximum and minimum path delays for each timing path in the design. These requirements are compared with the actual path delay to determine slack, which is the difference between the actual and required arrival times.

You can display design timing information with the `report_timing` command, described in Chapter 11, “Analyzing Timing.”

The timing analyzer represents a netlist as a directed graph in which nodes in the graph represent pins in the logic network. Edges between nodes (timing arcs) represent two types of connections:

- Net delay—interconnect relationships between a driver pin and its fanout.
- Cell delay—timing relationships between an input pin and an output pin.

Delay analysis is the calculation of each timing arc's value, which is either a cell delay or a net delay (interconnect). Rise and fall delay values for a timing path are calculated by addition of the timing arc values.

positive unate timing arc

> Combines rise delays with rise delays and fall delays with fall delays. Examples are an AND gate cell delay and an interconnect (net) delay.

negative unate timing arc

> Combines incoming rise delays with local fall delays and incoming fall delays with local rise delays. An example is a NAND gate.

non-unate timing arc

> Combines local delay with the worst-case incoming delay value. Non-unate timing arcs are present in logic functions whose output value change cannot be predicted by the direction of the change on the input value. An example is an XOR gate.

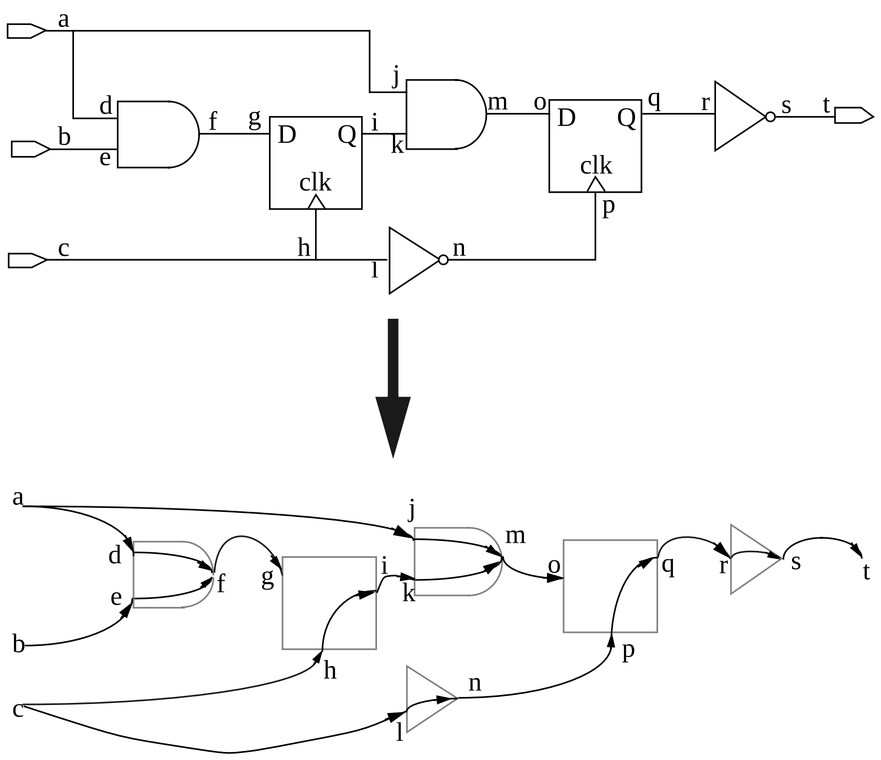
Because the delay attributes are associated with a timing arc and not with a single pin, both minimum and maximum paths between two pins can be modeled.

Note:

The timing analyzer does not store delay equation parameters in terms of specific units. The only restriction the timing analyzer places on the values used is that the units must form an internally consistent system. Synopsys engineers suggest using a system of nanoseconds (delay), picofarads (load), and kilohms (resistance).

Figure A-1 shows how a logic network is converted to a timing graph.

Figure A-1 Converting a Logic Network to a Timing Graph



The timing arcs from the clock inputs (clk) of the flip-flops to the data outputs (q) of the flip-flops. No direct timing arcs exist from a flip-flop's D input to its Q output.

Reporting Arc Delay Calculations

The `report_delay_calculation` command prints the details of a timing arc delay calculation. The details include cell arcs (from an input pin to an output pin) as well as net arcs (from a source pin to a load pin).

The syntax is

```
report_delay_calculation -from from_pin -to to_pin
    [-min | -max][-nosplit]
```

`-from` *from_pin*

Names the input or source pin.

`-to` *to_pin*

Names the output or load pin.

`-min`

Shows how a minimum delay value is computed.

`-max`

Shows how a maximum delay value is computed. This option is the default. If neither `-min` nor `-max` is specified, the report shows how a maximum delay value is computed.

`-nosplit`

Prevents line splitting in the report.

Note:

The `report_delay_calculation` command does not print delay information unless the library was loaded in source format (using the `read_lib` command).

If a cell timing arc exists between the given pins, the details of the cell arc delay calculation are provided. If the given pins are part of the same net, the details of the net arc delay calculation are provided.

The following conditions constitute an error (no delay information is printed):

- The technology library is not loaded in source format.
- No cell or net delay arc exists between the given pins.
- An undefined pin is specified.
- The pins are associated with a nonleaf cell.
- More than one pin is defined with the `-from` or `-to` option.
- The arc between the defined pins is not supported.

The format of the output varies on the basis of delay model type and the interconnect delay tree type (for net delay arcs). If delay values have been back-annotated for the defined arc, the annotated delay is given instead of the calculated delay.

Example

This example shows a sample output for a cell arc and a net arc. The library is generic_cmos, and the tree type is balanced_case_tree.

```
From pin: U28/A
To pin: U28/Z

arc type :                      cell
arc sense :                     unate
Input net transition times:     Dt_rise = 0.1458, Dt_fall = 0.0653

Rise Delay computation:
rise_intrinsic                  0.48 +
rise_slope * Dt_rise            0 * 0.1458 +
rise_resistance * (pin_cap + wire_cap) / driver_count
0.1443 * (2 + 0) / 1
rise_transition_delay :         0.2886
------------------------------------
Total                           0.7686


Fall Delay computation:
fall_intrinsic                  0.77 +
fall_slope * Dt_fall            0 * 0.0653 +
fall_resistance * (pin_cap + wire_cap) / driver_count
0.0523 * (2 + 0) / 1
fall_transition_delay :         0.1046
------------------------------------
Total                           0.8746

Net arc output
    From pin:             U28/Z
    To Pin:               U29/A

    arc type: net
    Wire Loading Model Mode: top

    Design            Wire Loading Model         Library
    ----------------------------------------------------
    RDC_GENERIC             BASIC_ONE            basic
    Operating Conditions: BASIC_WORST   Library: basic
    Balanced case tree
    equation : (r_wire/load_count) * (c_pins + c_wire/load_count)
    (0 / 1) * (1 + 0 / 1)
    delay rise, fall :    0 , 0
```

Debugging Delay Calculations Along a Critical Path

A typical use of the `report_delay_calculation` command is to assist in debugging the delay calculations along a critical path.

To locate the critical path, use `report_timing -path full -input_pins`. You get a listing similar to the following.

```
Point                                   Incr     Path
-------------------------------------------------------------------------
input external delay                    0.00     0.00 f
i2 (in)                                 0.00     0.00 f
cell1/i2 (lower1)                       0.00     0.00 f
cell1/C/B (AN2)                         0.00     0.00 f
cell1/C/Z (AN2)                         0.82     0.82 f
cell1/o1 (lower1)                       0.00     0.82 f
cell2/i1 (lower2)                       0.00     0.82 f
cell2/C/A (IV)                          0.00     0.82 f
cell2/C/Z (IV)                          0.38     1.20 r
cell2/o1 (lower2)                       0.00     1.20 r
o1 (out)                                0.00     1.20 r
data arrival time                                1.20
```

The `-input_pins` option causes the input pins to be listed in the path, in addition to the output pins.

Reporting Details of a Cell Delay Arc

Here are some examples of how to print details of a cell delay and net delay arcs.

- To print details of a cell delay arc, use the `report_delay_calculation` command by defining the input and output pin of a leaf cell along the path. For example, enter

  ```
  report_delay_calculation -from cell1/C/B -to cell1/C/Z
  ```

- To print the details of a net delay arc, give the command a driver and a load pin on the same net along the path (the pins must be associated with leaf cells). For example, enter

  ```
  report_delay_calculation -from cell1/C/Z -to cell2/C/A
  ```

  This example is valid, because both cells are leaf cells.

- The following command is not valid, because there is no net delay arc associated with from_pin (it is not on a leaf cell):

  ```
  report_delay_calculation -from cell2/i1 -to cell2/C/A
  ```

The operating conditions and wire load are taken into account when generating the report data but timing ranges are not, because timing ranges typically apply to an entire path (as opposed to a single timing arc).

Delay Models

The timing analyzer uses the timing parameters and the environment attributes described in a technology library to calculate timing delays for designs.

The types of delay analysis are

- Linear (generic_cmos)
- CMOS2 (cmos2)
- Nonlinear (table_lookup)

Note:

> The capacitance of all pins on an interconnected wire contributes to the delay. In this section, references to pins or sums over pins include all pins, both driver and input, unless stated otherwise. In many libraries, however, the delay contribution due to the capacitance of an output pin is assigned to other delays and the capacitance is set to zero. For example in Figure A-3 on page A-18, the calculation uses capacitance values for the three input pins and none for the output pin. This produces the correct result when the library includes a capacitance of zero for the output pin.

Linear Delay Model

The linear delay equation for computing gate delay values is the sum of four terms:

slope delay (DS)

> The delay incurred from an input pin to an output pin because of a slow logic transition at the input pin.

intrinsic delay (DI)

> The built-in delay from an input pin to an output pin.

transition time (DT)

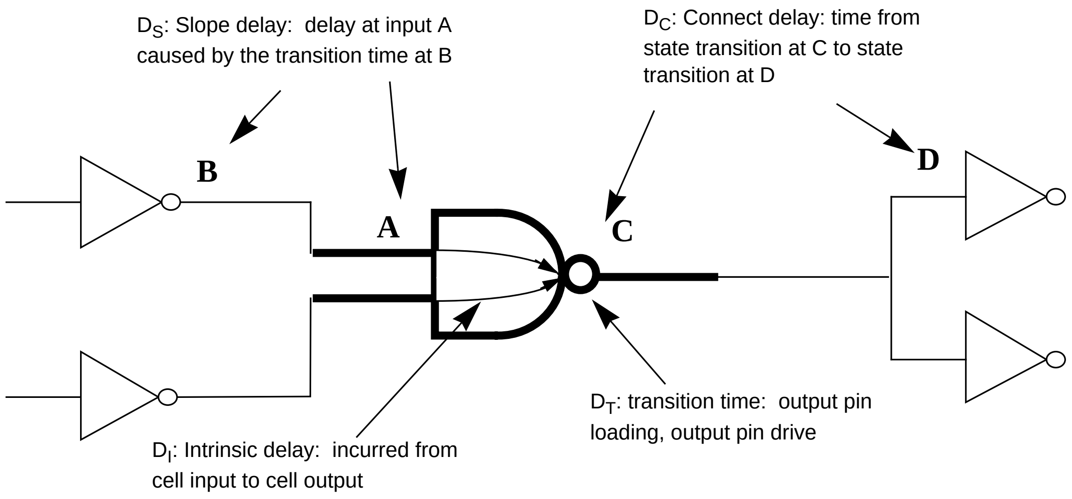
> The time a state change takes to complete on a net. This can be constrained as a design rule (max_transition) and can also be used as a parameter in delay calculations for cells in the fanout of the net.

connect delay (DC)

The time-of-flight delay—the time a logic transition takes to propagate through an interconnect network.

Figure A-2 shows a diagram of the four terms in the delay equation.

Figure A-2 Delay Equation Terms and Timing Arcs (Linear)



D_S: Slope delay: delay at input A caused by the transition time at B
D_C: Connect delay: time from state transition at C to state transition at D
B
D
A
C
D_T: transition time: output pin loading, output pin drive
D_I: Intrinsic delay: incurred from cell input to cell output

The timing analyzer does not store delay equation parameters in terms of specific units. The only restriction the timing analyzer places on the values used is that the units must form an internally consistent system. Synopsys engineers suggest a system of nanoseconds (delay), picofarads (load), and kilohms (resistance).

Example

This example shows the rise and fall delay equations.

```
Drise = Dslope-rise + Dintrinsic-rise + Dtransition-rise + Dconnect-rise

Dfall = Dslope-fall + Dintrinsic-fall + Dtransition-fall + Dconnect-fall
```

The parameters of these equations are

$$D_{\text{slope-rise}} = D_{\text{Tprevious\_stage}} \times S_{\text{rise}}$$
$$D_{\text{slope-fall}} = D_{\text{Tprevious\_stage}} \times S_{\text{fall}}$$

The $D_{\text{Tprevious\_stage}}$ value is determined by the arc type.

For rise:

arc_type	$D_{\text{Tprevious\_stage}}$
positive unate	rise
negative unate	fall
nonunate	max(rise,fall)

For fall:

arc_type	$D_{\text{Tprevious\_stage}}$
positive unate	fall
negative unate	rise
nonunate	max(rise,fall)

S_{rise} = input rise slope sensitivity

S_{fall} = input fall slope sensitivity

$D_{\text{intrinsic-rise}}$ = Intrinsic rise delay from library

$D_{intrinsic\text{-}fall}$ = Intrinsic fall delay from library

For non-piecewise:

$D_{transition\text{-}rise} = R_{rise} (C_{pins} + C_{wire})$/(number of non-three-state drivers)
$D_{transition\text{-}fall} = R_{fall}(C_{pins} + C_{wire})$/(number of non-three-state drivers)

For piecewise:

$D_{transition\text{-}rise} = R_{drivei\text{-}rise} \times (C_{pins} + C_{wire}) + Y_{adji}$ /
(number of non-three-state-drivers)

$D_{transition\text{-}fall} = R_{drivei\text{-}fall} \times (C_{pins} + C_{wire})$ + Yadji /
(number of non-three-state-drivers)

```
D_connect-rise = D_connect-fall = {
        if tree type is best case: 0.0
        if tree type is worst case: R_wire (C_wire + C_pin)
        if tree type is balanced: R_wire/N (C_wire/N+C_pin)}
```

R_{wire} = wire resistance

c_{pins} = sum of all pin capacitances on net

C_{pin} = individual pin capacitance value

N = number of pins being driven

$D_{T\text{-}rise\_previous\_stage}$ = transition rise delay at previous stage

$D_{T\text{-}fall\_previous\_stage}$ = transition fall delay at previous stage

Slope Delay (D_S)

The slope delay of an element, D_S, represents the gate delay resulting from the ramp time of the input signal. A slower input transition results in more slope delay.

```
Ds = DTprevious x S
```

$D_{Tprevious}$

The transition time of the previous gate (see “Transition Time (DT)” on page A-14). The appropriate transition direction is selected based on the unateness of the timing arc being evaluated.

S

The slope delay factor for the specified timing arc (slope sensitivity).

Intrinsic Delay (D_I)

The intrinsic delay of an element, D_I, is the portion of the total delay that is independent of the element’s usage. This is the fixed (or zero-load) delay of an element.

The total intrinsic delay is calculated by scaling these constant values by their corresponding k-factors (see “Environmental Scaling” on page A-40).

Transition Time (D_T)

Transition time is the delay the capacitive load on a gate’s output pin introduces. It represents the time it takes the output to switch from one logical state to another. The transition time is computed in one of two ways, depending on the technology library:

Linear transition time

$$D_T = R_{drive} \times \left(\sum_{pins} C_{pin} + C_{wire} \right)$$

Piecewise linear transition time

$$D_T = R_{drivei} \times \left(\sum_{pins} C_{pin} + C_{wire} \right) + Y_{adji}$$

In the piecewise linear model, the resistance value (R_{drivei}) and a constant term (Y_{adji}) can vary with different loading conditions. The piece_type statement in the technology library determines how the appropriate resistance and constant are selected. The selection is done on the basis of one of the following criteria:

- Total net length
- Total output capacitance
- Output pin capacitance
- Output wire capacitance

A piece_define statement in the library determines the correlation between resistance and constant term values and the piece_type. In the case of parallel drivers, the arc transition time is divided by the number of non-three-state drivers.

The total transition time is calculated by scaling the constant values by their corresponding k-factors (see “Environmental Scaling” on page A-40).

Connect Delay (D_C)

The connect delay, D_C, is the time it takes the voltage at an input pin to change after the transition of the driving output pin. Connect delay is also called the time-of-flight delay (the time it takes for a waveform to travel along a wire). The way this delay is calculated is important in the analysis of interconnect network delay. The timing analyzer supports three cases for an estimated interconnect topology (tree type).

- Best case (best_case_tree) models the case where the load pin is physically adjacent to the driver. All wire capacitance is incurred, but none of the wire resistance must be overcome. The best-case connect delay is calculated from the following equation. Because Rwire is always 0 in this case, the resulting D_C is always 0.

$$D_{C_{best}} = R_{wire}(C_{wire} + C_{pin}) = 0$$

- Balanced case (balanced_tree) models the case where all load pins are on separate, equal branches of the interconnect wire. Each load pin incurs an equal portion of the wire capacitance and wire resistance.

$$D_{C_{balanced}} = \frac{R_{wire}}{N}\left(\frac{C_{wire}}{N} + C_{pin}\right)$$

- Worst case (worst_case_tree) models the case where the load pin is at the extreme end of the wire. Each load pin incurs both the full wire capacitance and the full wire resistance.

$$D_{C_{worst}} = R_{wire}\left(C_{wire} + \sum_{pins} C_{pin}\right)$$

The three previous equations are used for calculating both the rise and fall delays. The components of these equations are described below. Where applicable, use the rise_ parameter for calculating the rise delay and the fall_ parameter for calculating the fall delay.

R_{wire}

Estimated wire resistance on the net determined by the wire load model. Wire length is computed with a global estimation function whose parameter is the number of fanout pins on the net being estimated.

C_{wire}

The estimated wire capacitance for the net attached to the head of the timing arc for which the delay value is being computed. Wire length is computed with the actual number of fanout pins on the net being estimated and the fanout_length specifications in the wire_load group. The estimated value is scaled by the capacitance factor.

C_{pin}

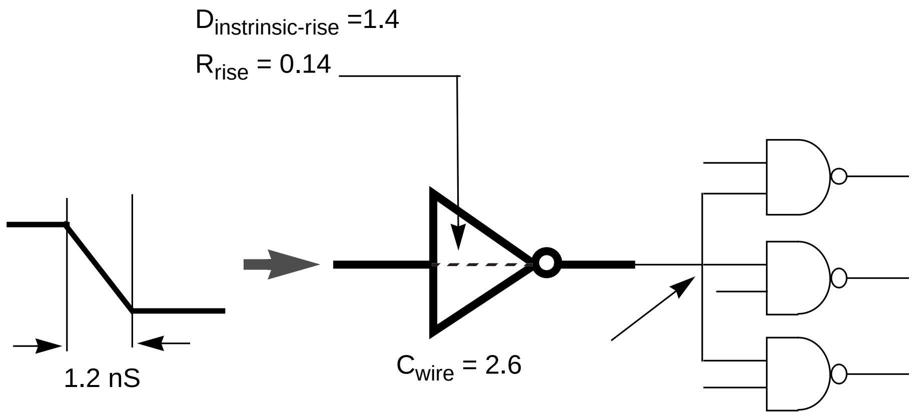
Capacitance values for the load pin.

Total connect delay is calculated by scaling the constant values by their corresponding k-factors (see “Environmental Scaling” on page A-40).

Delay Calculation (Linear) Example

Figure A-3 shows the rise delay values for an inverter.

Figure A-3   Delay Values for an Inverter



The inverter input pin is driven by a falling signal with a transition time (DT) of 1.2 ns. The inverter fans out to three NAND gates, each with an input pin capacitance of 1.1. The inverter has an intrinsic rise delay of 1.4, a rise slope sensitivity of 0.02, and an output rise resistance of 0.14. Assuming a best-case RC-interconnect tree type and an estimated interconnect wire capacitance of 2.6, the rise delay is 2.25.

The following is the rise delay calculation for the inverter shown in Figure A-3:

$D_{intrinsic\text{-}rise} = 1.4$
$S_{rise} = 0.02$
$R_{rise} = 0.14$

$C_{pins} = 3 * (1.1) = 3.3$

$C_{wire} = 2.6$

$D_{T\text{-}fall\_previous\_stage} = 1.2$

$D_{connect\text{-}rise} = 0.0$ for a best case RC-tree

D_{rise} = Intrinsic + Slope + Transition + Connect
$= 1.4 + (1.2 * 0.02) + (0.14)(3.3 + 2.6) + 0.0$

```
        = 1.4  +  0.02  +  0.826 + 0.0
        = 2.25
```

CMOS2 Delay Model

The CMOS2 delay model is similar to the linear delay model that except it uses delay due to edge rate instead of slope delay. The CMOS2 delay equation for computing gate delay values is the sum of four terms:

intrinsic delay (D_I)

The built-in delay from an input pin to an output pin.

delay due to input edge rate (D_E)

The delay incurred at an input pin due to edge rate. At each input pin, the timing analyzer computes the actual edge rate. The edge rate can be different for different pins on the net. Also, the cell delay can have a two-piece dependency on input edge rate. See Figure A-4 for an example of an input rise dependency.

transition delay (D_T)

The time it takes to change logic value due to loading at an output pin (output resistance times load).

connect delay (D_C)

The time-of-flight delay—the time it takes a logic transition to propagate through an interconnect network.

Figure A-4 Rise Dependency on Edge Rate (CMOS2)

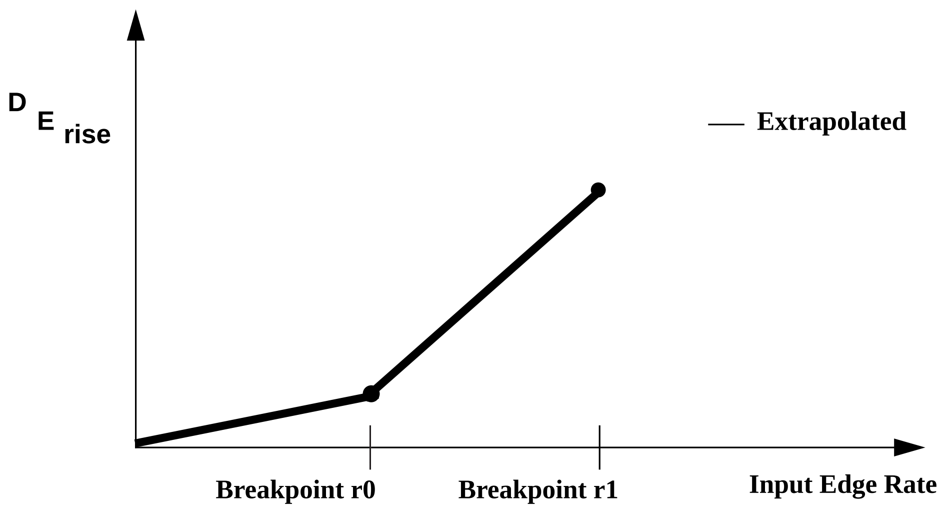
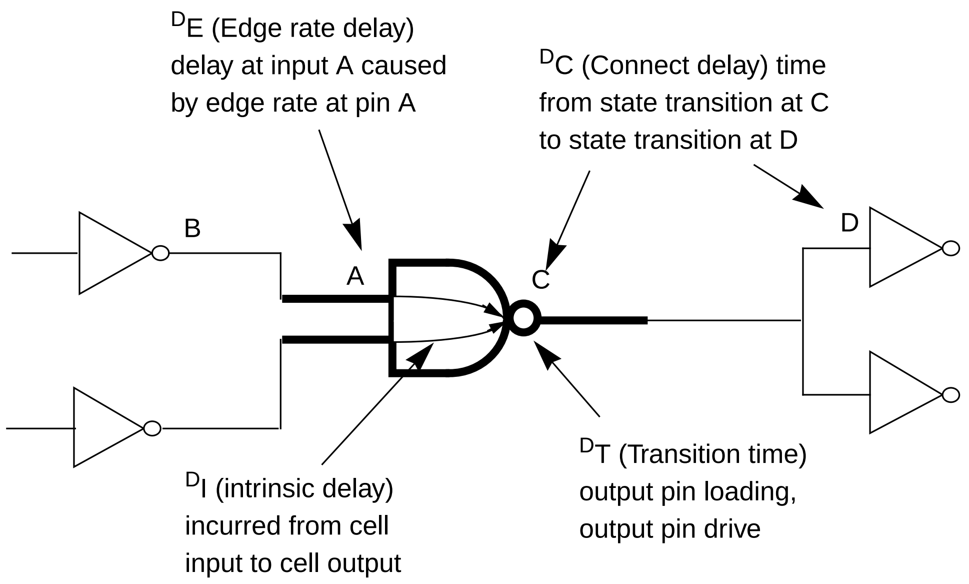


Figure A-5 shows a diagram of the four terms in the delay equation.

Figure A-5 Delay Equation Terms and Timing Arcs (CMOS2)


DE (Edge rate delay) delay at input A caused by edge rate at pin A
DC (Connect delay) time from state transition at C to state transition at D
B
A
C
D
DI (intrinsic delay) incurred from cell input to cell output
DT (Transition time) output pin loading, output pin drive

The timing analyzer does not store delay equation parameters in terms of specific units. The only restriction the timing analyzer places on the values used is that the units must form an internally consistent system. Synopsys engineers suggest a system of nanoseconds (delay), picofarads (load), and kilohms (resistance).

Example

This example shows the rise and fall delay equations.

- For positive unate arcs:

  $D_{rise} = D_{edge\text{-}rise} + D_{intrinsic\text{-}rise} + D_{transition\text{-}rise}$
  $D_{fall} = D_{edge\text{-}fall} + D_{intrinsic\text{-}fall} + D_{transition\text{-}fall}$

- For negative unate arcs:

  $D_{rise} = D_{edge\text{-}fall} + D_{intrinsic\text{-}rise} + D_{transition\text{-}rise}$
  $D_{fall} = D_{edge\text{-}rise} + D_{intrinsic\text{-}fall} + D_{transition\text{-}fall}$

- For non-unate arcs:

  $D_{rise} = D_{edge\text{-}max} + D_{intrinsic\text{-}rise} + D_{transition\text{-}rise}$
  $D_{fall} = D_{edge\text{-}max} + D_{intrinsic\text{-}fall} + D_{transition\text{-}fall}$

Equation Parameters

The parameters of these equations are

```
Dedge-rise = edge_rate_sensitivity_r0  x
        min(edge_rate_breakpoint_r1 – edge_rate_breakpoint_r0,
        edge_rate_rise_delay – edge_rate_breakpoint_r0) +
        edge_rate_sensitivity_r1 x max(0, edge_rate_rise –
        edge_rate_breakpoint_r1)
Dedge-fall = edge_rate_sensitivity_f0  x
        min(edge_rate_breakpoint_f1 – edge_rate_breakpoint_f0,
        edge_rate_fall_delay – edge_rate_breakpoint_f0) +
        edge_rate_sensitivity_f1 x max(0, edge_rate_fall –
        edge_rate_breakpoint_f1)
edge_rate_sensitivity_r0 = (arc) rising edge-rate sensitivity of first piece
edge_rate_sensitivity_r1 = (arc) rising edge-rate sensitivity of second piece
```

edge_rate_breakpoint_r0 = (input pin) first breakpoint on rising edge-rate sense curve
edge_rate_breakpoint_r1 = (input pin) second breakpoint on rising edge-rate sense curve
edge_rate_rise_delay = edge_rate_rise + edge_rate_load_rise x (interconnect and input pin capacitance – reference capacitance of the output pin driving the net + (estimated connect delay/rise resistance of driving cell))
edge_rate_rise = the zero-load rise edge rate for a driver pin
edge_rate_sensitivity_f0 = (arc) falling edge-rate sensitivity of first piece
edge_rate_sensitivity_f1 = (arc) falling edge-rate sensitivity of second piece
edge_rate_breakpoint_f0 = (input pin) first breakpoint on falling edge-rate sense curve
edge_rate_breakpoint_f1 = (input pin) second breakpoint on falling edge-rate sense curve
edge_rate_fall_delay = edge_rate_fall + edge_rate_load_fall x (input net capacitance – reference capacitance + (estimated connect delay/fall resistance of driving cell))
edge_rate_fall = the zero-load fall edge rate for a driver pin
edge_rate_load_fall = the load dependent falling edge rate capability for a driver pin.
edge_rate_load_rise = the load dependent rising edge rate capability for a driver pin.
Dintrinsic-rise = Intrinsic rise delay from library
Dintrinsic-fall = Intrinsic fall delay from library

For non-piecewise:
$D_{transition-rise} = R_{rise} \times (C_{pins} + C_{wire})$
$D_{transition-fall} = R_{fall} \times (C_{pins} + C_{wire})$

For piecewise:
$D_{transition-rise} = R_{drivei-rise} \times (C_{pins} + C_{wire}) + Y_{adji}$
$D_{transition-fall} = R_{drivei-fall} \times (C_{pins} + C_{wire}) + Y_{adji}$

$D_{connect-rise}$ = { if tree type is best case: 0.0
if tree type is worst case: $(R_{wire}/N\ (C_{wire} + C_{pin}) \times K_{rc\_rise})$
if tree type is balanced: $(R_{wire}/N\ (C_{wire}/N + C_{pin}) \times K_{rc\_rise})$ }

$D_{connect-fall}$ = { if tree type is best case: 0.0
if tree type is worst case: (Rwire/N (Cwire + Cpin) x K_{rc_fall})
if tree type is balanced: $(R_{wire}/N\ (C_{wire}/N + C_{pin}) \times K_{rc\_fall})$ }

R_{wire} = Wire resistance

C_{pins} = Sum of all pin capacitances on net
C_{pin} = Individual pin capacitance value

N = Number of pins being driven

K_{rc_rise} = Library multiplication factor

Edge Rate Delay (D_E)

The delay due to the input edge rate of an element, DE, represents the delay incurred at an input pin due to edge rate.

Intrinsic Delay (D_I)

The intrinsic delay of an element, DI, is the portion of the total delay that is independent of the element’s use. This is the fixed (or zero-load) delay of an element.

Total intrinsic delay is calculated by scaling these constant values by their corresponding k-factors (see “Environmental Scaling” on page A-40).

Transition Time (D_T)

Transition time is the delay introduced by the capacitive load on a gate’s output pin. It represents the time it takes the output to switch from one logical state to another. Transition time is computed in one of two ways, depending on the technology library:

Equation A-1 Linear transition time

$$D_T = R_{drive} \times \left(\sum_{pins} C_{pin} + C_{wire} \right)$$

Equation A-2   Piecewise linear transition time

$$D_T = R_{drivei} \times \left(\sum_{pins} C_{pin} + C_{wire} \right) + Y_{adji}$$

In the piecewise linear model, the resistance value (R_{drivei}) and a constant term (Y_{adji}) can vary with different loading conditions. The piece_type statement in the technology library determines how the appropriate resistance and constant are selected. The selection is done on the basis of one of the following criteria:

- Total net length
- Total output capacitance
- Output pin capacitance
- Output wire capacitance

A piece_define statement in the library determines the correlation between resistance and constant term values and the piece_type.

Total transition time is calculated by scaling the constant values by their corresponding k-factors (see “Environmental Scaling” on page A-40).

Connect Delay (D_C)

Connect delay, D_C, is the time it takes the voltage at an input pin to change after the transition of the driving output pin. Connect delay is also called the time-of-flight delay (the time it takes for a waveform to travel along a wire). The way this delay is calculated is important in the analysis of interconnect network delay. The timing analyzer supports three cases for an estimated interconnect topology (tree type).

- Best case (best_case_tree) models the case where the load pin is physically adjacent to the driver. All wire capacitance is incurred, but none of the wire resistance must be overcome. The best-case connect delay is calculated from the following equation. Because Rwire is always 0 in this case, the resulting D_C is always zero.

$$D_{C_{best}} = R_{wire}(C_{wire} + C_{pin}) = 0$$

- Balanced case (balanced_tree) models the case where all load pins are on separate, equal branches of the interconnect wire. Each load pin incurs an equal portion of the wire capacitance and wire resistance.

$$D_{C_{balanced}} = \frac{R_{wire}}{N}\left(\frac{C_{wire}}{N} + C_{pin}\right)$$

- Worst case (worst_case_tree) models are different for the CMOS2 model than they are for other models (the resistance is divided by the fanout N). The load pin is at the extreme end of the wire. Each load pin incurs both the full wire capacitance and a portion of the wire resistance.

$$D_{C_{worst}} = \frac{R_{wire}}{N}\left(C_{wire} + \sum_{pins} C_{pin}\right)$$

These equations are used for calculating both the rise and fall delays. Where applicable, use the rise_ parameter for calculating the rise delay and the fall_ parameter for calculating the fall delay. Descriptions of the components in the equation follow.

R_{wire}

Estimated wire resistance on the net determined by the wire load model. Wire length is computed with a global estimation function whose parameter is the number of fanout pins on the net being estimated. The estimated value is scaled by the resistance factor.

C_{wire}

Estimated wire capacitance for the net attached to the head of the timing arc for which the delay value is being computed. Wire length is computed with the actual number of fanout pins on the net being estimated and the fanout_length specifications in the wire_load group. The estimated value is scaled by the capacitance factor.

C_{pin}

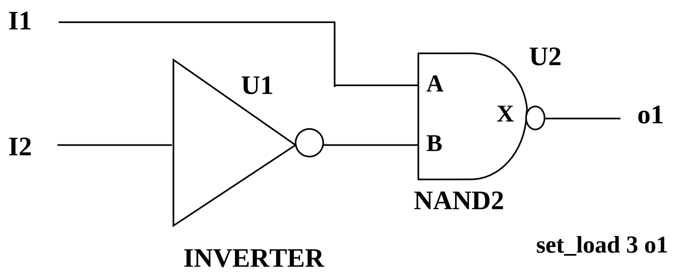
Capacitance values for the load pin.

Total connect delay is calculated by scaling the constant values by their corresponding k-factors (see “Environmental Scaling” on page A-40).

Delay Calculation (CMOS2) Example

This example shows the cell delay across a NAND2 gate from pin B to pin X.

```
wire_load_model: MEDIUM
tree_type: balanced_case
operating conditions: nominal
port_edge_rate: 0.0
```



The previous driver, an INVERTER gate, has the following characteristics:

```
  cell(INVERTER) {
    area : 1;
    pin(X) {
      function : "A'";
      direction : output;
      edge_rate_rise : 0.12;
      edge_rate_fall : 0.13;
      edge_rate_load_rise : 4.5;
      edge_rate_load_fall : 2.5;
      timing() {
      edge_rate_sensitivity_r0 : 0.20;
      edge_rate_sensitivity_f0 : 0.10;
      edge_rate_sensitivity_r1 : 0.15;
      edge_rate_sensitivity_f1 : 0.05;
      intrinsic_rise  : 0.10;
      intrinsic_fall  : 0.12;
      rise_resistance : 2.0;
      fall_resistance : 1.0;
      related_pin : "A";
      }
    }
    pin(A) {
      direction : input;
      capacitance :  0.10;
    }
  }
```

NAND2 Gate Library Description

The NAND2 gate library description is

```
  cell(NAND2) {
    area : 1;
    pin(X) {
      function : "(A B)'";
      direction : output;
      edge_rate_rise : 0.24;
      edge_rate_fall : 0.14;
      edge_rate_load_rise : 5.4;
      edge_rate_load_fall : 3.4;
      timing() {
      intrinsic_rise  : 0.34;
      intrinsic_fall  : 0.24;
      rise_resistance : 3.4;
      fall_resistance : 1.4;
      edge_rate_sensitivity_r0 : 0.24;
      edge_rate_sensitivity_f0 : 0.14;
      edge_rate_sensitivity_r1 : 0.14;
      edge_rate_sensitivity_f1 : 0.04;
      related_pin : "A";
      }
      timing() {
      intrinsic_rise  : 0.34;
      intrinsic_fall  : 0.24;
      rise_resistance : 3.4;
      fall_resistance : 1.4;
      edge_rate_sensitivity_r0 : 0.24;
      edge_rate_sensitivity_f0 : 0.14;
      edge_rate_sensitivity_r1 : 0.14;
      edge_rate_sensitivity_f1 : 0.04;
      related_pin : "B";
      }
    }
    pin(A) {
      direction : input;
      capacitance : 0.10;
    }
    pin(B) {
      direction : input;
      capacitance : 0.10;
    }
  }
```

Library Global Values

The library global values are

```
  delay_model : cmos2;
  time_unit : "1ns";
  default_max_transition : 12.00;

  default_edge_rate_breakpoint_r0 : 0.500;
  default_edge_rate_breakpoint_r1 : 3.000;
  default_edge_rate_breakpoint_f0 : 0.500;
  default_edge_rate_breakpoint_f1 : 3.000;
  default_reference_capacitance : 0.000;
  default_setup_coefficient : 1.0;
  default_hold_coefficient : 1.0;
  default_rc_rise_coefficient : 1.0;
  default_rc_fall_coefficient : 1.0;

  wire_load("MEDIUM") {
    resistance : 0.00003;
    capacitance : 0.0001;
    area : 0;
    fanout_length(1,800);
    fanout_length(6,2800);
    fanout_length(7,3200.0);
    fanout_length(14,6000.0);
    fanout_length(15,6500.0);
    slope : 400;
  }
```

For this example, a balanced-case tree type is used with the MEDIUM wire load model, and nominal operating conditions are assumed. Net N1 connects the driver, an INVERTER pin X, to the NAND2 pin B.

This net has the following characteristics:

```
pin_capacitance :       0.1
wire_capacitance :      0.08
total_capacitance :     0.18
wire_resistance :       0.024
```

Consider the rising cell delay across the NAND2 (B to X). The rising delay on NAND2 pin X requires a fall transition on the input B.

rise_cell_delay = $D_{e(input\ falling)} + D_{i\_rise} + D_{t\_rise}$

To determine the rise cell delay:

```
drive_pin = u1/X (INVERTER)
```

The effective capacitance noted by the previous driver is used to calculate the actual falling edge rate at B (ERFD). The fall_resistance is from the previous driver, the INVERTER cell.

```
c_eff = total_net_cap - drive_pin_cap
        - drive_pin_reference_capacitance
        + fall_connect_delay / drive_pin_fall_resistance
c_eff = 0.18 - 0 - 0 + 0.00432 / 1
c_eff = 0.18432

ERFD = driver_edge_rate_fall + driver_edge_rate_load_fall * c_eff
ERFD = 0.13 + 2.5 * 0.18432
ERFD = 0.5908
```

The delay due to input edge rate is a two-segment, piecewise linear dependence on ERFD. The breakpoints are found in the library. Calculating the delay due to edge_rate (De_fall) is as follows:

```
De_fall = edge_rate_sensitivity_f0 * min(bkpt_f1-bkpt_f0,ERFD-bkpt_f0) +
          edge_rate_sensitivity_f1 * max(0.0, ERFD-bkpt_f1)
De_fall = 0.14 * min(3-0.5,0.5908-0.5) + 0.04 * max(0.0, 0.5908-3)
De_fall = 0.012712
```

The intrinsic rise of the NAND2 is from the library.

```
Di_rise = rise_intrinsic
Di_rise = 0.34
```

The timing analyzer computes the NAND2 transition time, using the rise_resistance of this cell and the capacitance at net O1, which the NAND2 drives.

```
Dt_rise = rise_resistance * net_cap
Dt_rise = 3.4 * 3.08
Dt_rise = 10.472
```

To compute the total cell delay,

```
rise_cell_delay = De + Di_rise + Dt_rise
rise_cell_delay = 0.012712 + 0.34 + 10.472
rise_cell_delay = 10.8247
```

The following is the full timing report:

```
Design           Wire Loading Model      Library
------------------------------------------------
TWO_INV                MEDIUM            cmos2_c


  Startpoint: I1 (input port)
  Endpoint: O1 (output port)
  Path Group: (none)
  Path Type: max

  Point    Incr    Path
------------------------------------------------
  input external delay         0.00      0.00 r
```

```
I1 (in)                         0.00       0.00 r
u1/A (INVERTER)                 0.00       0.00 r
u1/X (INVERTER)                 0.20       0.20 f
u2/B (NAND2)                    0.00       0.21 f
u2/X (NAND2)                   10.82      11.03 r
O1 (out)                        0.07      11.11 r
data arrival time                         11.11
```

Nonlinear Delay Model

The nonlinear delay model stores vendor-specific delay information in the technology library in the form of lookup tables. This model supports a close correlation between nonlinear vendor delay models and the timing analyzer calculations.

Delay analysis involves calculating total delay, which comprises cell and connect delay.

$D_{total} = D_{cell} + D_c$

D_{cell}

The delay contributed by the gate itself, typically measured from the 50 percent input pin voltage to the 50 percent output pin voltage.

D_c

The connect delay. It is either calculated by using the tree_type attribute in the operating_conditions group and the selected wire_load model, or read in from an SDF file as in the standard delay equation. Connect delay is calculated by the same method used in other delay equations.

The CMOS nonlinear timing model supports two methods of computing D_{cell}. Although you can mix the two methods in a technology library, you typically specify only the method that best correlates to the characterized library data.

The timing analyzer can compute D_{cell} directly by performing table lookup and interpolation in a cell delay table provided in the library or it can compute D_{cell} by using the propagation and transition tables:

$$D_{cell} = D_{propagation} + D_{transition}$$

$D_{propagation}$

A typical measurement for $D_{propagation}$ is the time from the 50 percent input pin voltage until the gate output just begins to switch, for example the 10 percent output voltage. Thus, when a $D_{transition}$ value defined from the 10 percent to 50 percent output voltage is added to $D_{propagation}$, the result is a 50 percent input to 50 percent output cell delay.

$D_{transition}$

The time required for the output pin to change state. This is sometimes referred to as the output ramp time.

$D_{transition}$ is the time between two reference voltage levels on the output pin. These levels can be 20 percent to 80 percent or 10 percent to 50 percent, for example. $D_{transition}$ is computed by performing table lookup and interpolation.

If cell delay tables are provided for a timing arc, the total delay equation used is

$D_{total} = D_{cell} + D_{c}$

If propagation delay tables are provided instead, the total delay equation becomes

$D_{total} = D_{propagation} + D_{transition} + D_{c}$

When transition tables in the library are indexed by input pin transition, the transition or slew of a gate’s input pin affects the transition of an output pin. As cells are being swapped and evaluated during optimization, this input or output transition effect can cause delay changes to ripple outside of the cells local to the change. Early in the design flow, you might not need to account for this ripple effect.

Design Compiler offers a variable switch to enable approximate delays. This variable is `compile_use_low_timing_effort`. When this variable is true, optimization uses approximate delays, ignoring the effect of input transition on output transition. This low-timing-effort switch does not affect timing or constraint reports. These reports are always generated using accurate timing.

Library Cell Timing Arcs

The following tables are defined for each library cell delay timing arc:

- Rise propagation
- Cell rise
- Fall propagation
- Cell fall
- Rise transition
- Fall transition

Note:

> Every delay arc can have propagation tables or cell tables, but not both. Also, every delay arc must have transition tables.

Each delay table is indexed by one through three of the following six variables:

- `input_net_transition`
- `output_net_length`
- `total_output_net_capacitance`
- `related_out_total_output_net_capacitance`
- `output_net_pin_cap`
- `output_net_wire_cap`

The values stored in the table are the propagation or transition time value. Interpolation is used to determine values between points. Breakpoints in the table can be arbitrarily defined. Each table maintains its own breakpoints, independent of other tables. Tables of different dimensions, breakpoints, and index variables can be intermixed within one technology library.

The following three tables are defined for each library-cell-constraint timing arc, as with setup and hold:

- rise_constraint
- fall_constraint

Each constraint table can be indexed by one through three of the following two variables:

- Constrained pin transition time
- Related pin transition time

This time the values stored in the table are the constraint values. The constraint tables allow setup and hold values to vary depending on the transition at the D-pin (constrained pin) and the clock pin (related pin) of a latch.

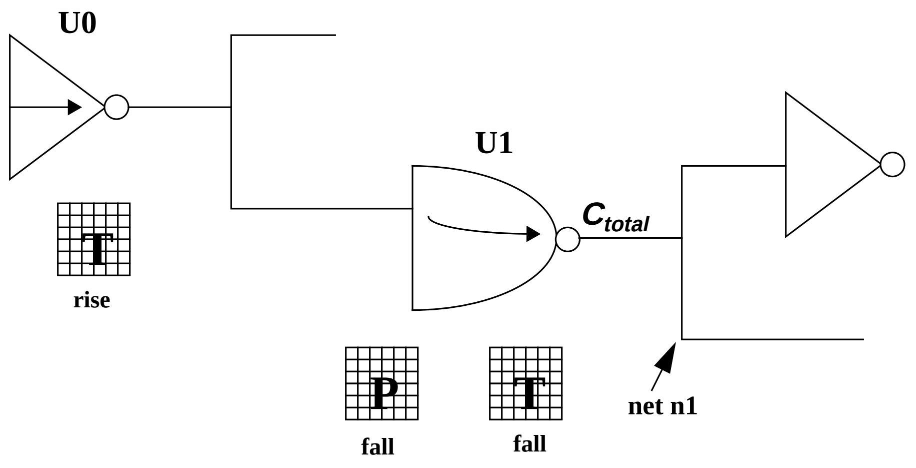
Because much of the information stored in a table is common to other tables, table templates and inheritance are supported. See the *Library Compiler Reference Manual, Volumes 1 and 2*, for more details.

To use the nonlinear delay model, activate a technology library that specifies the nonlinear delay model.

Delay Calculation Example

Figure A-6 illustrates the process for determining the fall delay across a timing arc of cell U1.

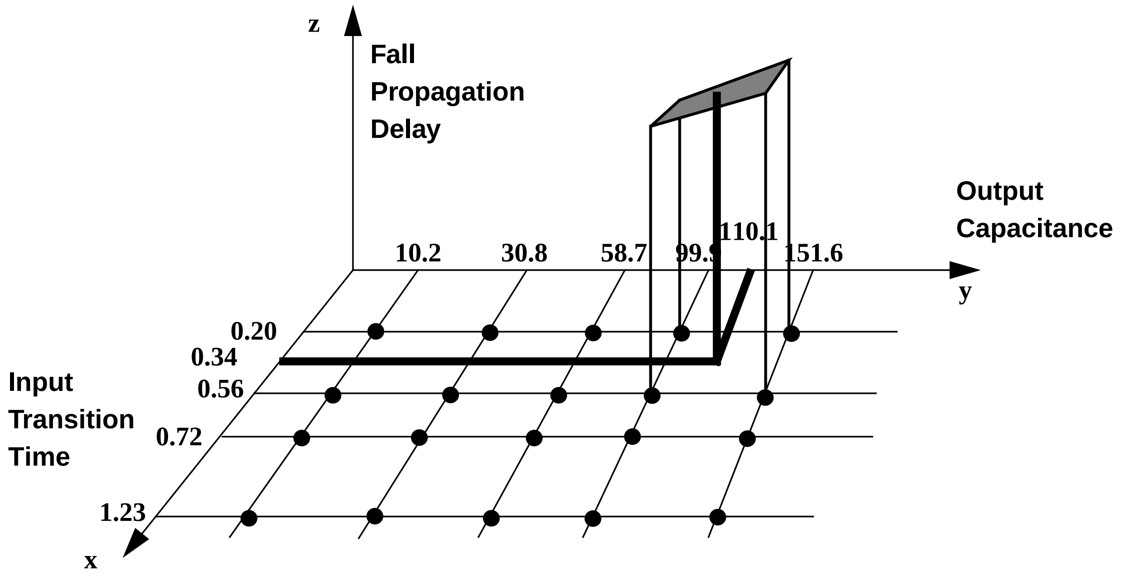
Figure A-6   Nonlinear Delay Calculation Example



Examine the fall propagation table: a two-dimensional table indexed by total output capacitance and input transition time. The total output capacitance is the sum of pin and wire capacitance on net n1 (Ctotal in Figure A-6). Input transition time is determined by evaluation of the transition time at the previous gate U0. Because the arc under evaluation in U1 is negative unate, the rise transition table at U0 is evaluated to determine the U1 input transition time. Assume that Ctotal is 110.1 and the input transition time is 0.34. Use these two values to index into the U1 fall propagation table.

Figure A-7 shows two-dimensional interpolation.

Figure A-7 Two-Dimensional Interpolation



In Figure A-7, the black dots represent points defined in the table. The four points with heights shown in the Z-axis are the neighboring points chosen for interpolation. The shaded area is the surface used for interpolation.

The fall transition time is computed by evaluation of the fall transition table. In this case, the table is a one-dimensional table based on total output capacitance. The output capacitance value (110.1) was calculated previously. Through simple linear interpolation within the table, the fall transition time is determined.

The fall propagation delay is then added to the fall transition time to compute the cell delay for a falling transition across the given timing arc of cell U1.

If a cell delay table is defined in the technology library for U1 instead of a propagation table, the table evaluations occur in the same way, but the transition result is not included when calculating the cell delay for U1.

Environmental Scaling

When calculating total delay, the timing analyzer individually scales each scalable parameter of Dtotal. Each scalable component of the total delay has its own global parameters to model the effects of variation in process, temperature, and voltage on the nominal case.

The following scaling factor is applied to individual components of the delay equation:

$$(1 + \Delta_V K_V)\ (1 + \Delta_T K_T)\ (1 + \Delta_P K_P)$$

Δ_V = Change in voltage from library-specified nominal value.

K_V = Scale factor for change in voltage.

Δ_T = Change in temperature from library-specified nominal value.

K_T = Scale factor for change in temperature.

Δ_P = Change in process from library-specified nominal value.

K_P = Scale factor for change in process.

Each component of the delay equation can have different values for K_V, K_T, and K_P, allowing it to be scaled independently of the other components. See the *Library Compiler Reference Manual, Volumes 1 and 2*, for more details on environmental scaling.

B

Equation Design Format

The logic equation (.eqn) design format is a way of specifying combinational logic in the form of Boolean equations. The equation format is technology-independent and designed to accept many common Boolean formats.

To use the equation design format successfully, you need to understand

- Equation format
- Lexical conventions
- Warning and error messages

You can use a combination of formats to describe your design, but you cannot mix formats in one design file. Read each part of a mixed-format design into Design Compiler separately. You can then combine designs by using the `link` command (see *Design Compiler User Guide*).

A 3-input AND gate in equation format follows:

```
.design example_and
.inputnames A B C
.outputnames Z

Z = A & B & C ;
```

Reading and Writing Equation Design Files

Equation design format files have the suffix .eqn.

Reading Equation Design Files

Use the `read` command `-format equation` option to read in an equation design description file.

Designs read in equation format can use buses.

```
dc_shell> read -format equation my_design.eqn
Loading equation file ’/designs/my_design.eqn’
Current design is now ’/designs/DESIGN.db:DESIGN’
{DESIGN}
```

Writing Equation Design Files

Use the `write` command `-format equation` option to write an equation design description file.

```
dc_shell> write -format equation -output ex.eqn DESIGN
```

To write out a design in equation format, the design must consist solely of combinational logic, either mapped or unmapped. To separate combinational logic from sequential elements, use the `group -logic` command.

For example, if you try to write out a design containing sequential elements (such as flip-flops or latches), you get the following message:

```
dc_shell> write -format equation -output ex.eqn DESIGN
Error: Cells at the current level of the design are not
       combinational. (PLAO-1)
Error: Write command failed. (UID-25)
0
```

To write the combinational logic portion of DESIGN, use `group -logic`; then write the newly created design. For example,

```
dc_shell> group -logic -design_name LOGIC
Performing group on cell ’U50’.
Performing group on cell ’U51’.
...
Performing group on cell ’U99’.
dc_shell> write -format equation -output LOGIC.eqn LOGIC
1
```

Three dc_shell variables (in the io variable group) control how equation-format design descriptions are written out:

```
equationout_and_sign
equationout_or_sign
```

Equation Format

Each equation format design must be in a separate file. Every equation file consists of a header and a body.

Equation Header

The header specifies the name of the design and its input and output ports.

An equation header consists of the following three required statements:

.design name

Defines the name of the design.

`.inputnames` *`input_name1 input_name2 .. input_nameN`*

Defines the names of the design's input ports (input signal names). There can be as many `.inputnames` commands as necessary for naming all input ports.

.outputnames *output_name1 output_name2 .. output_nameN*

Defines the names of the design's output ports (output signal names). There can be as many `.outputnames` commands as necessary for naming all output ports.

The order in which input and output ports are specified can be significant when instantiating the equation-format design within a hierarchy. Many netlist formats make order-based connections to a subdesign. That is, the first wire is connected to the first port, the second wire is connected to the second port, and so on. The ordering of ports in inputnames and outputnames statements determines the ordering of ports on the equation-format design.

Equation Body

The body of the equation defines the Boolean equations representing the design.

The body of the equation description consists of a set of Boolean equations. Each variable in an equation represents a single-bit signal in a design.

Variables whose names are declared in the .inputnames and .outputnames statements represent signals tied to design ports. Other variable names represent internal signals.

Each equation represents a single-bit output combinational component.

Example

This example describes a 2-bit adder.

```
.design example_adder
.inputnames a0 a1 b0 b1
.outputnames s0 s1
.outputnames carry_out       # another output

s0 = a0 b0’ + a0’ b0;
c0 = a0 b0;                  # c0 is an internal signal
s1 = a1 ^ b1 ^ c0;

carry_out = a1 b1 +
c0 ( a1 + b1 );  # multi-line equation
```

An equation can span several lines, as shown by the previous carry_out equation, and must be terminated by a semicolon (;).

The construction a0 b0 (a0 space b0) means a0 and b0, as described by Table B-1.

Equation Syntax

The general format of an equation is

`output_name = boolean_expression ;`

`output_name`

Is the name of a single port or signal.

`boolean_expression`

Is composed of port and variable (signal) names combined with operators.

Several kinds of operators are available for different styles of equations. Equation operators are summarized in Table B-1.

Table B-1 Equation Operators

Character	**Operation**	**Precedence**	**Example**
zero (0)	constant zero	5	0
one (1)	constant one	5	1
apostrophe (’)	invert	4	signal’
exclamation (!)	invert	4	!signal
caret (^)	xor	3	s1 ^ s2
ampersand (&)	and	2	s1 & s2

Table B-1 Equation Operators (continued)

Character	Operation	Precedence	Example
asterisk (*)	and	2	s1 * s2
space ( )	and	2	s1 s2
plus (+)	or	1	s1 + s2
vertical bar (|)	or	1	s1 | s2

You can override operator precedence by using parentheses. For example,

```
a + b * c      # a or (b and c)
(a + b) * c    # (a or b) and c
```

Of the two inversion operators, the exclamation point (!) precedes its argument and the apostrophe (') follows its argument. Arguments can be variables, ports, or expressions.

When two variables, ports, or expressions are separated by a space, they are logically ANDed together. For example, (a b) is equivalent to (a & b).

Because several operators might be available for each Boolean operation, you can write a logic description in several styles.

Example

The following are three equations (out1, out2, and out3), each of which describes a 4-input multiplexer:

```
.design example_multiplexer

.inputnames a b c d            # 4 input bits
.inputnames s1 s2              # 2 select bits
.outputnames out1 out2 out3    # 1 multiplexer output bit
```

```
out1 = a s1 s2 + b s1 s2′ + c s1′ s2 + d s1′ s2′ ;

out2 = (a & s1 & s2) | (b & s1 & !s2) |
       (c & !s1 & s2) | (d & !s1 & !s2) ;

out3 = s1 * ( a * s2 | b * s2′) |
       s1′ * ( c * s2 | d * s2′) ;
```

Specifying Don’t Care Conditions

The equation format allows for the specification of don’t care conditions. A don’t care condition is one in which an output variable can be assigned any value. During optimization, Design Compiler can use this don’t care information to simplify a design more effectively.

The don’t care condition for a particular output is specified by the don’t care specifier before that output’s equation. The specifier is a question mark (?).

For example, in a 3-input multiplexer, the two select bits (s1 and s2) will never both be 1. Specify the don’t care output for this condition as follows:

```
output = a s1′ s2′ + b s1′ s2 + c s1 s2′ ;

?output = s1 s2 ; # don’t care if s1 and s2 are both 1
```

Lexical Conventions

An equation description consists of operators, variables, commands, and comments. Each part is described in the following sections.

Operators

Table B-2 lists the operator symbols and meanings.

Table B-2   Equation Lexical Operators

Symbol	Meaning
#	Single-line comment character
/* */	Multiple-line comment delimiter characters
" \	Quotation characters
?	Don’t care specifier
=	Assignment operator
( )	Logic-grouping operators
;	Statement terminator
0 1	False and true constant values
! ’	Inversion operators
& *	AND operators
| +	OR operators
^	XOR operator

Three variables (in the io variable group) control how the equations are written out for equation-format files. The variables are

equationout_and_sign

Specifies the character used to represent AND, which is an asterisk (*) by default. For example,

```
and_variable = a*b;
```

equationout_or_sign

Specifies the character used to represent OR, which is a plus sign (+) by default. For example,

```
or_variable = a+b;
```

equationout_postfix_negation

If true (default), negated expressions are written out with a single quotation mark (') after the expression. For example,

```
not_variable = (expression)’;
```

If false, negated expressions are written out with an exclamation mark (!) before the expression. For example,

```
not_variable =!(expression);
```

Variables

Variables specify signals within a set of equations, representing wires in the equivalent netlist. The rules below apply to naming variables.

- The first character must be a letter.
- Variable names are case-sensitive.
- Enclose special characters (listed in Table B-2) in double quotation marks ("").

- Precede any quotation marks by a backslash (\), and enclose the entire variable name in double quotation marks.
- Precede any backslash by another backslash, and enclose the entire variable name in double quotation marks.

Example Variable Names

```
signal
reset
bus_23
net1
"1st_pass"
"v(20)"
"quote\"contained"
"backslash\\contained"
```

Commands

Command keywords start with a dot (.). You must put them at the beginning of a line.

Comments

You can make comments a single line or multiple lines.

Single-line comments start with a pound sign (#) and do not extend beyond the current line. There is no concluding delimiter for a single-line comment.

Multiple-line comments start with a slash-asterisk (/*) and continue until an asterisk-slash (*/) is encountered. The comment contains all the characters between these delimiters.

Error and Warning Messages

If Design Compiler detects an error while reading an equation-format file, it displays one of the following warning or error messages with the line number on which the error was found:

Error: Can’t read file

The specified file is not an equation-format file.

Error: Design name must be specified

The file contains no design statement.

Error: Illegal function

A syntax error was encountered.

Error: Illegal left-hand side

The left side of a equation is not a simple variable.

Error: Signal set a second time

A signal appears on the left side of two equations.

Error: Port declared twice

A port name appears twice in an inputnames or outputnames list.

Error: Invalid use of '?'

A ? (don’t care) was encountered on the right side of an equation or in a command.

Warning: Unknown command ".xxx" ignored

An unrecognized command (a line beginning with a dot) was encountered.

C

PLA Design Format

The programmable logic array (PLA) design format is a way of specifying combinational logic as a truth table. The PLA format is technology-independent and can accept several different styles of PLA descriptions, including ones that allow function and don’t care conditions.

To use the PLA format successfully, you need to understand

- PLA format
- Lexical conventions
- Error and warning messages

The Synopsys VHDL Compiler Reference Manual and the HDL Compiler for Verilog Reference Manual describe the HDL input formats.

Netlist formats are defined by their manufacturers.

Design Compiler reads hierarchical, sequential, and combinational circuit descriptions in a variety of input formats. These formats include logic equations, programmable logic array (PLA) truth tables, finite state machine state tables, hardware description languages (HDLs), and netlists.

You can use a combination of formats to describe your design, but you cannot mix formats in one design file. Each part of a mixed-format design must be read into Design Compiler separately. You can then combine designs by using the `link` command (see *Design Compiler User Guide*).

Example

This example shows a 3-input AND gate in PLA format.

```
.design example_and
.inputnames A B C
.outputnames Z

# in   out
 000    0
 001    0
 010    0
 011    0
 100    0
 101    0
 110    0
 111    1
```

Reading and Writing PLA Design Files

Use the `read` command `-format pla` option to read in a PLA design description file.

```
dc_shell> read -format pla my_design.pla
```

Use the `write` command `-format pla` option to write a PLA design description file.

```
dc_shell> set_structure false
dc_shell> set_flatten true
dc_shell> compile -no_map
dc_shell> write -format pla -output example.pla
```

PLA Format

Each PLA-format design must be in a separate file. Each PLA file consists of a header and a body. The header specifies the name of the design and its input and output ports, along with any options directing how the PLA is to be interpreted. The body defines the Boolean behavior of the design in the form of a single truth table.

PLA Header

A PLA header must contain the following three statements:

.design name

Name of the design. If not specified, the design name is the name of the design description file.

.inputnames *name1 [name_2 ...]*

Names of the design’s input ports (input signal names). There can be as many .inputnames commands as needed to name all input ports.

.outputnames *name_1 [name_2 ...]*

Names of the design’s output ports (output signal names). There can be as many .outputnames commands as needed to name all output ports.

A PLA header can also have one or more of the following three optional statements:

.type pla_type

The type of PLA, as described in “PLA Types” later in this appendix.

.phase phase_string

The phase of the PLA output ports, as described in “Output Phases” later in this appendix.

.ordered_percent

The ordering of rows using the percent (%) character is significant, as described in “Ordered Default Values” later in this appendix.

Port Order

The order in which input and output ports are specified is significant, for the following two reasons:

- Each column in a truth table is associated with either an input port or an output port. If there are i inputs and o outputs specified, the truth table is expected to contain i + o columns of characters. The first i columns are associated with the input ports in their specified order. The remaining o columns are associated with the output ports, in their specified order.

- When PLA-format designs are instantiated within a hierarchy, many netlist formats make order-based connections to the new subdesign. That is, the first wire is connected to the first port, the second wire is connected to the second port, and so on. The ordering of ports in .inputnames and .outputnames statements determines the ordering of ports on the PLA-format design.

PLA Body

The body of a PLA description consists of a single truth table. Each line (row) of the truth table specifies a set of inputs (the input plane) and the set of outputs to be generated (the output plane). Each column in a truth table represents one single-bit input or output port value.

Example

This example shows a truth table for a 2-bit adder.

```
# PLA for a 2-bit adder
.design test_adder
.inputnames a0 a1 b0 b1
.outputnames s0 s1
.outputnames carry_out      # more outputs

# in plane  out plane
  1-0-      100
  0-1-      100
  00-1      010
  -100      010
  01-0      010
  -001      010
  1010      010
  1111      010
  111-      001
  1-11      001
  -1-1      001
```

The truth table can also be viewed as a two-level AND-OR logic structure, where the input plane is the AND plane and the output plane is the OR plane. Each row of a PLA truth table specifies a product term of the function you want.

Input Plane

The input plane for each product term specifies an AND function of the input ports, which can affect certain output ports. An input column can contain a 0 or a 1, which specifies that the corresponding input port is used in that row’s (product’s) AND function, or a minus sign (–), which specifies that input port is not used in that row’s AND function.

For example, the first line of the truth table specifies its AND function, using only the first input port and the complement of the third input port (that is, a0 & b0’):

```
1-0-    100
```

Output Plane

The output plane for a product term determines how the specified AND function affects those outputs. The effect of a product term on outputs depends on the PLA type, as described in “PLA Types” later in this appendix. Types are summarized in Table C-1 later in this appendix.

Default Values

The percent (%) character is allowed in the input (AND) plane of a PLA truth table, meaning either 0 or 1. The % character is used in a product term to define a default output value for a set of otherwise unspecified input conditions.

Example

This example shows how you can explicitly define default values using numbers.

```
.design example
.inputnames a b c
.outputnames f0 f1
.type fr

# Meaningful conditions
001 01
010 10
100 11

# Default values
000 00
011 00
101 00
110 00
111 00
```

The function specifies that for input conditions 001, 010, or 100, the output values are 01, 10, or 11, respectively. For all other input conditions, the output values are 00.

Example

This example shows how you can define the default values by using the % character.

```
.design example
.inputnames a b c
.outputnames f0 f1
.type fr

# Meaningful conditions
001 01
010 10
100 11

# Default values
%%% 00
```

The % character can be used with the symbols 0 and 1 in a row's input plane. You can specify any number of product terms with the % character, and the order of rows within the PLA is irrelevant unless you use the `.ordered_percent` command (see the following section).

Ordered Default Values

By default, the percent (%) character in an input plane is expanded to 0 or 1 and both rows are added to the PLA truth table. All rows of a PLA truth table are considered together, in no particular order.

If the `.ordered_percent` command is used in a PLA header, the ordering of rows with the % character is significant. In this context, the % character is interpreted as any unspecified values of {0, 1, or %} in this position. If a truth table is unambiguous, any following product terms using the % character are also unambiguous.

In a truth table using `.ordered_percent`, rows not containing the % character are combined into a base table. Each row containing the % character is evaluated in order against the base table.

Example

This example shows a PLA description that is ambiguous without the `.ordered_percent` command.

```
.design example
.inputnames a b c
.outputnames f0 f1
.ordered_percent
.type fr

# Meaningful conditions
001 01
010 10

# First default condition
11% 11
# Second default condition
%%% 00
```

If the `.ordered_percent` command is not used, the ambiguous (conflicting) input values for the two default conditions are 111. With ordered default values, the first default condition is for input values 111 and the second default condition is for input values 011 and 101.

PLA Fields

Normally, each row in a PLA must contain all inputs and outputs. For large PLAs, this can be cumbersome when some rows use only a small number of inputs or outputs. The .field statement specifies a subset of inputs and outputs to be used for the following PLA rows.

The format of the .field statement is

```
.field name_1 [ name_2 ... ]
```

name_1, name_2, and so forth are the names of input and output signals that are specified in the columns of the PLA immediately following.

In the .field specification, specify input ports before output ports. If the field specification is too long to fit on one line, you can use multiple.field statements.

Example

The following PLA descriptions are equivalent. The second example uses the .field statement.

```
.design example
.inputnames a b c d e f
.outputnames x y

11----   00
--11--   01
----11   10

.design example
.inputnames  a b c d e f
.outputnames x y

.field a b x y
11   00

.field c d x y
11   01

.field e f x y
11   10
```

Any input or output not included in the .field statement is considered irrelevant for the rows following. If the omitted inputs or outputs are meaningful for other rows, they must be specified prior to their use.

A .field statement remains in effect until the next .field statement. To return to the full PLA description, use a .field statement specifying all input and output names.

It is often convenient to specify the default values for an output as the first row of the PLA.

Example

The first row in this example specifies that the first two outputs have a default value of 0 for all unspecified conditions. The last two outputs have a default value of 1 for all unspecified conditions.

```
.design example
.inputnames a b
.outputnames f0 f1 f2 f3

%%  0011

.field a b f0 f1
11  00

.field a b f2 f3
11  00
```

PLA Types

The interpretation of a PLA output plane depends on the type of the PLA. A PLA type determines the interpretation of its truth table. There are three categories of PLA types: f, d, and r. If a PLA type is not specified, the default is fd. See “.type fd PLA” later in this appendix.

PLA types use the concepts of on sets, off sets, and don’t care sets.

on set

The conditions under which a particular output of the PLA is 1. Type f PLAs specify on set.

off set

The conditions under which a particular output of the PLA is 0. Type r PLAs specify off set.

don’t care set

The conditions under which an output is assigned any value. The optimization process in Design Compiler can use the specified don’t care information to more effectively simplify a design. Type d PLAs specify their don’t care set.

The .type Command

A PLA type is specified by a `.type` command in the PLA header. The `.type` command is followed by a short keyword that specifies the PLA type: f, fd, fr, r, or dr. Types f, fd, and fr are described in the following sections.

.type f PLA

The simplest PLA type is .type f. In these PLAs, only the on set is specified. If a 1 appears in the output plane part of a product term, the corresponding output is set to 1 whenever the AND function of the input plane part evaluates to 1. If any other character (0 or ~) appears in the output plane, the corresponding output is not affected by the product term.

Example

These examples show a function in equation format and the same function as a .type f PLA.

```
y = a b’ + c
z = c
.design example
.inputnames a b c
.outputnames y z
.type f
10-  10
--1  11
```

The first product term specifies that output y is 1 when a is 1 and b is 0. The second product term specifies that output y is also 1 when c is 1 and that output z is 1 when c is 1.

In .type f PLAs, the off set of the outputs is implicitly declared as the complement of the on set. The don’t care set is empty. To specify a don’t care set, use one of the PLA types that include type d, such as .type fd.

.type fd PLA

This is the default PLA type if no .type is specified.

A .type fd PLA is a .type f PLA with the addition of the don’t care set.

A minus sign (–) in the output plane part of the product term specifies a don’t care condition. The corresponding output can be any value when the AND of the input plane evaluates to 1.

Example

This example adds a don’t care condition to the previous equation format function. This specifies that y can take on any value when a and b are both 1 and can be described by the PLA that follows the function.

```
y = a b′ + c
?y = a b
z = c
.design example
.inputnames a b c
.outputnames y z
.type fd
10-  10
--1  11
11-  -0
```

In .type fd PLAs, the off set of the outputs is implicitly declared as the complement of the union of the on set with the don’t care set. That is, anything not specified is included in the off set. If the on set and the don’t care set overlap, the don’t care set has priority.

Zero and tilde characters (0 and ~) in the output plane mean that the product term does not affect the output.

.type fr PLA

A .type fr PLA explicitly specifies both the on set and the off set of a function. A 1 in the output plane part of a product term adds to the on set. The corresponding output is 1 when the AND function of the input plane part evaluates to 1. A 0 in the output plane part of a product term adds to the off set. The corresponding output is 0 when the AND function of the input plane part evaluates to 1.

The don't care set of outputs is implicitly declared as the complement of the union of both the on set and the off set. Anything not specified is included in the don't care set.

It is an error for the on set and the off set to overlap.

This example shows a PLA for the previous equation format function.

```
.design example
.inputnames a b c
.outputnames y z
.type fr
10-  1~
0-1  1~
0-0  0~
--1  ~1
--0  ~0
```

Minus (–) and tilde (~) characters in the output plane mean that the product term does not affect the output.

.type fdr PLA

The .type fdr PLAs are not supported. If a PLA is specified as fdr, it is treated as type fd.

Output Plane Characters With Different PLA Types

Table C-1 summarizes how characters are interpreted in the output plane for each PLA type.

Table C-1 Characters in the Output Plane

Type	1	0	–	~
f	on set			
fd	on set		dc-set	
fr	on set	off set		
r		off set		
dr		off set	dc-set	

Output Phases

Use the optional `.phase` command to specify that selected outputs are inverted (with respect to the given output plane). If the `.phase` command is omitted, all outputs are assumed to be noninverted.

The `.phase` command is followed by a set of 0s and 1s, one for each PLA output. Each character specifies the polarity of its corresponding output:

0

Indicates that the PLA actually specifies the complement of the output function.

1

Indicates that the output is not inverted.

Example

This example shows two PLAs that describe the same function.

```
.design example       .design example
.inputnames a b c     .inputnames a b c
.outputnames y z      .outputnames y z
.phase 01             1--  10
000  10               -1-  10
--1  01               --1  11
```

Note:

> When writing out a PLA-format design description file, Design Compiler inserts a `.phase` command if the `set_flatten -phase` command was used (see Chapter 4, “Controlling Logic-Level and Gate-Level Optimization”).

Espresso Format

Design Compiler accepts descriptions written for the PLA minimization program Espresso. Design Compiler uses the Espresso commands `.i`, `.o`, `.ilb`, `.ob`, `.type`, `.phase`, and `.e` but ignores other Espresso commands.

Lexical Conventions

A PLA description consists of commands, comments, and a single truth table.

Commands

PLA command keywords start with a dot (.) in the first column of a line.

.design name

The name of the design.

.inputnames *name1 [name_2 ...]*

The names of the design's input ports (input signal names). There can be several `.inputnames` commands if there are many input ports.

.outputnames *name1 [name_2 ...]*

The names of the design's output ports (output signal names). There can be several `.outputnames` commands if there are many output ports.

.phase phase_string

The phase of PLA output ports, as described in "Output Phases" in this appendix.

.type pla_type

The type of PLA, as previously described in "PLA Types" in this appendix.

.field *name_1 [name_2 ...]*

An ordered subset of the names of input and output signals that are specified in the columns of the PLA immediately following.

Comments

Comments begin with a pound sign (#) in the first column and continue to the end of the line. You can place comment lines anywhere in a PLA description.

Truth Tables

A truth table is a block of special characters that specify a PLA's function. Each row of characters in a truth table specifies part of the overall function. You can use spaces, tabs, and carriage returns anywhere in the truth table to improve readability. Table C-2 lists the valid characters in a truth table.

Table C-2 Valid Truth Table Characters

Character	**Interpretation**
zero (0)	noninverted
one (1)	inverted
minus (–)	don't care
tilde (~)	don't care

Error and Warning Messages

If an error is detected during the reading in of a PLA-format file, its line number and one of the following error or warning messages appears:

Error: Can’t read file

The specified file is not a PLA-format file.

Error: Design name must be specified

The file contains no .design statement.

Error: No product terms for PLA found in file.

The file contains no truth table.

Error: Invalid character

The file contains an invalid character.

Error: Illegal command

A command has been found inside the truth table.

Error: Ports incompletely specified

Not all .inputnames and .outputnames are included before the truth table.

Error: Mismatched number of inputs

The number of names in .inputnames does not match the number specified in the .i command.

Error: Mismatched number of outputs

The number of names in .outputnames does not match the number specified in the .o command.

Error: Port declared twice

A port name appears twice in an .inputnames or .outputnames list.

Error: Unknown port in .field statement

An invalid port name is specified in a .field command.

Error: Invalid order of input and output ports

An output port name appears before an input port name in a .field command.

Error: ON set and OFF set overlap

The on set and off set overlap in a .type fr PLA.

Error: Mismatched number of outputs and phase values

Positive phase is used for all outputs. The `.phase` command specifies too many or too few phase values for the number of output ports.

Warning: Unknown command ---- ignored

An unrecognized command (a line beginning with a dot) has been encountered.

Warning: Unknown PLA type

PLA types must be f, fd, fr, fd, r, or dr.

Warning: Unknown value for .phase keyword

Positive phase is used for all outputs. The `.phase` command specifies a character other than 0 or 1.

D

State Table Design Format

A state table represents the behavior of a finite state machine (FSM), by describing the conditions that change the internal state of the machine and the corresponding output values.

The state table format is an interface format for exchanging FSM descriptions with CAE tools.

To use the state table format successfully, you need to understand

- State table header
- State table body
- State table fields
- State encoding
- Don’t care sets

A state table description consists of a port specification header, a state table body, optional comments, and an optional state encoding. The state table format is an extension of the .type fr PLA input format.

The header names the design, the input and output port signals, and the clock and reset signals. The body defines, for each set of inputs, the state transitions and the resulting outputs. The optional state encoding specifies how you want the state bit vectors encoded.

State Table Description Example

This example shows a state table description for a simple soft drink vending machine that accepts nickels and dimes and dispenses soda and change.

```
# Soft drink machine -- Price is 15 cents
#
# Assumptions: Only one coin is deposited at a time
#              and slower than the clock for the machine
# Design name
.design soft_drink_machine
# Input port names
.inputnames clk reset nickel_in dime_in quarter_in
# Output port names
.outputnames nickel_out dime_out dispense
# Clock signal’s name and type
.clock clk rising_edge
# Reset signal’s name, type, and resulting state
 .asynchronous_reset reset rising IDLE
# State table body
100     IDLE       FIVE       000
010     IDLE       TEN        000
001     IDLE       IDLE       011
100     FIVE       TEN        000
010     FIVE       IDLE       001
001     FIVE       IDLE       111
100     TEN        IDLE       001
010     TEN        IDLE       101
001     TEN        OWE_DIME   011
000     OWE_DIME   IDLE       010
# Wait in current state until money is deposited
000     IDLE       &          000
000     FIVE       &          000
000     TEN        &          000
# Preferred state encoding
.encoding
IDLE      2#00
FIVE      2#01
```

```
TEN       2#11
OWE_DIME  2#10
```

The pound sign (#) is used for two purposes:

- Comment lines. If the pound sign is at the beginning of a line, the rest of that line is considered a comment and is ignored.
- Based literals (2#10). Based literals are described in“State Encoding” later in this appendix.

Reading and Writing State Table Design Files

To read in a state table design description file, use the `read` command `-format st` option:

```
dc_shell> read -format st example_c1.st
```

To write out a state table design description file after synthesis, you must first extract the state table information and then use `write -format st`:

```
dc_shell> set_structure false
dc_shell> compile -no_map
dc_shell> extract
dc_shell> write -format st -output example.st
```

For more information about extracting state tables, see Chapter 10, “Optimizing Finite State Machines.”

State Table Header

These statements define various aspects of the FSM design, such as the design's name, input and output port names, clock signal name and type, and reset behavior.

A state table header contains the following keyword statements:

```
.design  name
.inputnames  name_1  [ name_2 ... ]
.outputnames  name_1  [ name_2 ... ]
.clock  signal  sense
.asynchronous_reset  signal  sense  state
.synchronous_reset  signal  sense  state  output
.synchronous_reset  state  output
.ordered_percent
```

The .design Statement

The `.design` statement specifies the name of the design.

The format is

```
.design name
```

The .inputnames Statement

The `.inputnames` statement specifies the input ports' names. The order of the input names determines the port order of the design. If there are many input port names, multiple `.inputnames` statements can be used.

The format is

```
 .inputnames name_1 [ name_2 ... ]
```

name_1, name_2...

The names of input ports. Some of these input ports can be used as the .clock, .synchronous_reset, or .asynchronous_reset inputs.

The .outputnames Statement

The `.outputnames` statement specifies the output port names. The order of the output names determines the port order of the design. If there are many output port names, multiple `.outputnames` statements can be used. The format is

```
 .outputnames name_1 [ name_2 ... ]
```

name_1 name_2...

The names of output ports.

The .clock Statement

The `.clock` statement specifies the name of the input signal used for the clock, and the clock sense (rising or falling edge). The clock signal is connected to the clock pin on the state vector flip-flops. The design is required to have a single clock. The format is

```
.clock signal sense
```

`signal`

Name of an input port. This signal is omitted from the state table body.

`sense`

Clock sense; either rising_edge or falling_edge.

`rising_edge`

Indicates that the flip-flops are activated when the clock signal changes from low to high.

`falling_edge`

Indicates that the flip-flops are activated when the clock signal changes from high to low.

The .asynchronous_reset Statement

An asynchronous reset is implemented by choosing a flip-flop from the library with an asynchronous set or reset signal and connecting the reset signal appropriately to force the machine into the reset state.

The `.asynchronous_reset` statement specifies the name of the asynchronous reset signal, its sense (rising or falling), and the reset state. The format is

```
.asynchronous_reset signal sense state
.async_reset        signal sense state
```

signal

Name of an input port. This signal is omitted from the state table body.

`sense`

Timing sense, either rising or falling. If rising, the reset is active high. If falling, the reset is active low.

`state`

Name of the reset state— that is, the state that results when the asynchronous reset signal is asserted.

The .synchronous_reset Statement

A synchronous reset is implemented by use of additional combinational logic in the synchronous part of the FSM. The synchronous reset behavior of a machine can also be specified with the plus (+) operator as described in the “State Table Body” section later in this appendix.

Two types of synchronous reset are supported, depending on whether the reset is caused by a signal or by entry into an invalid state.

Synchronous Reset From Signal

This type of synchronous reset specifies that if the reset signal is asserted, a reset occurs.

This type of `.synchronous_reset` statement specifies the name of the synchronous reset signal, its sense (rising or falling), the reset state, and the reset state’s output values. The format is

```
.synchronous_reset signal sense state output
.sync_reset         signal sense state output
```

signal

Name of an input port. This signal is omitted from the state table body.

`sense`

Timing sense, either rising or falling. If rising, the reset is active high. If falling, the reset is active low.

`state`

Name of the reset state— that is, the state that results when the synchronous reset signal is asserted.

`output`

Output values for the synchronous reset state. Valid output values are described in the “State Table Body” section.

Synchronous Reset From Invalid State

This type of synchronous reset specifies that if an invalid state is entered, a reset occurs.

This type of `.synchronous_reset` statement specifies the reset state and the reset state’s output values. The format is

```
.synchronous_reset state output
.sync_reset         state output
```

`state`

Name of the reset state— that is, the state that results when an invalid state is entered. Invalid states are specified in a state table body with the plus (+) character, as described in the next section.

`output`

Output values for the synchronous reset state. Valid output values are described in the state table body.

The .ordered_percent Statement

The optional `.ordered_percent` statement specifies that the ordering of rows containing the percent (%) character is significant. See “Primary Input Column” later in this appendix.

```
.ordered_percent
```

State Table Body

The state table body for an FSM is analogous to a PLA description for combinational logic.

This example shows only the state table from the previous example.

```
# State table body
100     IDLE       FIVE       000
010     IDLE       TEN        000
001     IDLE       IDLE       011
100     FIVE       TEN        000
010     FIVE       IDLE       001
001     FIVE       IDLE       111
100     TEN        IDLE       001
010     TEN        IDLE       101
001     TEN        OWE_DIME   011
000     OWE_DIME   IDLE       010

# Wait in current state until money is deposited
000     IDLE       &          000
000     FIVE       &          000
000     TEN        &          000
```

Each row of a state table is arranged into four columns: primary inputs, present state, next state, and primary outputs.

Following the PLA analogy, the primary inputs and the present state are inputs to the design and the next state and the primary outputs are outputs of the design. Each row in the state table is interpreted as follows: If the machine is in the current state and all primary inputs have the specified values, the specified outputs are asserted and the machine transitions to the specified next state on the next clock cycle.

State Table Format

Table D-1 lists the valid characters for each column in the state table.

Table D-1 Valid Characters for State Table

Primary Inputs	Present State	Next State	Primary Outputs
0	*	&	0
1	+	–	1
–	name	~	–
%		name	~

The next four sections describe the meaning of each column and its characters.

Primary Input Column

The primary input section of a state table row is exactly the same as that of a PLA. Each column represents one input port, in the order specified by the `.inputnames` statements. The clock and reset signals are omitted from the state table body, because their functions are explicitly described.

The valid characters for the primary input columns are 0, 1, % (percent), and – (minus):

0

The signal must be logical 0.

1

The signal must be logical 1.

% (percent)

Specifies defaults for otherwise unspecified values of 0 and 1 for this signal. An FSM is often described only in terms of the input conditions that lead to states of interest. The % character is used to specify default conditions for the remaining set (0, 1, or both) of this column’s input signal’s conditions. See “Specifying Default Input Conditions By Using the % Character” on page D-12 for more information.

If the `.ordered_percent` command is used in the state table header, the ordering of rows containing the % character is significant. See “Ordered Default Input Conditions” later in this appendix.

–(minus)

Don’t care; that is, the signal can be either logical 0 or 1.

Specifying Default Input Conditions By Using the % Character

This example shows an FSM description that uses %.

```
.design example
.inputnames a b c d clk
.outputnames f0 f1
.clock clk rising_edge
```

```
1001 S0  S1 01
1010 S0  S2 10
1100 S0  S3 11
0--- S0  S4 11

# Default action
1000 S0  S0 00
1011 S0  S0 00
1101 S0  S0 00
1110 S0  S0 00
1111 S0  S0 00
```

If the FSM is in state S0 and input conditions 1001, 1010, or 1100 occur, the machine transitions to states S1, S2, or S3 and asserts output values 01, 10, or 11, respectively. Otherwise, the machine stays in the same state and asserts output values 00.

Example

This example uses the % character to specify that the machine is to stay in the same state and assert output values 00 for unspecified combinations of 0 and 1 when in present state S0.

```
.design example
.inputnames a b c clk
.outputnames f0 f1
.clock clk rising_edge

1001 S0  S1  01
1010 S0  S2  10
1100 S0  S3  11
0--- S0  S4  11

# Default action
1%%% S0  S0 00
```

The % character can appear in the same row with the symbols 0 and 1. You can specify any number of rows with the % character, and the order of these rows within the state table is irrelevant (unless `.ordered_percent` is used, as described in the following section).

Ordered Default Input Conditions

All rows of the state table that do not contain the percent operator are combined into a base table. Normally, all rows containing the percent operator are applied against the base table at once.

When you use the `.ordered_percent` command, each row containing the percent operator is applied against the base table in the order in which the row appears. If two rows containing the percent operator imply different conditions, the first row takes precedence.

If there are rows containing an ordered percent operator that do not add unique conditions to the state table, the following warning message appears:

```
Ignoring row at line 19 since previous row(s) containing the
percent operator cover all input conditions implied by this
row.
```

The following examples describe a simple FSM with unordered default conditions, then show how to add ordered default conditions.

Example

This example describes an FSM that detects the sequence 1 0 1 on the input signal serial. When this sequence is found, the output signal found_seq is set to 1. If two zeros or two ones are found, the FSM returns to state INITIAL and the output signal found_seq is set to 0. If the input signal reset is 1, the FSM also resets to state INITIAL and output value 0.

```
.design seq_detect
.inputnames serial reset clk
.outputnames found_seq
.clock clk rising_edge10 INITIAL SEEN_1  0

00 INITIAL INITIAL 0

00 SEEN_1  SEEN_0  0
10 SEEN_1  INITIAL 0

00 SEEN_0  INITIAL 0
10 SEEN_0  INITIAL 1

-1 INITIAL INITIAL 0
```

Only one row exists where the next state is INITIAL and the output value is 1.

Example

This example sequence detector with defaults shows how an unordered percent operator is used to represent all transitions to next state INITIAL with an output value of 0.

```
.design seq_detect
.inputnames serial reset clk
.outputnames found_seq
.clock clk rising_edge

10 INITIAL SEEN_1  0
00 SEEN_1  SEEN_0  0
10 SEEN_0  INITIAL 1
%% *       INITIAL 0
```

Example

This example sequence detector with ordered defaults shows how using the order-dependent percent operator adds an input signal stall to the FSM. If stall is 1, the FSM stays in the current state and sets the output to 0. The reset signal has precedence over the stall signal.

```
.design seq_detect
.inputnames serial stall reset clk
.outputnames found_seq
.clock clk rising_edge
.ordered_percent

100 INITIAL SEEN_1  0
000 SEEN_1  SEEN_0  0
100 SEEN_0  INITIAL 1

# If reset=1, then reset
#    machine
%%1 *       INITIAL 0

# Else if stall=1,
#    then stay in same state
%1% *       &       0

# Else, must have invalid
# sequence, so reset machine
%%% *       INITIAL 0
```

Present State Column

The present state column of the state table contains a symbolic state name or one of the special symbols * and +, which match any valid state, and any invalid state, respectively.

name

Any state name.

* (asterisk)

  In the present state column, matches any valid state in the machine. This row of the state table takes effect in all the valid states.

+ (plus)

  In the present state column, specifies the behavior of the FSM when the machine is in an invalid present state. For example, consider an FSM with three states encoded as 2#00, 2#01, and 2#10. By default, the behavior of the machine is undefined when the present state is 2#11. This don't care condition is used to optimize the design. However, sometimes it is necessary to describe what happens in the machine if the present state has the unassigned (invalid) value of 2#11.

The synchronous_reset keyword described earlier is shorthand for a particular style of using the + symbol. This is useful when specifying invalid input state behavior.

Example

This example uses the + symbol to force the machine to state S0 when the signal reset is asserted.

```
.design example
.inputnames reset a clk
.outputnames found error
.clock clk rising_edge

00 S0 S0 00
01 S0 S1 01
00 S1 S0 10
01 S1 S2 00
00 S2 S0 00
01 S2 S1 10

# Reset behavior
```

```
1- *  S0 01
1- +  S0 01
```

The machine is reset only when the reset signal is asserted. If the machine is in an invalid state and no reset signal is applied, the behavior of the machine is still undefined.

Example

This example shows an equivalent design using .synchronous_reset.

```
.design example
.inputnames reset a clk
.outputnames found error
.clock clk rising_edge
.synchronous_reset reset rising S0 01

0 S0 S0 00
1 S0 S1 01
0 S1 S0 10
1 S1 S2 00
0 S2 S0 00
1 S2 S1 10
```

The first reset specification in the previous example (1– * S0 01) means that for any valid state in the machine (*), if the reset signal is high (1), the machine transitions to state S0 and asserts output values 01. Similarly, the second reset specification in this example (1– + S0 01) means that for any invalid state in the machine (+), if the reset signal is high (1), the machine transitions to state S0 and asserts output values 01.

Because * refers to all valid states and + refers to all invalid states, including both of these rows specifies that if signal reset is asserted, the machine transitions to state S0 and asserts output values 01, regardless of the present state.

An alternative design description style forces the machine into its reset state on entering any invalid state. The following example shows this style, which is equivalent to the description in the example equivalent machine using .synchronous_reset.

Example

```
.design example
.inputnames clk a
.outputnames found error
.clock clk rising_edge

0 S0 S0 00
1 S0 S1 01
0 S1 S0 10
1 S1 S2 00
0 S2 S0 00
1 S2 S1 10
- +  S0 01
```

The last line specifies that if the present state is invalid (in this case, the unassigned state code 2#11), the machine transitions to state S0 and asserts output values 01, regardless of the value of input signal a.

In many cases, extra logic is required for performing these additional transitions. For example, in a one-hot encoded machine, detecting that the machine is in an illegal state is very expensive in terms of circuit area. However, if the machine does enter an illegal state, it transitions immediately to the reset state without waiting for a reset signal.

The + operator can be used in any number of rows in the state table, as long as the behavior of the machine is unambiguous.

Next State Column

The next state column of a state table contains either a symbolic state name or one of the three special characters & (ampersand), – (minus), or ~ (tilde).

name

Any state's name.

& (ampersand)

Stay in present state— that is, the next state is to be the same as the current state.

– (minus)

Don't care— that is, the next state can be any state.

~ (tilde)

Don't know— that is, the next state is not specified. Another row can specify the next state.

Primary Outputs Column

The primary output section of a row in the state table body is interpreted the same as for a .type fr PLA (see Appendix C, "PLA Design Format"). Each column represents one output port in the order specified by the `.outputnames` statements.

The valid characters for the primary output columns are 0, 1, – (minus), and ~ (tilde):

0

The output is logical 0.

1

The output is logical 1.

– (minus)

Don’t care— that is, this output can be either logical 0 or 1.

~ (tilde)

Don’t know; this output is not specified. Another row can specify values for this output. If no other row specifies the values for this output, it is treated as a don’t care.

State Table Fields

A state table body can optionally specify that only a subset of the input ports (a subset field) is used. Use the following command:

```
.field  name_1  [ name_2 ... ]
```

The .field Command

Normally, each row in a state table must contain all inputs and outputs. For large state tables, this can be cumbersome when some rows use only a small number of inputs or outputs. The `.field` statement specifies a subset of inputs and outputs to be used for the following state table rows. The format is

```
.field name_1 [name_2 ...]
```

name_1 (name

The names of input and output signals that are specified in the columns of the state table immediately following.

In the `.field` specification, input ports must be specified before output ports. If the field specification is too long to fit on a single line, you can use multiple `.field` statements.

Example

This example shows two equivalent machine descriptions. The second uses the `.field` statement.

```
.design example
.inputnames clk a b c d e f
.outputnames f0 f1
.clock clk rising_edge

11---- S0  S1 00
--11-- S0  S2 01
----11 S0  S3 10

.design example
.inputnames  clk a b c d e f
.outputnames f0 f1
.clock clk rising_edge

.field a b f0 f1
11 S0  S1 00

.field c d f0 f1
11 S0  S2 01

.field e f f0 f1
11 S0  S3 10
```

Any input or output not included in the `.field` statement is considered irrelevant to the following rows. If the omitted inputs or outputs are meaningful for other rows, they must be specified in other rows.

A .field statement remains in effect until the next `.field` statement. To return to the full machine description, use a `.field` statement specifying all input and output names.

You must specify present and next states.

It is often convenient to specify the default values for an output as the first row of the state table.

Example

The first row in this example specifies that the first two outputs have a default value of 0 for all unspecified conditions and the last two outputs have a default value of 1 for all unspecified conditions.

```
.design example
.inputnames clk a b c d
.outputnames f0 f1 f2 f3
.clock clk rising_edge

%%%%   *      ~    0011

.field a b f0 f1
11   S0   S1   00

.field c d f2 f3
11   S0   S1   00
```

State Encoding

The optional .encoding section allows you to specify the encodings you want for states used in the state table. The encoding for a state is the actual binary values (bit vectors) used to represent that state. Every encoding must be unique and must contain the same number of bits.

The .encoding section is optional but if it is used, the encoding section must follow the state table body.

The syntax is

```
state_name  [encoding]
```

state_name

Name of any state.

encoding

Optional numeric value.

Example

This example shows the encoding section of the description in the first example.

```
.encoding
IDLE      2#00
FIVE      2#01
TEN       2#11
OWE_DIME  2#10
```

The sequence in which the states are listed defines an order. This order is important only if you want to reassign the encodings during state assignment, using the binary or gray encoding styles. You can list a state in the encoding section without an encoding; this defines an ordering for a state while leaving encoding unassigned.

If the .encoding section is omitted, the state codes are unspecified and the ordering of the states is arbitrary. You can manually specify or change the state encodings or state order with the commands `set_fsm_encoding` and `set_fsm_order`.

A subset of the states in the machine can be assigned encodings. During the compilation of the FSM, defined encodings are kept and any unassigned states are automatically encoded.

Specify encodings by using one of two formats: numeric base or alphabetic base.

Numeric Base Encoding

The numeric base encoding format uses a numeric base specifier (2, 8, 10, or 16), followed by the pound (#) character, followed by a string of digits in the given base.

The string of digits can be separated by underscore characters (_) to make the encodings easier to read.

If the base specification and the # character are omitted, the string of digits is interpreted as a decimal value.

For example, decimal 15 can be entered as

```
2#1111

8#17

10#15
15

16#f
```

Alphabetic Base Encoding

The alphabetic base encoding format uses a caret (^) character, followed by a one-character base specifier (B – binary, O – octal, D – decimal, or H – hex), followed by a string of digits in the given base.

The base specification character can be either uppercase or lowercase (for example, H or h).

If the ^ character and the base specifier are omitted, the string of digits is interpreted as a decimal value.

In this format, you can enter decimal 15 as

```
^B1111
^b1111

^O17
^o17

^D15
^d15

^Hf
^hf
```

Don't Care Sets

From the state table and knowledge of the encodings for each state, a don't care set is derived and used during optimization. The don't care sets are obtained from unspecified next state transitions, unused state encodings, and any other conditions in which the behavior of the machine is unspecified.

For example, if state codes 00 and 01 are used, unused state codes 10 and 11 are don’t care conditions. Also, the following row

```
001 IDLE  IDLE 011
```

results in the don’t care conditions of

```
--0 IDLE  ~  ---
-1- IDLE  ~  ---
1-- IDLE  ~  ---
```

These don’t care conditions can be examined by writing out a state table immediately after reading the state table.

Specifying ambiguous behavior for the machine is an error. For example, the following two state table rows cause a compilation error, because they specify transitions to two different states (S1 and S3) for the input condition 11:

```
1- S0   S1 10
-1 S0   S3 10
```

E

SDF Constructs

Table E-1 summarizes the SDF constructs used by the `read_timing -format sdf-v2.1` command.

Table E-1 SDF Constructs Used by read_timing -format sdf-v2.1

SDF construct	**SDF 2.1 support**
DELAYFILE	Parsed, not used.
SDFVERSION	Used. The SDFVERSION entry is mandatory for SDF 1.0 as well as SDF 2.1 files. The Design Compiler SDF reader parses this entry and, depending on the version number read, automatically invokes the SDF 1.0 parser or the SDF 2.1 parser.
DESIGN	Used. The design name must be the name of the current instance.
DATE	Parsed, not used.
VENDOR	Parsed, not used.
PROGRAM	Parsed, not used.
VERSION	Parsed, not used.

Table E-1 SDF Constructs Used by read_timing -format sdf-v2.1(continued)

SDF construct	SDF 2.1 support
DIVIDER	Used.
VOLTAGE	Parsed, not used.
PROCESS	Parsed, not used.
TEMPERATURE	Parsed, not used.
TIMESCALE	Used.
CELL	Used.
CELLTYPE	Used.
CORRELATION	Parsed, not used.
INSTANCE	Used.
DELAY	Used.
TIMINGCHECK	Used.
ABSOLUTE	Used.
INCREMENT	Not used.
PATHPULSE	Parsed, not used.
GLOBALPATHPULSE	Parsed, not used.

Table E-1 SDF Constructs Used by read_timing -format sdf-v2.1(continued)

SDF construct	**SDF 2.1 support**
IOPATH	Edge specification on input ports is ignored. 12 values are read; 6 values are used (combinational, enable and disable for rise and fall). The only mapping done is in the case of a single-valued SDF delay entry. This single value gets mapped to both the 01 (rise) and 10 (fall) timing arcs. If the library cell contains arrayed ports, the SDF file must contain bit-blasted entries. A range specification in the SDF file is not supported. The SDF file cannot reference an unindexed composite port name. In case of multiple annotations to the same delay site, the delay value selected is the last value specified in the SDF file. The worst value is annotated when the `-worst` option is specified. See the read_timing man page. If the library cell has conditional delays, the SDF IOPATH entry automatically applies to all conditional paths specified in the library cell. IOPATH delays between two output ports are supported although SDF does not allow output to output IOPATH. If the SDF file contains delays from input to output but there is no library timing arc between the two pins, a warning appears and the IOPATH is ignored.
IOPATH	Conditional expressions are ignored by Design Compiler. In case of multiple conditions to the same delay site, the delay value selected is the last value specified in the SDF file. The worst value is annotated when the `-worst` option is specified. See the read_timing man page. In case of multiple annotations to the same delay site, the delay value selected can be specified through the SDFPOLICY setup file variable.

Table E-1 SDF Constructs Used by read_timing -format sdf-v2.1(continued)

SDF construct	SDF 2.1 support
PORT	Delays are annotated between all pins in the fanin and the input port. If the SDF file contains INTERCONNECT to the same input port, the interconnect value overrides the port value. In case of multiple annotations to the same delay site, the delay value selected is the last value specified in the SDF file. The worst value is annotated when the -worst option is specified. See the read_timing man page. 12 values parsed; 2 values supported: 01, 10. The only mapping done is in the case of a single valued SDF delay entry. This single value gets mapped to both the 01 (rise) and 10 (fall) of the timing arc. If the library cell contains arrayed ports, the SDF file must contain bit-blasted entries. A range specification in the SDF file is not supported. The SDF file cannot reference an unindexed composite port name. The input port must be a leaf-cell pin. A warning message appears and the PORT is ignored if the input port is a nonleaf pin.

Table E-1 SDF Constructs Used by read_timing -format sdf-v2.1(continued)

SDF construct	**SDF 2.1 support**
INTERCONNECT	Net delays are annotated between the two given pins.
	If the SDF file contains PORT to the same input port, the interconnect value overrides the port value.
	In case of multiple annotations to the same delay site, the delay value selected is the last value specified in the SDF file. The worst value is annotated when the `-worst` option is specified. See the read_timing man page.
	12 values parsed; 2 values supported: 01, 10. The only mapping done is in the case of a single valued SDF delay entry. This single value gets mapped to both the 01 (rise) and 10 (fall) of the timing arc.
	If the library cell contains arrayed ports, the SDF file must contain bit-blasted entries. A range specification in the SDF file is not supported. The SDF file cannot reference an unindexed composite port name.
	Both pins must be leaf-cell pins. A warning message appears and the INTERCONNECT is ignored if either pin is a nonleaf pin.
NETDELAY	Parsed, not used.
DEVICE	Parsed, ignored.
conditional constraints	Parsed, not supported.
SETUP	Edge specification on both ports is supported.
	In case of multiple annotations to the same delay site, the delay value selected is the last value specified in the SDF file. The worst value is annotated when you specify the `-worst` option. See the read_timing man page.

Table E-1 SDF Constructs Used by read_timing -format sdf-v2.1(continued)

SDF construct	**SDF 2.1 support**
HOLD	Edge specification on both ports is supported. In case of multiple annotations to the same delay site, the delay value selected is the last value specified in the SDF file. The worst value is annotated when you specify the `-worst` option. See the `read_timing` man page.
SETUPHOLD	Edge specification on both ports is supported. In case of multiple annotations to the same delay site, the delay value selected is the last value specified in the SDF file. The worst value is annotated when you specify the `-worst` option. See the `read_timing` man page.
RECOVERY	Edge specification on both ports is supported. In case of multiple annotations to the same delay site, the delay value selected is the last value specified in the SDF file. The worst value is annotated when you specify the `-worst` option. See the `read_timing` man page.
REMOVAL	You can specify removal timing checks with the HOLD statement.
SKEW	Ignored.
WIDTH	Ignored.
PERIOD	Ignored.
NOCHANGE	Ignored.
PATHCONSTRAINT	Ignored.
SUM	Ignored.
DIFF	Ignored.
SKEWCONSTRAINT	Ignored.
C and C++ style comments	Supported.

F

Test Protocol File Syntax

The syntax of the Synopsys test protocol file is similar to that of the Synopsys Library Compiler source files. This appendix explains

- How to understand the structure of the test protocol file
- How to specify vectors in the test protocol file
- How to specify actions in the test protocol file
- How to debug the test protocol file

You can use this information to interpret a default test protocol or to create an initialization protocol. See the *Scan Synthesis User Guide* for information about creating an initialization protocol.

Understanding the Test Protocol File Structure

A test protocol file is organized as a series of nested groups; each group is bound by a pair of curly braces, {}. The test protocol file supports the following groups:

test protocol

> The test protocol group defines timing information for scan testing the design.

protocol

> The protocol group defines the steps (vectors) required for scan-testing the design.

program

> The program group defines the steps (vectors) required in a complete vector file.

pattern

> The pattern group defines the actions required to apply a single scan test pattern.

Each of these groups is required in the test protocol file, and there must be only one of each group. The following sections discuss each of these groups.

The test protocol file supports comments, which are delimited by the /* and */ characters. If a statement in your protocol file takes more than one line, use the line continuation character (\).

Example F-1 shows the structure of the test protocol file.

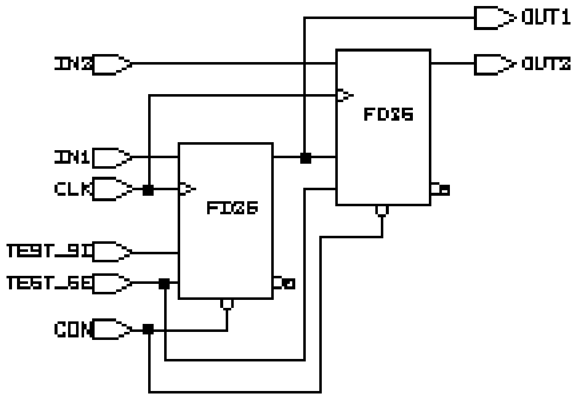
Example F-1 Test Protocol File Structure

```
test_protocol(){
   /* test protocol group */

   protocol_start(){
      /* protocol group */

      foreach_program(){
         /* program group */

         foreach_pattern(){
            /* pattern group */
         }
      }
   }
}
```

Test Protocol File Example

Figure F-1 shows a simple design example. This appendix uses default test protocol file for this design (shown in Example F-2) to illustrate the syntax components.

Figure F-1 Multiplexed Flip-Flop Design Example



Example F-2   Default Test Protocol for Multiplexed Flip-Flop Design

```
test_protocol() {
   period : 100.00 ;
   delay : 5.00 ;
   bidir_delay : 55.00 ;
   strobe : 95.00 ;
   strobe_width : 0.0;

   clock() {
      period : 100.00 ;
      waveform : {45.00, 55.00} ;
      sources : CLK ;
   }
   protocol_start() {
      foreach_program() {
         vector() {
            set(all_ports,”X,0,[4]X,[2]M”);
         }
         vector() {
            set(all_ports,”1,0,[4]X,[2]M”);
         }
         foreach_pattern() {
            stream(2) {
               set(all_ports,”1,C,[2]U,1,Si,M,So”);
            }
            vector() {
               set(all_ports,”Pi,0,[4]Pi,[2]Po”);
            }
            vector() {
               set(all_ports,”U,Cp,[4]U,[2]M”);
            }
            vector() {
               set(all_ports,”1,0,[2]U,1,U,M,So”);
            }
         }
      }
   }
```

Test Protocol Group

The test protocol group contains the timing information for scan testing the design. The timing information includes

- Test timing parameters
- Test clock waveforms

Test Timing Parameters

The test timing parameters include the test period, input delay time, bidirectional delay time, strobe time, and strobe width. These timing parameters influence test design rule checking and determine the timing in the test vector files produced by Test Compiler vector formatting. Timing parameter values ensure the accuracy of the rule checking process. They also provide important information that makes the test protocol file a complete test program template.

Table F-1 lists the keyword used in the test protocol file for each test timing parameter. The table also provides the dc_shell variables used to specify the timing parameter values. The parameter values must be positive real numbers. The time unit is nanoseconds (ns). For more information about specifying test timing parameters, see “Timing Analysis in Latches” in Chapter 13.

Table F-1 Test Protocol Timing Parameters

Parameter	**Keyword**	**Variable**
test period	period	test_default_period
input delay time	delay	test_default_delay
bidirectional delay time	bidir_delay	test_default_bidir_delay
strobe time	strobe	test_default_strobe

Table F-1 Test Protocol Timing Parameters (continued)

Parameter	Keyword	Variable
strobe width	strobe_width	test_default_strobe_width

The following timing parameters appear in the default test protocol (Example F-2) for the multiplexed flip-flop design example:

```
period : 100.00 ;
delay : 5.00 ;
bidir_delay : 55.00 ;
strobe : 95.00 ;
strobe_width : 0.0;
```

Test Clock Waveforms

The test protocol file uses clock group statements to specify test clock waveforms. The clock group statement has the following syntax:

```
clock() {   period : period_value ;
     waveform : {rise, fall} ;
     sources : clock_port_list ;
}
```

Note:

All clock timing values are in nanoseconds.

`period_value`

Specifies the period of the test clock. The `period_value` argument must be a positive real integer.

Note:

DC Expert *Plus* requires identical values for the test clock period value and the test period timing parameter.

`rise`

Specifies the time, relative to the start of the tester cycle, of the rising edge of the clock (clock makes a low-to-high transition). The rise argument must be a positive real integer. If the rise value is greater than period_value, DC Expert *Plus* calculates the rise value as (rise mod period_value).

`fall`

Specifies the time, relative to the start of the tester cycle, of the falling edge of the clock (clock makes a high-to-low transition). The fall argument must be a positive real integer. If the fall value is greater than period_value, DC Expert *Plus* calculates the fall value as (fall mod period_value).

`clock_port_list`

Specifies the list of clock ports that have the specified timing characteristics. The syntax for clock_port_list is the same as for dc_shell commands (if the list contains more than one port, enclose it in curly braces, {}).

Use the `create_test_clock` command to specify test clock waveforms. See “Timing Analysis in Latches” in Chapter 13 for information about specifying clock waveforms.

Protocol Group

During scan testing, vectors specified in the protocol group (not nested in the program group) are applied once at the start of the pattern set. If the pattern set generated by Test Compiler ATPG is divided into multiple vector files during vector formatting (`write_test` command), the unnested vectors specified in the protocol group occur only once in the first vector file. See the *Test Compiler Reference Manual* for information about Test Compiler ATPG and vector formatting.

A typical test protocol file does not contain any unnested vectors in the protocol group. See "Specifying Vectors in the Test Protocol File" on page F-10 for information about vector specification syntax.

Program Group

During scan testing, vectors specified in the program group (not nested in the pattern group) are applied at the start of each vector file. If the pattern set generated by Test Compiler ATPG is divided into multiple vector files during vector formatting, the unnested vectors specified in the program group occur once in each vector file.

For example, the program group in the default test protocol, shown in Example F-2, defines the following tester initialization sequence:

```
vector() {
   set(all_ports,”X,0,[4]X,[2]M”);
}
vector() {
   set(all_ports,”1,0,[4]X,[2]M”);
}
```

See "Specifying Vectors in the Test Protocol File" on page F-10 for information about vector specification syntax.

Pattern Group

The pattern group defines how to scan in the data, perform the parallel measure cycle, perform the parallel capture cycle, and scan out the data. During scan testing, the actions specified in the pattern group are applied once for each generated pattern.

For example, the pattern group in the default test protocol shown earlier in Example F-2 defines the following scan pattern application actions:

```
stream(2) {
   set(all_ports,”1,C,[2]U,1,Si,M,So”);
}
vector() {
   set(all_ports,”Pi,0,[4]Pi,[2]Po”);
}
vector() {
   set(all_ports,”U,Cp,[4]U,[2]M”);
}
vector() {
   set(all_ports,”1,0,[2]U,1,U,M,So”);
}
```

DC Expert *Plus* infers actions in the pattern group during test design rule checking. You cannot modify these actions. See “Understanding Actions in the Test Protocol File” on page F-15 for information about action specification syntax.

Specifying Vectors in the Test Protocol File

A vector corresponds to one tester cycle on your automated test equipment. The test protocol file uses vector group statements to specify vectors.

The vector group statement has the following syntax:

```
vector(n) {
vector_information
}
```

n

Specifies an optional vector count. If you omit the *n* argument, the specified vector is executed once.

`vector_information`

Defines the logic values for the vector or vectors using one or more set statements.

The syntax of the set statement is

```
set (port_list,”value_list”);
```

`port_list`

Specifies the list of ports to which logic values are assigned.

The `port_list` argument consists of one or more port names (or port sets) separated by commas. A port set is a variable that defines a group of ports. See the “Port Sets” section later in this appendix for more information about port sets.

You can include a port in multiple set statements. However, DC Expert *Plus* retains only the last value assigned to a port within a vector group.

Any port not assigned a value in the current set statement retains its previous value.

`value_list`

Specifies logic values for the ports specified in `port_list`.

The `value_list` argument contains a logic value for every port in the port list. Values in the `value_list` argument are separated by commas, and the entire list must be enclosed in double quotation marks. If you have a long list with repeated values, you can use shorthand notation with a value repeater. See the “Value Repeaters” section later in this appendix for more information.

Logic values can represent the logic level applied to a nonclock input port, a clock pulse applied to a clock input port, or a masked output port. DC Expert *Plus* does not support the use of logic values in the test protocol file to represent expected response data for output ports. Table F-2 lists the logic values used in the test protocol file and their meaning in test protocol files.

Table F-2 Logic Values in Test Protocol Files

Logic value	Definition
1	Test Compiler ATPG must apply a logic 1 value.
0	Test Compiler ATPG must apply a logic 0 value.
X	Test Compiler ATPG can apply either a logic 0 or logic 1 value (input don’t care value).
Z	Specifies an inactive bidirectional port.
M	Test Compiler ATPG masks the output port (output don’t care value)
C	Test Compiler ATPG applies a clock pulse.

Although DC Expert *Plus* explicitly defines the values for each port in the design (`all_ports`), you do not need to do so for manually generated initialization protocols. Any port not assigned a value in the current set statement retains its previous value.

Port Sets

DC Expert *Plus* provides four predefined port sets:

all_inputs

Contains the input ports of the design in alphanumeric order.

all_outputs

Contains the output ports of the design in alphanumeric order.

all_bidirectionals

Contains the bidirectional ports of the design in alphanumeric order.

all_ports

Contains all the ports in the design in the following order: all input ports, all output ports, all bidirectional ports.

In an inferred test protocol, DC Expert *Plus* uses the all_ports port set in each set statement it generates.

A comment at the end of the test protocol inferred by the `check_test` command shows the port order for each of the predefined port sets. Example F-3 shows the predefined port sets for the multiplexed flip-flop design example in the earlier Figure F-1.

Example F-3 Predefined Port Sets for Multiplexed Flip-Flop Design

```
/* Default Port Sets:

  (The all_ports set is comprised serially of the
   all_inputs, all_outputs, and all_bidirectionals sets)

ordered ‘all_inputs’ members:
CDN,
CLK,
IN1,
IN2,
TEST_SE,
TEST_SI
ordered ‘all_outputs’ members:
OUT1,
OUT2,
No ‘all_bidirectionals’ members:
*/
```

Value Repeaters

The syntax for a value repeater is

`[n]logic_value`

n

Indicates the number of times to repeat the logic value that follows. If the n argument is omitted, the logic value is repeated for every port in the port list.

`logic_value`

One of the logic values shown in Table F-2.

Using a value repeater you can rewrite the value list “1,1,1,1,1” as “[5]1”. You can mix single values and repeated values to improve file readability. For example, instead of writing “[5]1”, you can write “[3]1,1,1” or “[2]1,1,[2]1”.

Vector Example

The following vector groups define the two tester initialization vectors in the default test protocol for the multiplexed flip-flop design in the earlier Table F-2.

```
vector() {
   set(all_ports,”X,0,[4]X,[2]M”);
}
vector() {
   set(all_ports,”1,0,[4]X,[2]M”);
}
```

By default, the first tester initialization vector applies don’t care values to all nonclock input ports (IN1, IN2, TEST_SI, TEST_SE, and CDN), applies a logic 0 to the system clock (CLK) to force it inactive, and masks all output ports (OUT1 and OUT2).

Refer to the port order shown for the all_ports port set in Example F-3 to see that the set statement in the first vector group

- Applies a don’t care input (X) to the first port (CDN)
- Applies a logic 0 to the next port (CLK)
- Applies a don’t care input (X) to the next four ports (IN1, IN2, TEST_SE, and TEST_SI)
- Masks (M) the next two ports (OUT1 and OUT2)

The second tester initialization vector performs a similar function as the first vector, except that it holds the asynchronous input (CDN) inactive (logic 1).

Although DC Expert *Plus* explicitly defines the values for each port in the design (all_ports), you do not need to do so for manually generated initialization protocols. You can write the second vector group as

```
vector() {
   set (CDN,”1”);
}
```

All other ports retain their previous values.

Understanding Actions in the Test Protocol File

A vector specification that includes symbolic values is called an action. The test protocol file uses vector group statements or a stream group statement to specify actions.

A symbolic value represents the pattern data applied to an input port or compared at an output port. The pattern data can be either scan data or parallel data. When you format the vectors, Test Compiler replaces the symbolic value in the test protocol file with the logic values in the generated scan pattern. DC Expert *Plus* infers the actions in the pattern group during test design rule checking. You cannot modify these actions.

Table F-3 lists the symbolic values used in the test protocol file and their meaning in test protocol files.

Table F-3 Symbolic Values in Test Protocol Files

Symbolic value	**Data type**
Si	Scan-in pattern data
So	Scan-out pattern data

Table F-3 Symbolic Values in Test Protocol Files (continued)

Symbolic value	**Data type**
Pi	Parallel pattern input data
Po	Parallel pattern output data
Pio	Parallel pattern input/output data (bidirectional ports)
U	Data remains in previous state (unchanged)
Cp	Parallel cycle clock pulse pattern data

Note:

> The symbolic value Cp is used for the system clock during the capture cycle (referred to as the capture clock). If capture clocks do not have the symbolic value Cp during the parallel capture cycle, DC Expert *Plus* generates test design rule warnings. A capture clock is not pulsed in every parallel capture cycle. If the scan pattern specifies an active clock, the capture clock is pulsed; if the scan pattern specifies an inactive clock, the capture clock is not pulsed for that pattern.

The test protocol file uses a stream group statement in the program group to define scan shift behavior. A stream group statement implies that Design Compiler applies or compares a stream of serial data at specified ports during many test vectors and maps efficiently into the scan-specific constructs in the test vector files.

A stream group has the following syntax:

```
stream(n) {
/* stream_information */
}
```

n

Specifies an optional vector count. If the n argument is omitted, the stream is unbounded. An unbounded stream generates all the scan-in and scan-out vectors required to load and unload the scan chains specified in the set statements. If the n argument is present, exactly n vectors are generated (scan data is shifted n times).

`stream_information`

Defines the logic and symbolic values for streams using one or more set statements.

Example F-4 shows the pattern group of the test protocol for the design example in Figure F-1. This pattern group provides an example of a stream group. See Table F-2 and Table F-3 for definitions of the logic values and symbolic values used in the set statements.

Example F-4 Pattern Group for Multiplexed Flip-Flop Design

```
foreach_pattern() {
   stream(2) {
      set(all_ports,”1,C,[2]U,1,Si,M,So”);
   }
   vector() {
      set(all_ports,”Pi,0,[4]Pi,[2]Po”);
   }
   vector() {
      set(all_ports,”U,Cp,[4]U,[2]M”);
   }
   vector() {
      set(all_ports,”1,0,[2]U,1,U,M,So”);
   }
}
```

In the multiplexed flip-flop design example, scan load involves holding the scan enable signal (TEST_SE) at logic 1, applying scan data at the scan input port (TEST_SI), and applying the system clock (CLK). In the inferred test protocol, scan load and scan unload occur simultaneously to minimize the required test time. The expected response is compared at the scan output port (OUT2) at the specified strobe time. The asynchronous control port is held at logic 1 to prevent any resetting of scan data.

The set statement in the stream group accomplishes this by

- Applying logic 1 to CDN
- Applying a clock pulse to CLK
- Leaving IN1 and IN2 unchanged from the previous vector
- Applying logic 1 to TEST_SE
- Applying scan-in pattern data (Si) to TEST_SI
- Masking OUT1
- Comparing TEST_SO with scan-out pattern data (So)

For the symbolic values Si and So, the bits of the scan word generated by ATPG for the current scan pattern replace the Si and So values in the vectors generated when the stream group is expanded during vector formatting. In a similar manner, the parallel stimulus and expected parallel response for the current scan pattern replace the Pi and Po values in the vector groups of the pattern group. (These vector groups represent the parallel cycles.)

Debugging Test Protocol Files

During test design rule checking, DC Expert *Plus* performs a symbolic simulation of the test protocol for a design. You can consider a test protocol to be a testbench for the symbolic simulator of the design rule checker.

To understand the behavior of the design under the conditions imposed by the test protocol, you often need to examine internal simulation data. You can see this data using the net tracing capabilities of the check_test command. When you enable net tracing, DC Expert *Plus* prints a table of simulation values of the selected nets after rule checking and before producing any summary information. See “Tracing Nets During Protocol Simulation” in Chapter 12 for information about enabling net tracing during test design rule checking.

Example F-5 shows the table generated for the multiplexed flip-flop design in Figure F-1 as a result of the following command sequence:

```
dc_shell> trace_nets {CLK, IN*, OUT*, TEST*, CDN}
dc_shell> check_test
```

Example F-5   Net Tracing Results Table

```
/*****************************************************************/
/* TRACE NET LEGEND:                                             */
/*   1. VECTORS                                                  */
/*      IV `i' : `i'th Initialization Vector                     */
/*       SI(i) : Serial Scan In (repeated `i' times)             */
/*         PMV : Parallel Measure Vector                         */
/*         CAP : Capture Vector                                  */
/*         1SO : First Scan-out Measure Vector                   */
/*       SO(i) : Serial Scan Out (repeated `i' times)            */
/*         VEC : Parallel Vector                                 */
/*                                                               */
/*   2. VALUES                                                   */
/*           0 : Logic 0                                         */
/*           1 : Logic 1                                         */
/*           z : High Impedance Value                            */
/*           u : Uninitialized                                   */
/*          Si : Scanned-in Value                                */
/*          Is : Initial State                                   */
/*          Rd : Response Data                                   */
/*           x : Unknown                                         */
/*          So : Scanned-out Value                               */
/*          Na : Net Values Not Available (Vector Not Simulated) */
/*****************************************************************/

STARTING PRINT OF TRACE INFORMATION FOR:

Nets Traced In the Command Line:
  CDN
  CLK
  IN1
  IN2
  OUT1
  OUT2
  TEST_SE
  TEST_SI
                  C   C   I   I   O   O   T   T
                  D   L   N   N   U   U   E   E
                  N   K   1   2   T   T   S   S
                  .   .   .   .   1   2   T   T
                  .   .   .   .   .   .   _   _
                  .   .   .   .   .   .   S   S
                  .   .   .   .   .   .   E   I
          ========================================
     IV 1 ----------------------------------------
          0.0     x   0   x   x   x   x   x   x
     IV 2 ----------------------------------------
          5.0     1   0   x   x   x   x   x   x
```

```
SI(2) ----------------------------------------
      0.0     1   0   x   x   Is  Is  x   x
      5.0     1   0   x   x   Is  Is  1   Si
      45.0    1   1   x   x   Si  Is  1   Si
      55.0    1   0   x   x   Si  Is  1   Si
  PMV ----------------------------------------
      0.0     1   0   x   x   Si  Si  1   Si
      5.0     1   0   x   x   Si  Si  x   x
      5.0     x   0   x   x   x   x   x   x
  CAP ----------------------------------------
      45.0    Na  Na  Na  Na  Na  Na  Na  Na
      55.0    Na  Na  Na  Na  Na  Na  Na  Na
  1SO ----------------------------------------
      0.0     x   0   x   x   x   x   x   x
      5.0     1   0   x   x   x   x   1   x
      95.0    1   0   x   x   Rd  So  1   x
SO(2) ----------------------------------------
      45.0    1   1   x   x   x   Rd  1   x
      55.0    1   0   x   x   x   Rd  1   x
      95.0    1   0   x   x   x   So  1   x
      ========================================
```

The legend at the beginning of the table briefly describes vector acronyms and vector values that might appear in the table of simulation values. Table F-4 and Table F-5 provide more-detailed descriptions of the vector acronyms and vector values, respectively.

After the legend, Test Compiler lists the traced nets. In the table of simulation values, each column represents a net specified for tracing.

The `check_test` command generates a section, delimited by horizontal lines, for each vector in the test protocol. A vector acronym (see Table F-4) to the left of the top line identifies the protocol vector.

Each section contains one or more rows. The `check_test` command prints a row of net values each time a traced net changes value. A vector-relative time stamp appears at the beginning of each row. A vector value (see Table F-4) defines the current state of each traced net.

Table F-4 Vector Acronyms in Net Tracing Results

Acronym	Description
IV i	The initialization vector tag includes a number (i) that corresponds to the vector's position in the sequence. For example, if an initialization sequence contains 20 vectors, the first vector in the sequence is tagged IV 1 and the twentieth vector in the sequence is tagged IV 20.
SI(i)	The serial scan-in vector is associated with the stream statement in the protocol. The tag includes a number enclosed in parentheses (i) that indicates the number of times the serial scan-in vector is repeated. For example, if the longest scan chain in a design that uses a default protocol is 120 vectors long, the vector is denoted by SI(120).
PMV	The parallel measure vector applies primary inputs and bidirectionals in input mode. This vector also strobes primary outputs and bidirectionals in output mode.
CAP	The capture vector in the protocol.
1SO	This is the first scan-out measure vector in the protocol. It is not present in strobe-before-clock protocols. The strobe-before-clock protocol type is discussed in the *Scan Synthesis User Guide*.
SO(i)	The serial scan-out vector is associated with the stream statement in the protocol. The tag includes a number in parentheses (i) that has a value equal to the longest scan chain.

Table F-5 Vector Values in Net Tracing Results

Value	Description
0	Logic 0
1	Logic 1
Z	High-impedance
u	Unassigned (The net was never assigned a value.)

Table F-5 Vector Values in Net Tracing Results (continued)

Value	Description
Si	Scanned-in (The net has the scan-in value propagated from a scan-input port.)
Is	Initial state (The symbolic state of a sequential element before the start of the scan operation.)
Rd	Response data (The symbolic state of a sequential element immediately after the capture operation.)
x	Unknown value
So	Scanned-out (The net has response data that is observable at a scan-output port.)
Na	Not available (This value appears for all nets when the vector has not been simulated.)

See the *Scan Synthesis User Guide* for details about debugging test protocols.

G

Latch-Based Design Code Examples

This chapter provides code examples of designs that use various types of latches. It includes these sections:

- SR Latch
- D Latch
- D Latch With Asynchronous Reset
- D Latch With Asynchronous Set and Reset
- D Latch With Enable (avoiding clock gating)
- D Latch With Enable and Asynchronous Reset
- D Latch With Enable and Asynchronous Set
- D Latch With Enable and Asynchronous Set and Reset

SR Latch

This section shows the VHDL and Verilog code that implements a design that uses an SR latch. It includes these subsections:

- VHDL and Verilog Code Examples for SR Latch
- Inference Report for an SR Latch
- Synthesized Design for an SR Latch

VHDL and Verilog Code Examples for SR Latch

To implement an SR latch in either VHDL or Verilog, you must set the following variable to true:

```
hdlin_report_inferred_modules
```

You must also set the following attribute:

```
attribute async_set_reset of RESET, SET : signal is "true";
```

Example G-1 shows the VHDL code that infers an SR latch.

Example G-1   VHDL Code for an SR Latch

```
library IEEE, SYNOPSYS;
use IEEE.std_logic_1164.all;
use SYNOPSYS.attributes.all;

entity SR_LATCH is
  port ( RESET, SET : in std_logic;
         Y : out std_logic );
end SR_LATCH;

architecture BEHAVIORAL of SR_LATCH is
attribute async_set_reset of RESET, SET : signal is "true";
begin
  infer : process ( RESET, SET )
  begin
    if ( RESET = ’0’ ) then
        y <= ’0’;
    elsif ( SET = ’0’ ) then
        y <= ’1’;
    end if;
  end process infer;
end BEHAVIORAL;
```

Example G-2 shows Verilog code that infers an SR latch.

Example G-2   Verilog Code Example for an SR Latch

```
module SR_LATCH( reset,set, y);
input reset,set ;
output y ;
// synopsys async_set_reset "reset,set"
reg y ;

always @(set or reset)
     begin : infer
        if (reset == 0)
         y = 1’b0 ;
        else if (set == 0)
         y = 1’b1 ;
     end
endmodule
```

Inference Report for an SR Latch

Example G-3 shows the inference report generated for an SR latch from the VHDL code shown in Example G-1 or the Verilog code in Example G-2.

Example G-3   Inference Report for an SR Latch

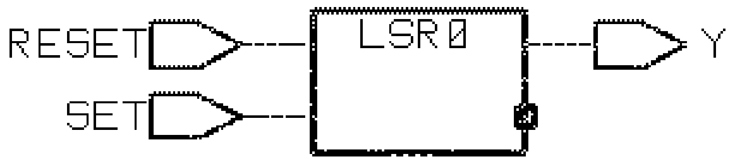
```
Inferred memory devices in process ’infer’
        in routine SR_LATCH line 13 in
         file ’/home/sudipto/work/latch_appl/rtl/vhdl/sr_latch.vhdl’.

=============================================================================
|    Register Name   |    Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
==============================================================================
|      Y_reg         |    Latch   |   1    |  - | -  | Y  | Y  | -  | -  | -  |
==============================================================================

 Y_reg
-----
    Async-reset: RESET’
    Async-set: SET’
    Async-set and Async-reset ==> Q: 0
```

Synthesized Design for an SR Latch

Figure G-1 shows the synthesized design for the SR-latch-based design resulting from compilation of the code shown in Example G-1 or Example G-2. In this synthesis, LSR0 is an SR latch and both S and R are active-low Inputs. Here is the target library description of the latch for the cell LSR0:

```
  latch ("IQ","IQN") {
    clear    : "R’";
    preset   : "S’";
    clear_preset_var1 : L;
    clear_preset_var2 : L;
  }
```

Figure G-1   Synthesized Design for an SR Latch



D Latch

This section shows the VHDL code that implements a design that uses a simple D latch. It includes these subsections:

- VHDL Code for a D Latch
- Inference Report for a D Latch
- Synthesized Design for a D Latch

VHDL Code for a D Latch

To implement a D latch in VHDL, you must set the following variable to true:

```
hdlin_report_inferred_module
```

Example G-4 shows the VHDL code that implements a design using a simple D latch.

Example G-4 VHDL Code for a D Latch

```
library IEEE, SYNOPSYS;
use IEEE.std_logic_1164.all;
use SYNOPSYS.attributes.all;

entity d_latch is
  port ( enable, data : in std_logic;
         y : out std_logic );
end d_latch;

architecture behavioral of d_latch is
begin
  infer : process ( enable, data )
  begin
    if ( enable = ’1’ )
    then
      y <= data;
    end if;
  end process infer;
end behavioral;
```

Inference Report for a D Latch

Example G-5 shows the inference report for a D latch resulting from compilation of the code in Example G-4.

Example G-5 Inference Report for a D Latch

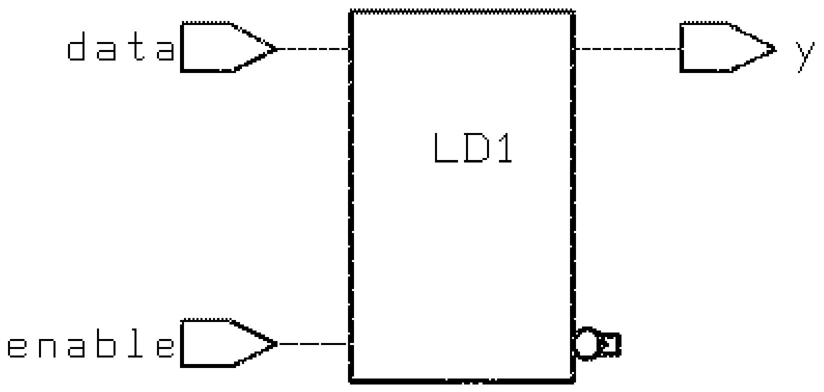
```
Inferred memory devices in process ’infer’
        in routine d_latch line 13 in file
         ’/home/sudipto/work/latch_appl/rtl/vhdl
         /d_latch.vhdl’.
===============================================================================
|    Register Name    |   Type   | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|        y_reg        |   Latch  |   1   |  -  | -  | N  | N  | -  | -  | -  |
===============================================================================
y_reg
-----
reset/set: none
```

Synthesized Design for a D Latch

Figure G-2 shows the synthesized design for the D latch resulting from compilation of the code in Figure G-4. In this design, LD1 is the simple D latch. Here is the target library description of the latch for the cell LD1:

```
latch ("IQ","IQN") {
  enable : "G";
  data_in : "D";
}
```

Figure G-2 Synthesized Design for a D Latch



D Latch With Asynchronous Reset

This section shows the VHDL and Verilog code that implements a design that uses a D latch with asynchronous reset. It includes these subsections:

- VHDL and Verilog Code for a D Latch With Asynchronous Reset
- Inference Report for a D Latch With Asynchronous Reset
- Synthesized Design for a D Latch With Asynchronous Reset

VHDL and Verilog Code for a D Latch With Asynchronous Reset

To implement a D latch with asynchronous reset in either VHDL or Verilog, you must set the following variable to true:

```
hdlin_report_inferred_modules
```

You must also set the following attribute, as illustrated by the code examples:

```
attribute async_set_reset of RESET, SET : signal is "true";
```

Example G-6 shows VHDL code for a D latch with asynchronous reset.

Example G-6 VHDL Code for a D Latch With Asynchronous Reset

```
library IEEE, SYNOPSYS;
use IEEE.std_logic_1164.all;
use SYNOPSYS.attributes.all;

entity d_latch_async_reset is
  port ( enable, reset, data : in std_logic;
         q : out std_logic );
end d_latch_async_reset;

architecture behavioral of d_latch_async_reset is
attribute async_set_reset of reset : signal is "true";
begin
  infer : process ( enable, reset, data )
  begin
    if ( reset = ’1’ )
    then
      q <= ’0’;
    elsif ( enable = ’1’ )
    then
      q <= data;
    end if;
  end process infer;
end behavioral;
```

Example G-7 shows Verilog code for the D latch with asynchronous reset.

Example G-7   Verilog Code for a D Latch With Asynchronous Reset

```
module d_latch_async_reset (enable, reset, data, q) ;
input enable, data, reset ;
output q ;
// synopsys async_set_reset "reset"
reg q ;

always @(reset or enable or data)
     begin : infer
        if (reset == 1)
          q = 1’b0 ;
        else if (enable == 1)
          q = data ;
     end
endmodule
```

Inference Report for a D Latch With Asynchronous Reset

Example G-8 shows the inference report for a D latch with asynchronous reset resulting from compilation of the code in Example G-6 or Example G-7.

Example G-8   Inference Report for a D Latch With Asynchronous Reset

```
Inferred memory devices in process ’infer’
        in routine d_latch_async_reset
         line 13 in file
         ’/home/sudipto/work/latch_appl/rtl/vhdl/d_latch_async_reset.vhdl’.
===============================================================================
|    Register Name    |    Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|        q_reg        |   Latch    |   1   |  -  | -  | Y  | N  | -  | -  | -  |
===============================================================================

q_reg
-----
     Async-reset: reset
```

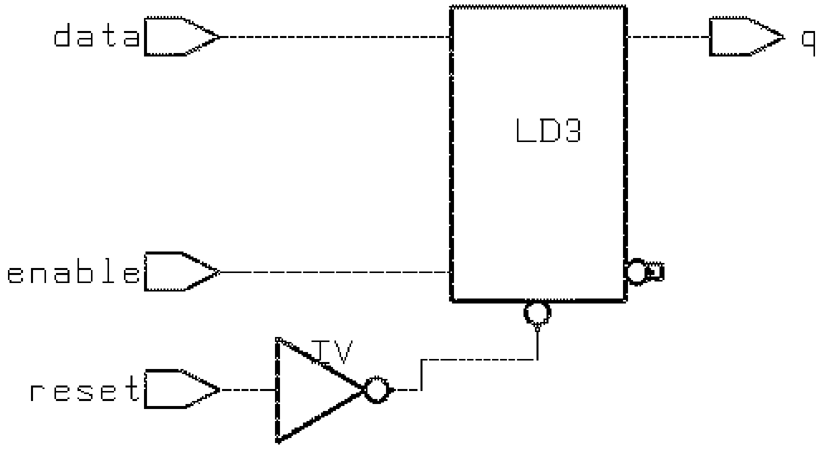
Synthesized Design for a D Latch With Asynchronous Reset

Figure G-3 shows the synthesized design for the D latch resulting from compilation of the code shown in Example G-6 and Example G-7. In this design, LD3 is the simple D latch. Here is the target library description of the latch for the cell LD3:

LD3 is a D latch with asynchronous active-low reset (CD). The description of the latch for the cell LD3 in the target library is as follows:

```
latch ("IQ","IQN") {
  enable   : "G";
  data_in  : "D";
  clear    : "CD’";
}
```

Figure G-3 Synthesized Design for a D Latch With Asynchronous Reset



D Latch With Asynchronous Set and Reset

This section shows the VHDL and Verilog code that implements a design that uses a D latch with asynchronous set and reset. It includes these subsections:

- VHDL and Verilog Code for a D Latch With Asynchronous Set and Reset
- Inference Report for a D Latch With Asynchronous Set and Reset
- Synthesized Design for a D Latch With Asynchronous Set and Reset

VHDL and Verilog Code for a D Latch With Asynchronous Set and Reset

To implement a D latch with asynchronous set and reset in either VHDL or Verilog, you must set the following variable to true:

```
hdlin_report_inferred_modules
```

You must also set the following attributes, as illustrated by the code examples:

```
attribute async_set_reset of set, reset : signal is "true";
attribute one_hot of set, reset : signal is "true";
```

Example G-9 shows the VHDL code for a D latch with asynchronous set and reset attributes.

Example G-9   VHDL Code for a D Latch With Asynchronous Set and Reset

```
library IEEE, SYNOPSYS;
use IEEE.std_logic_1164.all;
use SYNOPSYS.attributes.all;

entity d_latch_async_set_reset is
  port ( enable, set, reset, data : in std_logic;
         q : out std_logic );
end d_latch_async_set_reset;

architecture behavioral of d_latch_async_set_reset is
attribute async_set_reset of set, reset : signal is "true";
attribute one_hot of set, reset : signal is "true";
begin
  infer : process ( enable, set, reset, data )
  begin
    if ( reset = ’1’ )
    then
      q <= ’0’;
    elsif ( set = ’1’)
    then
      q <= ’1’;
    elsif ( enable = ’1’ )
    then
      q <= data;
    end if;
  end process infer;
end behavioral;
```

Example G-10 shows the Verilog code for a D latch with asynchronous set and reset attributes.

Example G-10 Verilog Code for a D Latch With Asynchronous Set and Reset

```
module d_latch_async_set_reset
(enable, set, reset, q, data) ;

input enable, set, reset, data ;
output q ;

// synopsys async_set_reset "set, reset"
// synopsys one_hot "set, reset"

reg q ;

always @(enable or set or reset or data)
     begin : infer
        if (reset == 1)
          q = 1’b0 ;
        else if (set == 1)
          q = 1’b1 ;
        else if (enable == 1)
          q = data ;
     end
endmodule
```

Inference Report for a D Latch With Asynchronous Set and Reset

Example G-11 shows the inference report for a D latch with asynchronous set and reset resulting from compilation of the code shown in Example G-9 or Example G-10.

Example G-11   Inference Report for a D Latch With Asynchronous Set and Reset

```
Inferred memory devices in process ‘infer’
        in routine d_latch_async_set_reset
         line 14 in file
        ’/home/sudipto/work/latch_appl/rtl/vhdl/d_latch_async_set_reset.vhdl’.
=============================================================================
|    Register Name    |   Type | Width | Bus | MB | AR | AS | SR | SS | ST |
=============================================================================
|        q_reg        | Latch  |   1   |  -  | -  | Y  | Y  | -  | -  | -  |
=============================================================================

q_reg
-----
    Async-reset: reset
    Async-set: set
    Async-set and Async-reset ==> Q: X
```

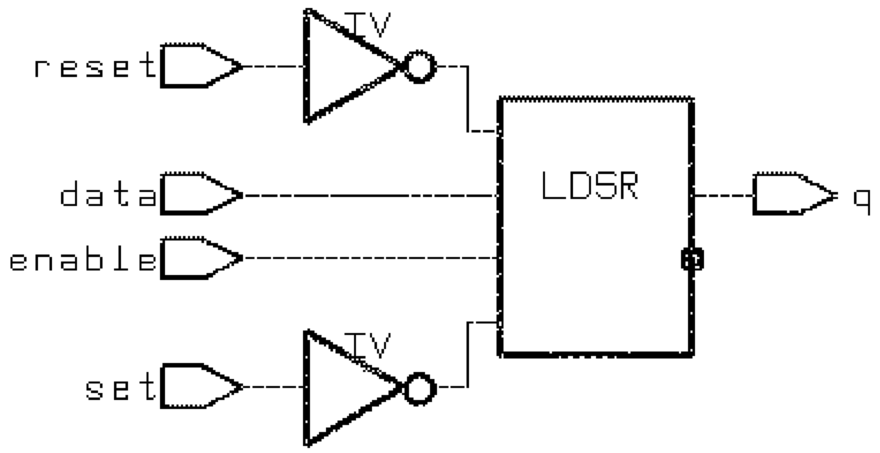
Synthesized Design for a D Latch With Asynchronous Set and Reset

Figure G-4 shows the synthesized design for the D latch resulting from compilation of the code in Example G-9 and Example G-10. In this design, LDSR is a D latch with active-low asynchronous set (SET) and reset (CLR).

The description of the latch for the cell LDSR in the target_library is as follows:

```
   latch ("IQ","IQN") {
    enable   : "G";
    data_in  : "D";
    clear    : "CLR’";
    preset   : "SET’";
    clear_preset_var1 : L;
    clear_preset_var2 : L;
  }
```

Figure G-4 Synthesized Design for a D Latch With Asynchronous Set and Reset



D Latch With Enable (avoiding clock gating)

This section shows the VHDL and Verilog code that implements a design that uses a D latch with the enable attribute to avoid clock gating. It includes these subsections:

- VHDL and Verilog Code for a D Latch With Enable
- Inference Report for a D Latch With Enable
- Synthesized Design for a D Latch With Enable
- Inferring Gated Clocks

VHDL and Verilog Code for a D Latch With Enable

To implement a D latch with enable in either VHDL or Verilog, you must set the following variable to true:

```
hdlin_report_inferred_modules
```

You must also set the following attribute to true:

```
hdlin_keep_feedback
```

Example G-12 shows the VHDL code for a D latch with enable.

Example G-12   VHDL Code for a D Latch With Enable

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity d_latch_enab is
  port ( enable, clock, data : in std_logic;
         q : buffer std_logic );
end d_latch_enab;

architecture behavioral of d_latch_enab is
begin
  infer : process ( enable, clock, data )
    begin
    if ( clock = ’1’ )
    then
      if ( enable = ’1’ )
      then
        q <= data;
      else
        q <= q;
      end if;
    end if;
  end process infer;
end behavioral;
```

Example G-13 shows the Verilog code for a D latch with enable.

Example G-13 Verilog Code for a D Latch With Enable

```
module d_latch_enab ( enable, clock, data, q) ;

input enable, clock, data ;
output q ;

reg q ;


always  @(enable or clock or data)
     begin :infer
        if  (clock == 1)
         begin
            if (enable == 1)
             q = data ;
            else
             q = q ;
         end
      end
endmodule
```

Inference Report for a D Latch With Enable

Example G-14 shows the inference report for a D latch with enable resulting from compilation of the code in Example G-12 or Example G-13.

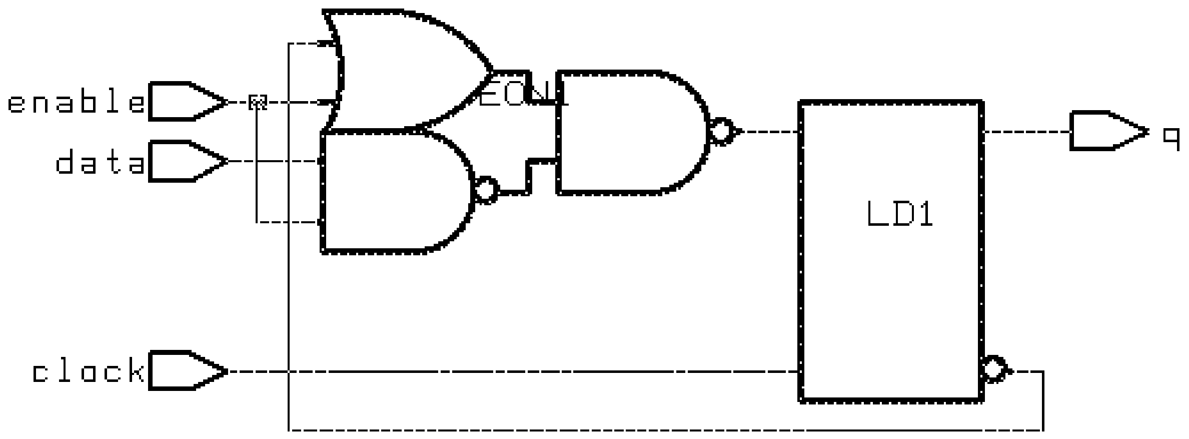
Example G-14 Inference Report for a D Latch With Enable

```
Inferred memory devices in process ’infer’
        in routine d_latch_enab line 13 in
         file ’/home/sudipto/work/latch_appl/rtl/vhdl/d_latch_enab.vhdl’.
=============================================================================
|    Register Name    |    Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
=============================================================================
|        q_reg         |   Latch  |   1   |  -  | -  | N  | N  | -  | -  | -  |
=============================================================================

q_reg
-----
    reset/set: none
```

Synthesized Design for a D Latch With Enable

Figure G-5 shows the synthesized design for the D latch with enable resulting from compilation of the code in Example G-12 or Example G-13.

Figure G-5 Synthesized Design for a D Latch With Enable



Inferring Gated Clocks

This section describes two cases—Case 1 and Case 2—in which HDL Compiler infers gated clocks.

Case 1

If the variable `hdlin_keep_feedback` is not set to TRUE, then HDL Compiler assumes the default value of false and removes all feedback loops. For example, feedback loops inferred from a statement such as the following

```
Q = Q
```

are removed.

The loop that is inferred from the following statement, shown in VHDL code, is removed.

```
if ( enable = ’1’ )
then
  q<= data;
else
  q <= q;
end if;
```

The code indicates that the gated clock in the synthesized design in Figure G-6, which does not have a feedback loop, is removed.

Case 2

Gated clocks can also be inferred from the coding style used to implement a design. For example, if the VHDL code is written in either of the coding styles in Example G-15 or Example G-16, regardless of whether `hdlin_keep_feedback` is set to true, Design Compiler will create a gated clock for the design.

Example G-15 implies a priority coding style— that is, the clock value is assessed first and enable is considered only if the clock is a certain value.

Example G-15 Coding Style A

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity d_latch_enab is
  port ( enable, clock, data : in std_logic;
         q : out std_logic );
end d_latch_enab;

architecture behavioral of d_latch_enab is
begin
  infer : process ( enable, clock, data )
  begin
    if ( clock = ’1’ )
    then
      if ( enable = ’1’ )
      then
        q <= data;
      end if;
    end if;
  end process infer;
end behavioral;
```

For Example G-16, no priority is implied.

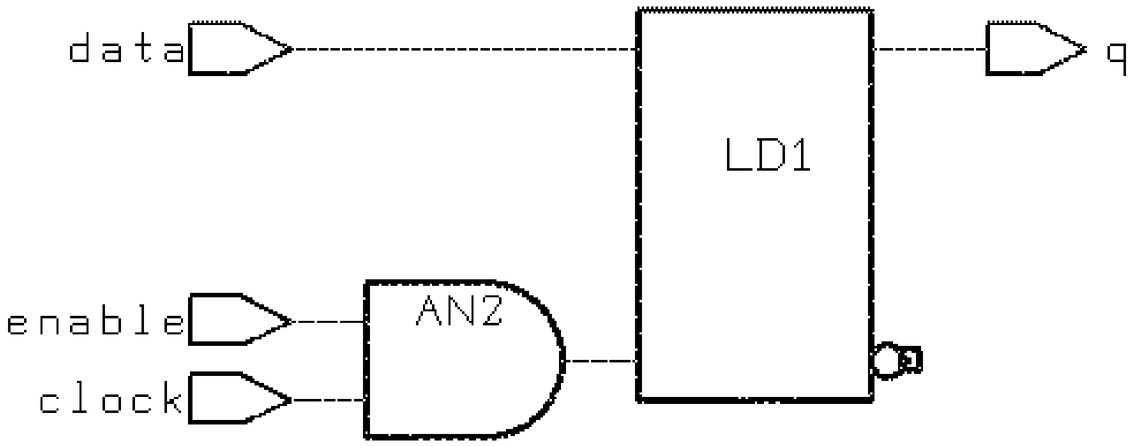
Example G-16   Coding Style B

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity d_latch_enab is
  port ( enable, clock, data : in std_logic;
         q : out std_logic );
end d_latch_enab;

architecture behavioral of d_latch_enab is
begin
  infer : process ( enable, clock, data )
  begin
    if ( clock = ’1’ and enable = ’1’)
      q <= data;
    end if;
  end process infer;
end behavioral;
```

Synthesized Design With Enable and Gated Clock

Figure G-6 shows the synthesized design for the D latch with enable and clock gating resulting from compilation of the code in Example G-15 and Example G-16.

Figure G-6   D Latch With Enable and Gated Clock



D Latch With Enable and Asynchronous Reset

This section shows the VHDL and Verilog code that implements a design that uses a D latch with the enable and asynchronous reset attributes. It includes these subsections:

- VHDL and Verilog Code for a D Latch With Enable and Asynchronous Reset
- Synthesized Design for a D Latch With Enable and Asynchronous Reset

VHDL and Verilog Code for a D Latch With Enable and Asynchronous Reset

To implement a D latch with enable and asynchronous reset in either VHDL or Verilog, you must set the following variables to true:

```
hdlin_report_inferred_modules
hdlin_keep_feedback
```

You must also set the following attribute:

```
attribute async_set_reset of reset : signal is "true";
```

Example G-17 shows the VHDL code for a D latch with enable and asynchronous reset.

Example G-17 VHDL Code for a D Latch With Enable and Asynchronous Reset

```
library IEEE, SYNOPSYS;
use IEEE.STD_LOGIC_1164.all;
use SYNOPSYS.attributes.all;

entity d_latch_enab_async_reset is
  port ( enable, clock, reset, data : in std_logic;
         q : buffer std_logic );
end d_latch_enab_async_reset;

architecture behavioral of d_latch_enab_async_reset is
attribute async_set_reset of reset : signal is "true";
begin
  infer : process ( enable, clock, reset, data )
  variable temp : std_logic;
  begin
    temp := q;
    if ( reset = ’1’ ) then
      q <= ’0’;
    elsif ( clock = ’1’ ) then
      case enable is
        when ’1’ => q <= data;
        when others => q <= temp;
      end case;
    end if;
  end process infer;
end behavioral;
```

Example G-18 shows the Verilog code for a D latch with enable and asynchronous reset.

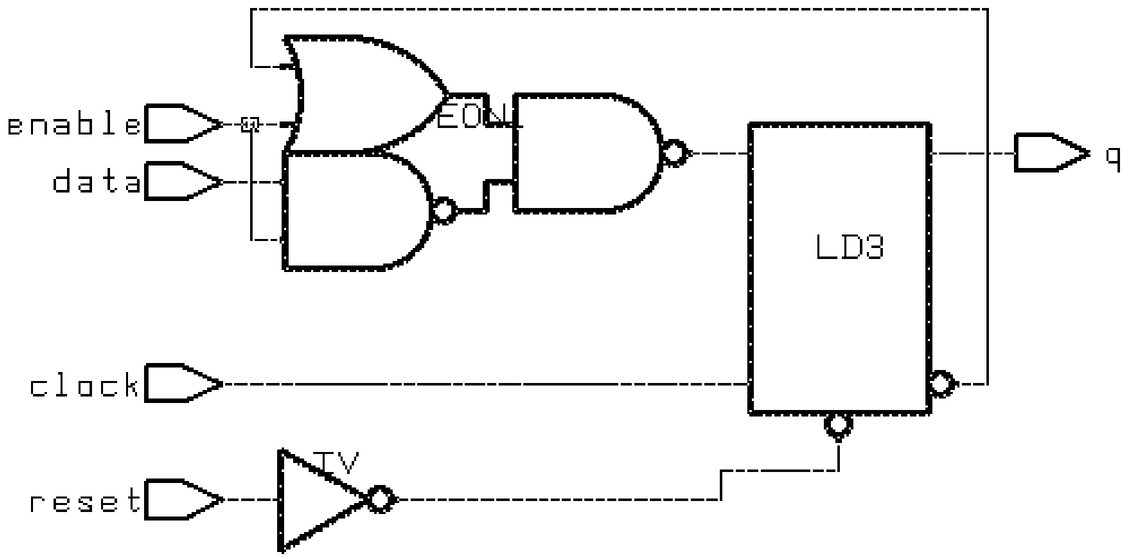
Example G-18   Verilog Code for a D Latch With Enable and Asynchronous Reset

```
module d_latch_enab_async_reset
    (enable, clock, reset, q, data) ;

input enable, clock, reset, data ;
output q ;

// synopsys async_set_reset "reset"

reg q ;

always @(enable or clock or reset or data)
    begin : infer
       if (reset == 1)
         q = 1'b0 ;

       else if (clock == 1)
         begin
            if (enable == 1)
              q = data ;
            else
              q = q ;
         end
    end
endmodule
```

Synthesized Design for a D Latch With Enable and Asynchronous Reset

Figure G-7 shows the synthesized design for the D latch with enable and asynchronous reset resulting from compilation of the code in Example G-17 or Example G-18.

Figure G-7 Synthesized Design for a D Latch With Enable and Asynchronous Reset



D Latch With Enable and Asynchronous Set

This section shows the VHDL and Verilog code that implements a design that uses a D latch with the enable and asynchronous set attributes. It includes these subsections:

- VHDL and Verilog Code for a D Latch With Enable and Asynchronous Set
- Synthesized Design for D Latch With Enable and Asynchronous Set

VHDL and Verilog Code for a D Latch With Enable and Asynchronous Set

To implement a D latch with enable and asynchronous set in either VHDL or Verilog, you must set the following variables to true:

```
hdlin_report_inferred_modules
hdlin_keep_feedback
```

You must also set the following attribute:

```
attribute async_set_reset of set : signal is "true";
```

Example G-19 shows the VHDL code for a D latch with enable and asynchronous reset.

Example G-19   VHDL Code for D Latch With Enable and Asynchronous Set

```
library IEEE, SYNOPSYS;
use IEEE.STD_LOGIC_1164.all;
use SYNOPSYS.attributes.all;

entity d_latch_enab_async_set is
  port ( enable, clock, set, data : in std_logic;
         q : buffer std_logic );
end d_latch_enab_async_set;

architecture behavioral of d_latch_enab_async_set is
attribute async_set_reset of set : signal is "true";
begin
  infer : process ( enable, clock, set, data )
  begin
    if ( set = ’1’ )
    then
      q <= ’1’;
    elsif ( clock = ’1’ )
    then
      if ( enable = ’1’ )
      then
        q<= data;
      else
        q <= q;
      end if;
    end if;
  end process infer;
end behavioral;
```

Example G-20 shows the Verilog code for a D latch with enable and asynchronous set.

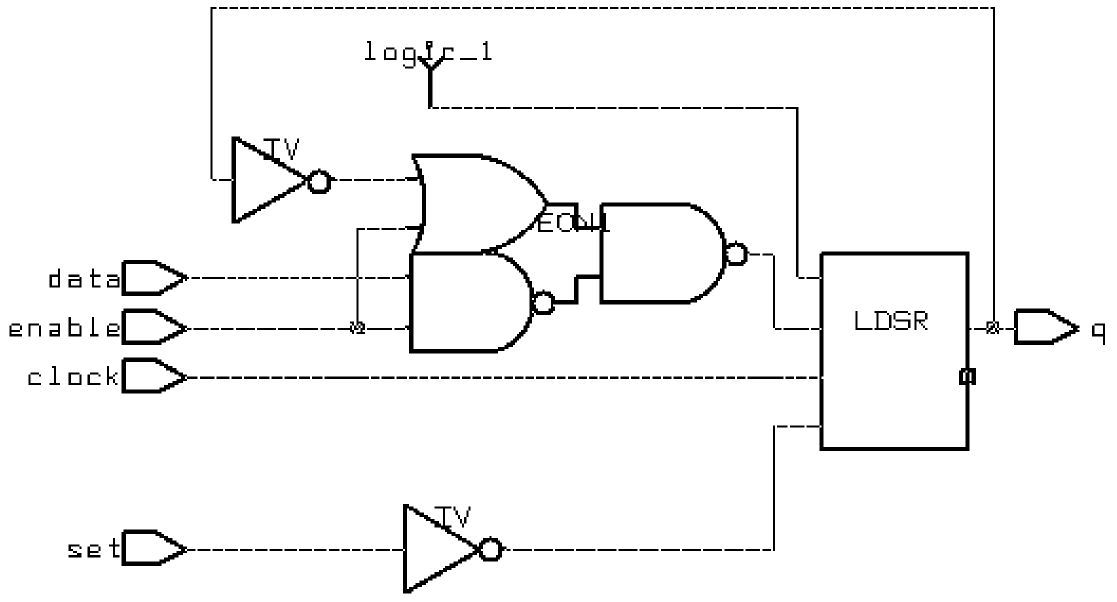
Example G-20   Verilog Code for D Latch With Enable and Asynchronous Set

```
module d_latch_enab_async_set (enable, clock, set, q, data) ;

input enable, clock, set, data ;
output q ;

// synopsys async_set_reset "set"

reg q ;

always @(enable or clock or set or data)
     begin : infer
        if (set == 1)
          q = 1’b1 ;

        else if (clock == 1)
          begin
             if (enable == 1)
               q = data ;
             else
               q = q ;
          end
     end
endmodule
```

Synthesized Design for D Latch With Enable and Asynchronous Set

Figure G-8 shows the synthesized design for the D latch with enable and asynchronous set resulting from compilation of the code shown in Example G-19 or Example G-20.

Figure G-8 Synthesized Design for D Latch With Enable and Asynchronous Set



D Latch With Enable and Asynchronous Set and Reset

This section shows the VHDL and Verilog code that implements a design that uses a D latch with the enable and asynchronous set and reset attributes. It includes these subsections:

- VHDL and Verilog Code for D Latch With Enable and Asynchronous Set and Reset
- Synthesized Design for D Latch With Enable and Asynchronous Set and Reset

VHDL and Verilog Code for D Latch With Enable and Asynchronous Set and Reset

To implement a D latch with enable and asynchronous set and reset in either VHDL or Verilog, you must set the following variables to true:

```
hdlin_report_inferred_modules
hdlin_keep_feedback
```

You must also set the following attributes:

```
attribute async_set_reset of set : signal is "true";
attribute one_hot of set, reset: signal is "true";
```

Example G-21 shows the VHDL code for a D latch with enable and asynchronous set and reset.

Example G-21 VHDL Code for D Latch With Enable and Asynchronous Set and Reset

```
library IEEE, SYNOPSYS;
use IEEE.STD_LOGIC_1164.all;
use SYNOPSYS.attributes.all;

entity d_latch_enab_async_set_reset is
  port ( enable, clock, set, reset, data : in std_logic;
         q : buffer std_logic );
end d_latch_enab_async_set_reset;

architecture behavioral of d_latch_enab_async_set_reset is
attribute async_set_reset of set, reset : signal is "true";
attribute one_hot of set, reset : signal is "true";
begin
  infer : process (enable,clock, set, reset, data)
  begin
    if ( set = ’1’ ) then
      q <= ’1’;
    elsif ( reset = ’‘1’ ) then
      q <= ’0’;
    elsif ( clock = ’1’ ) then
      if ( enable = ’1’) then
        q <= data;
      else
        q <= q;
      end if;
    end if;
  end process infer;
end behavioral;
```

Example G-22 shows the Verilog code for a D latch with enable and asynchronous set and reset.

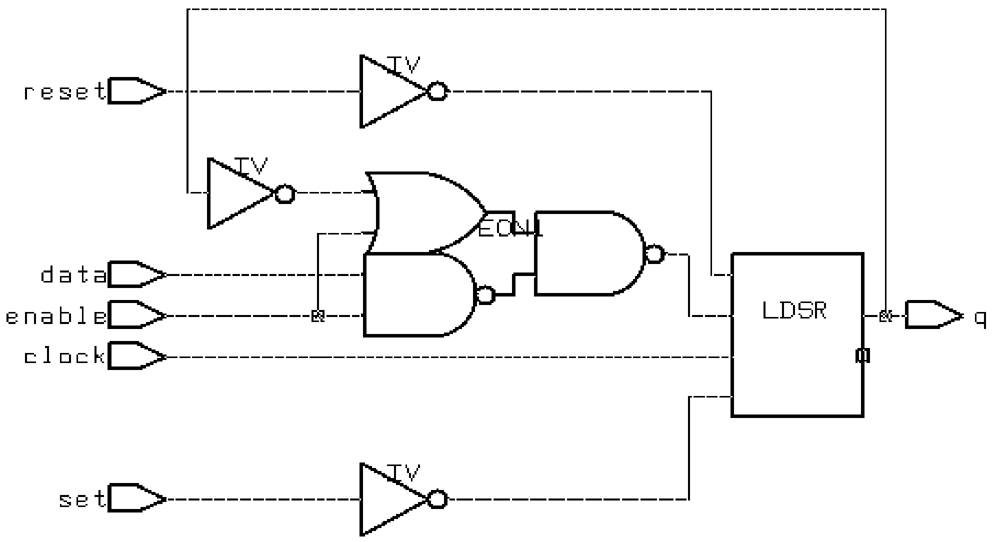
Example G-22 Verilog Code for D Latch With Enable and Asynchronous Set and Reset

```
module d_latch_enab_async_set_reset (enable, clock, set,
reset, q, data);
input enable, clock, set, reset, data ;
output q ;
// synopsys async_set_reset "set, reset"”
// synopsys one_hot "set, reset"
reg q ;

always @(enable or clock or set or reset or data)
     begin : infer
        if (reset == 1)
          q = 1’b0 ;
        else if (set == 1)
          q = 1’b1 ;
        else if (clock == 1)
          begin
             if (enable == 1)
               q = data ;
             else
               q = q ;
          end
     end
endmodule
```

Synthesized Design for D Latch With Enable and Asynchronous Set and Reset

Figure G-9 shows the synthesized design for the D latch with enable and asynchronous set and reset resulting from compilation of the code shown in Example G-21 or Example G-22.

Figure G-9 Synthesized Design for D Latch With Enable and Asynchronous Set and Reset



Index